```
//+----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
// FileName: AutoLock.h
// Purpose:    Multithreaded object wrappers for process object locking
//
// Contains:
//                  AutoLock Class
//                        CAutoLockT              -- Helper class used to Auto-Lock/Unlock
//                                                   classes derived from CLockableXXX classes
//                                                   listed below or from ATL's CComObjectRoot
//                                                   class.
//
//                  Lock Classes
//                        CLockableCS             -- Critical Section base class
//                        CLockableMutex          -- Mutex base class
//                        CLockableSWMR      a    -- Single Writer, Multiple Reader base class
//
//                  Exception Classes
//                        CAutoLockExc            -- CLockableMutex::Lock() thrown exception
//                        CAutoLockTimeoutExc     -- CLockableMutex::Lock() thrown t/o exception
//     History:
//     [0]         12-15-2004     Arjay Hawco -- Method Created
//-----------------------------------------------------------------------------
//+----------------------------------------------------------------------------
#pragma once

#include <atlbase.h> // Required for ATLASSERT macro
//#include <atlstr.h>

namespace AutoLock
{
      // SWMR enum wait lock types
      enum _LOCKTYPE { LT_WAITTOREAD = 0, LT_WAITTOWRITE = 1 } ;

//+----------------------------------------------------------------------------
// Class:      CAutoLockT< lockableclass >
//
// Purpose: Helper class used to lock a LockableObject [derived] class
//            on construction and unlock it on destruction.    The indicated object
//             will be locked from the time the CAutoLock object is created until
```

```
//          the CAutoLock object goes out of scope (or is otherwise destroyed).
//          The class supplied as a template argument must support accessible
//          functions named Lock() and Unlock().
//
//          For example, anything derived from CLockableXXX classes below, or
//          from ATL's CComObjectRoot is autolockable.
//
// Constructors:
//          CAutoLockT( T )                     -- Critical Section ctor
//+---------------------------------------------------------------------------
template<class T>
class CAutoLockT
{
// Attributes
private:
    T* m_pObject;           // the locked object

// Ctor/dtor
public:
    // Critical Section Ctor
    inline CAutoLockT( T* pObject ) throw()
        : m_pObject(pObject)
    {
        ATLASSERT( NULL != pObject );
        m_pObject->Lock();
    }

    // Mutex Ctor
    inline CAutoLockT( T* pObject, DWORD dwTimeout )
        : m_pObject(pObject)
    {
        ATLASSERT( NULL != pObject );
        m_pObject->Lock( dwTimeout );
    }

    // SRMR Ctor
    inline CAutoLockT( T* pObject,
                       const _LOCKTYPE uLockType,
                       LPCTSTR szOperation = NULL ) throw()
        : m_pObject(pObject)
```

```
    {
        ATLASSERT( NULL != pObject );
        m_pObject->Lock( uLockType, szOperation );
    }

    // dtor
    inline ~CAutoLockT()
    {
        m_pObject->Unlock();
    }
};

//+---------------------------------------------------------------------------
// Class:           CAutoLockWaitExc
//
// Purpose:         CLockableMutex::Lock() method will throw this exception when
//                  an error occurs.
//
// Comment:         NOTE: Callers should catch this exception.
//+---------------------------------------------------------------------------
class CAutoLockWaitExc
{
// Ctor
public:
    CAutoLockWaitExc( DWORD dwWaitResult, DWORD dwError )
        : m_dwWaitResult( dwWaitResult )
        , m_dwError( dwError ) {}

// Public Accessors
public:
    inline    const DWORD GetWaitResult() { return m_dwWaitResult; };
    inline    const DWORD GetLastError() { return m_dwError; };

// Attributes
private:
    DWORD m_dwWaitResult;       // WaitForSingleObject return
    DWORD m_dwError;            // GetLastError return code
};

//+---------------------------------------------------------------------------
```

```
// Class:             CAutoLockExc
//
// Purpose:                 CLockableMutex::Lock() method will throw this exception when
//                          an error occurs.
//
// Comment:                 NOTE: Callers should catch this exception.
//+---------------------------------------------------------------------------
class CAutoLockExc
{
// Ctor
public:
      CAutoLockExc( DWORD dwError )
            : m_dwError( dwError ) {}

// Public Accessors
public:
      inline      const DWORD GetLastError() { return m_dwError; };

// Attributes
private:
      DWORD m_dwError;                  // GetLastError return code
};

//+---------------------------------------------------------------------------
// Embedded Class:    CAutoLockTimeoutExc
//
// Purpose:                 CLockableMutex::Lock() method will throw this timeout exception
//                          when the timeout period has been exceeded.
//
// Comment:                 NOTE: Callers should catch this exception.
//+---------------------------------------------------------------------------
class CAutoLockTimeoutExc
{
// Ctor
public:
      CAutoLockTimeoutExc( DWORD dwTimeout ) : m_dwTimeout( dwTimeout ) {}

// Public Accessors
public:
      inline      const DWORD GetTimeout() { return m_dwTimeout; };
```

```
// Attributes
private:
	DWORD m_dwTimeout;		// Timeout value (although passed in
					// to the Lock method, we store it
					// here and make it available in the
					// exception for convenience
};

//+---------------------------------------------------------------------------
// Class:	CLockableCS
//
// Purpose: This class is intended to be used as a base class for a
//		multithreaded Critical Section lockable object.
//
//+---------------------------------------------------------------------------
class CLockableCS
{
// Methods
public:
	// Ctor / Dtor
	inline CLockableCS() throw()	{ InitializeCriticalSection( &m_CS ); }
	inline ~CLockableCS() throw()	{ DeleteCriticalSection( &m_CS ); }

	// Lock / Unlock
	inline void Lock() throw()		{ EnterCriticalSection( &m_CS ); }
	inline void Unlock() throw()	{ LeaveCriticalSection( &m_CS ); }

// Attributes
private:
	CRITICAL_SECTION m_CS;		// Internal Critical Section
};

//+---------------------------------------------------------------------------
// Class:	CLockableMutex
//
// Purpose: This class is intended to be used as a base class for a
//		Mutex lockable object.
//
// Comment:	NOTE: The lock method for this class will throw a
```

```
//                  CMutexLockExc exception (see below for declaration).
//                  Callees should catch this exception when calling the Lock method.
//
// Scope:  Single or Multiple Processes
//+---------------------------------------------------------------------------

class CLockableMutex
{
public:
      inline CLockableMutex( LPCTSTR szName ) //throw()
            : m_hMutex( ::CreateMutex( NULL, FALSE, szName ) )
                  , m_bAlreadyExists( FALSE )
      {
            // If the call has succeeded, check if the named mutex has already existed
            if( NULL != m_hMutex )
            {
                  m_bAlreadyExists = ( ERROR_ALREADY_EXISTS == ::GetLastError( ) );
            }
            else
            {
                  // The call has failed.
                  if( ERROR_ACCESS_DENIED == ::GetLastError( ) )
                  {
                        // We'll get the Access denied error when the mutex
                        // exists and the user does not have sufficient permissions
                        // to create an existing mutex (as in above).
                        // So we retry using OpenMutex

                        if( NULL == ( m_hMutex = ::OpenMutex( MUTEX_MODIFY_STATE,
                                                              FALSE,
                                                              szName ) ))
                        {
                              // OpenMutex failed also, so throw an exception
                              throw CAutoLockExc( ::GetLastError( ) );
                        }
                        else
                        {
                              m_bAlreadyExists = TRUE;
                        }
                  }
```

```
            else
            {
                // Other unknown error
                throw CAutoLockExc( ::GetLastError( ) );
            }
        }
}

inline ~CLockableMutex() throw()
{
        if(m_hMutex) ::CloseHandle( m_hMutex );
}

//+---------------------------------------------------------------------
//      Method:         AlreadyExists
//      Purpose:        Use this method when you want to use a mutex to limit
//                      an executable to a single instance.    Declare a CLockableMutex
//                      variable and check this param to find out if it existed
//                      prior to instantiating this class.
//
//      Params:         void
//      Return:         void
//
//      Comments:       See msdn docs on CreateMutex.    This
//+---------------------------------------------------------------------
const BOOL AlreadyExists() { return m_bAlreadyExists; };

//+---------------------------------------------------------------------
//      Method:         Lock
//      Purpose:  Used by the CAutoLockT class to limit inter-process
//                      access.
//
//      Params:         ms Timeout
//      Return:         void
//
//      Comments:       Throws CTimeoutException or CException exceptions
//+---------------------------------------------------------------------
inline void Lock( DWORD dwTimeout )
{
        DWORD dwWaitResult = 0;
```

```
            dwWaitResult = ::WaitForSingleObject( m_hMutex, dwTimeout );
            switch( dwWaitResult )
            {
            case WAIT_OBJECT_0:
                  break;
            case WAIT_TIMEOUT:
                  throw CAutoLockTimeoutExc( dwTimeout );
                  break;
            default:
                  throw CAutoLockWaitExc( dwWaitResult, ::GetLastError() );
            }
      }

      //+-------------------------------------------------------------------
      //    Method:         Unlock
      //    Purpose:  Used by the CAutoLockT class (CAutoLockT::dtor) to
      //                    release the mutex
      //
      //    Params:         void
      //    Return:         void
      inline void Unlock() throw()
      {
            if(m_hMutex)
            {
                  ::ReleaseMutex( m_hMutex );
            }
      }

private:
      HANDLE      m_hMutex;     // Name Mutex Handle
      BOOL        m_bAlreadyExists;     // TRUE if named mutex already exists
};

//+-----------------------------------------------------------------------
// Class:     CLockableSWMR
//
// Purpose: This class is intended to be used as a base class for a
//                SWMR (Single Writer, Multiple Reader) lockable object.
//
// Acknowledgements:
```

```
//          This class is base on work by Jeffrey Richter in his book
//          "Programming Applications for Microsoft Windows"
//
// Usage:
//          Create a class derived from this and another resource class (such as
//          an std::list<> class) when you have a scenario where the list seldom
//          gets updated, but is frequently read by multiple threads.    Under this
//          scenario, this class would be more efficient than the CLockableCS
//          class because multiple threads could read simultaneously whereas a
//          CLockableCS implementation would only allow a single thread to read
//          or write.    Conversely, because this class makes use of the kernel
//          semaphore object, it is not as efficient a CLockableCS under single
//          read thread scenarios.
//
//          To Lock to Read:
//                  CAutoLock< CLockableSWMR_Derived > lock( &m_SWMR,
LT_WAITTOREAD );
//
//          To Lock to Write:
//                  CAutoLock< CLockableSWMR_Derived > lock( &m_SWMR,
LT_WAITTOWRITE );
//
// Public Methods:
//     Lock( enumLockType, ... )
//     Unlock()
//
// Private Methods:
//     WaitToRead(...)
//     WaitToWrite(...)


// Scope:  Single Process
//+---------------------------------------------------------------------------
class CLockableSWMR
{
public:
      CLockableSWMR( ) throw()
        : m_hSemReaders( CreateSemaphore(NULL, 0, MAXLONG, NULL) )
            , m_hSemWriters( CreateSemaphore(NULL, 0, MAXLONG, NULL) )
            , m_nWaitingReaders( 0 )
```

```
            , m_nWaitingWriters( 0 )
            , m_nActive( 0 )
//          , m_sName( _T("") )
        {
        }

        CLockableSWMR( LPCTSTR szName ) throw()
         : m_hSemReaders( CreateSemaphore(NULL, 0, MAXLONG, NULL) )
            , m_hSemWriters( CreateSemaphore(NULL, 0, MAXLONG, NULL) )
            , m_nWaitingReaders( 0 )
            , m_nWaitingWriters( 0 )
            , m_nActive( 0 )
//          , m_sName( szName )
        {
        }

        ~CLockableSWMR() throw()
        {
            m_nWaitingReaders = 0;
            m_nWaitingWriters = 0;
            m_nActive = 0;
            CloseHandle(m_hSemReaders);
            CloseHandle(m_hSemWriters);
        }

// Attributes
private:
        CLockableCS  m_InternalCS;      // Ensures exclusive access
                                        // to member variables
        HANDLE          m_hSemReaders;      // Readers wait on if a writer has access
        HANDLE          m_hSemWriters;      // Writers wait on if a reader has access
        int             m_nWaitingReaders;  // # of readers waiting
        int             m_nWaitingWriters;  // # of writers waiting
        int             m_nActive;          // # of threads currently w/access
                                            // (0 == no threads,
                                            // >0 == # of readers,
                                            // -1 == 1 writer)

//      CString    m_sName;                 // Debug diagnostic string
```

```
        // Thread tracking enum
        enum _ACTIVETHREADS { AT_ONEWRITER = -1, AT_NONE = 0 };

// Operations
public:
        //+-------------------------------------------------------------------
        //      Method:         Lock
        //      Purpose:  Used by the CAutoLockT class to provide thread safe access.
        //
        //      Params:         enumLockType, szOperation (debug operation string)
        //      Return:         void
        //+-------------------------------------------------------------------
        void Lock(const _LOCKTYPE enumLockType,
                                LPCTSTR szOperation = NULL ) throw()
        {
                switch(enumLockType)
                {
                case LT_WAITTOREAD:
                        WaitToRead( szOperation );
                        break;
                case LT_WAITTOWRITE:
                        WaitToWrite( szOperation );
                        break;
                default:
                        ATLASSERT(0);
                }
        }

        //+-------------------------------------------------------------------
        //      Method:         Unlock
        //      Purpose:  Used by the CAutoLockT class (CAutoLockT::dtor) to
        //                      release the CS
        //
        //      Params:         void
        //      Return:         void
        //+-------------------------------------------------------------------
        void Unlock() throw()
        {
                HANDLE hSem = NULL;     // Assume no threads are waiting
                long lCount = 1;     // Assume only 1 waiter wakes;
```

```
                                // always true for writers

{       // Lock scope

        // Ensure exclusive access to member variables
        CAutoLockT< CLockableCS > lock( &m_InternalCS );

        //ATLTRACE( _T("\t\t%s: Unlock - m_nActive: %d"),
        //                m_sName,
        //                m_nActive);

        if (m_nActive > 0)
        {
                // Readers have control so a reader must be done
                m_nActive--;
        }
        else
        {
                // Writers have control so a writer must be done
                m_nActive++;
        }

        //ATLTRACE( _T("\t\t%s: Unlock - m_nActive: %d\n"),
        //                m_sName,
        //                m_nActive);

        if(m_nActive == 0)
        {
                // No thread has access, who should wake up?
                // NOTE: It is possible that readers could never get access
                //              if there are always writers wanting to write

                if (m_nWaitingWriters > 0)
                {
                        // Writers are waiting and they take priority over readers
                        m_nActive = -1;                 // A writer will get access
                        hSem = m_hSemWriters;       // Writers wait on this semaphore
                        m_nWaitingWriters--;    // One less writer will be waiting

                        // NOTE: The semaphore will release only 1 writer thread
```

```
                    //ATLTRACE( _T("\b\t\t\t\t%s: Unlock - m_nWaitingWriters: %d\n"),
                    //      m_sName,
                    //      m_nWaitingWriters);
                }
                else if (m_nWaitingReaders > 0)
                {
                    // Readers are waiting and no writers are waiting
                    m_nActive = m_nWaitingReaders;// All readers get access
                    m_nWaitingReaders = 0;// No readers will be waiting
                    hSem = m_hSemReaders;       // Readers wait on this semaphore
                    lCount = m_nActive;         // Semaphore releases all readers

                    //ATLTRACE( _T("\b\t\t\t\t%s: Unlock - m_nWaitingReaders: %d\n"),
                    //      m_sName,
                    //      m_nWaitingReaders);
                }
                else
                {
                    // There are no threads waiting at all;
                    // no semaphore gets released
                    //ATLTRACE( _T("\b\t\t\t\t%s: Unlock - m_nR/W = 0\n"),
                    //                  m_sName);
                }
            }
        } // End of CS Lock scope

        if (NULL != hSem)
        {
            // Some threads are to be released
            ReleaseSemaphore(hSem, lCount, NULL);
        }
    }

    // Implementation
private:
    void WaitToRead( LPCTSTR szOperation )
    {
        BOOL bResourceWritePending = FALSE;

        {   // Lock scope
```

```
            // Ensure exclusive access to member variables
            CAutoLockT< CLockableCS > lock( &m_InternalCS );

            // Are there writers waiting or is a writer writing?
            bResourceWritePending
                = (m_nWaitingWriters || (m_nActive < AT_NONE));

            if (bResourceWritePending)
            {
                // This reader must wait,
                // increment the count of waiting readers
                m_nWaitingReaders++;
            }
            else
            {
                //ATLTRACE( _T("%s: WaitToRead - m_nActive: %d\tOp:%s\n"),
                //          m_sName,
                //          m_nActive,
                //          ((szOperation) ? szOperation : _T("")));
                // This reader can read, increment the count of active readers
                m_nActive++;
            }

        } // End of Lock scope

        if (bResourceWritePending)
        {
            //ATLTRACE( _T("%s: WaitToRead - m_nWaitingReaders: %d\tOp:%s\n"),
            //          m_sName,
            //          m_nWaitingReaders,
            //          ((szOperation) ? szOperation : _T("")));
            // This thread must wait
            WaitForSingleObject(m_hSemReaders, INFINITE);
        }
    }

    void WaitToWrite( LPCTSTR szOperation )
    {
        BOOL bResourceOwned = FALSE;
```

```
        {     // Lock scope

              // Ensure exclusive access to member variables
              CAutoLockT< CLockableCS > lock( &m_InternalCS );

              // Are there any threads accessing the resource?
              bResourceOwned = (AT_NONE != m_nActive);

              if(bResourceOwned)
              {
                    // This writer must wait;
                    // increment the count of waiting writers
                    m_nWaitingWriters++;
              }
              else
              {
                    //ATLTRACE( _T("%s: WaitToWrite %s - Okay TID: %d\n"),
                    //          m_sName,
                    //          ((szOperation) ? szOperation : _T("")),
                    //          GetCurrentThreadId());
                    // This writer can write, prevent access to others
                    m_nActive = AT_ONEWRITER;
              }

        }     // End of lock scope

        if (bResourceOwned)
        {
              //ATLTRACE( _T("%s: WaitToWrite %s - m_nWaitingWriters: %d TID: %d\n"),
              //          m_sName,
              //          ((szOperation) ? szOperation : _T("")),
              //          m_nWaitingWriters,
              //          GetCurrentThreadId());

              // This thread must wait
              WaitForSingleObject(m_hSemWriters, INFINITE);
        }
    }
};
```

```
} // End of namespace AutoLock
```