```
#include <windows.h>
#include <string>
#include <sstream>
#include "serial.h"
#include "packet.h"
#include "exceptions.h"
#include "utils.h"
#include "message.h"

using namespace std;

// ========================= Serial ========================= //

/*
Constructs the serial port class
*/
Serial::Serial(): connected_(false), packetAvailable_(false) {
    InitializeCriticalSection(&portGuard_);
}

/*
Gets a new packet from the serial port.
Note: this function is called by Serial::thread!
*/
DWORD WINAPI Serial::thread(PVOID pvoid) {
    Serial *s = (Serial*) pvoid;

    bool live = true;

    bool success = false;
    char inbuff[256];
    DWORD nBytesRead, dwEvent;
    COMSTAT comstat;
    COMMTIMEOUTS timeOuts;
    OVERLAPPED osRead = {0};
    osRead.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

    //Set the total timeout interval
    memset(&timeOuts, 0, sizeof(timeOuts));
    timeOuts.ReadTotalTimeoutMultiplier = 5;
    timeOuts.ReadTotalTimeoutConstant = 50;
    SetCommTimeouts(s->hComm_, &timeOuts);

    SetCommMask(s->hComm_, EV_RXCHAR);


    char err = 0;

    while(live) {   // wait for packet to be complete
```

```
        if(WaitCommEvent(s->hComm_, &dwEvent, NULL)) {
            ClearCommError(s->hComm_, NULL, &comstat);
            if ((dwEvent & EV_RXCHAR) && comstat.cbInQue) {
                if (ReadFile(s->hComm_, &inbuff, min(255, comstat.cbInQue), &nBytesRead,
&osRead)) {
                    if (nBytesRead) {
                        live = s->foundString(string(inbuff, nBytesRead));
                    }
                }
            }
        } else {
            err = (char)GetLastError();
            s->sendString(&err, 1);
            myvoid(err);
        }

        ResetEvent(osRead.hEvent);
    }
    PurgeComm(s->hComm_, PURGE_RXCLEAR);
    return 0L;
}

bool Serial::foundString(string str) {
    static HANDLE found = CreateEvent(NULL, FALSE, FALSE, PACKET_FOUND_EVENT);
    static HANDLE listen[3] = {
        listen[0] = CreateEvent(NULL, FALSE, FALSE, GET_NEW_PACKET_EVENT),
        listen[1] = CreateEvent(NULL, FALSE, FALSE, GLOBAL_DIE_EVENT),
        listen[2] = CreateEvent(NULL, FALSE, FALSE, SERIAL_THREAD_DIE_EVENT)
    };
    for (int i=0; i < str.length(); ++i) {
        if (packet_.length() == 0) {
            if (str[i] == SOH) {
                packet_.append(str[i]);
            }
        } else {
            packet_.append(str[i]);
            if (packet_.complete()){
                packetAvailable_ = true;
                SetEvent(found);
                if (WaitForMultipleObjects(3, listen, FALSE, INFINITE) == 1) {
                    return false;
                    break;
                }
                packet_.clear();
            }
        }
    }
    return true;
}
```

```
/*
Tries to get a packet within a certain timeframe
@throws TIMEOUT_EXCEPTION if no packet found
@param timeout - max time to wait for a packet
@returns the new packet
*/
Packet Serial::getPacket(int timeout) {
    timeout = 100;
    static HANDLE found = CreateEvent(NULL, FALSE, FALSE, PACKET_FOUND_EVENT);
    static HANDLE get   = CreateEvent(NULL, FALSE, FALSE, GET_NEW_PACKET_EVENT);
    if (!packetAvailable_) {
        //WaitForSingleObject(found, timeout);  // give the serial thread time to find a
packet
        WaitForSingleObject(found, 100);
    }
    ResetEvent(found);
    ENSURE_BOOL_THROW(packetAvailable_, TIMEOUT_EXCEPTION); // check if a packet has
been found yet
    Packet p(packet_);
    SetEvent(get);
    return p;
}

bool Serial::sendPacket(Packet& packet){
    Sleep(1);
    return sendString(packet.toString().c_str(), packet.length());
}

bool Serial::sendString(const char *str, size_t len) {
    bool retVal = true;
    EnterCriticalSection(&portGuard_);
    // Create Overlap
    OVERLAPPED osWrite = {0};
    DWORD dwWritten;
    //Create this write operation's OVERLAPPED structure's hEvent.
    osWrite.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
    if(osWrite.hEvent == NULL) {
        retVal =  false;
    } else if(WriteFile(hComm_, str, (DWORD)len, &dwWritten, &osWrite)){
        CloseHandle(osWrite.hEvent);
    } else if (GetLastError() != ERROR_IO_PENDING) {
        retVal = false;
    }
    // Done
    LeaveCriticalSection(&portGuard_);
    return retVal;
}
```

```
bool Serial::connect(LPCTSTR port) {
    ENSURE_BOOL(!connected_);
    bool success = false;

    EnterCriticalSection(&portGuard_);

    GetCommState(hComm_, &dcb_);
    // setup settings
    FillMemory(&dcb_, sizeof(dcb_), 0);
    dcb_.BaudRate = CBR_9600;
    dcb_.Parity = NOPARITY;
    dcb_.StopBits = ONESTOPBIT;
    dcb_.ByteSize = 8;

    // Enables the COM port configuration
    SetCommState(hComm_, &dcb_);

    // connect
    if (port == NULL) {
        Message::ChoosePort();
    } // Only change made here was to change FILE_FLAG_OVERLAPPED to
FILE_ATTRIBUTE_NOMAL
    else if((hComm_ = CreateFile(port, GENERIC_READ | GENERIC_WRITE, 0,
                NULL, OPEN_EXISTING, FILE_FLAG_OVERLAPPED, NULL)) == // changed from
FILE_FLAG_OVERLAPPED
                INVALID_HANDLE_VALUE) {
        Message::ErrorFailedPort();
    }
    else if(!SetCommState(hComm_, &dcb_)) {
        Message::FailedSavePort();
    }
    else {
        success = true;
    }

    // clean up
    LeaveCriticalSection(&portGuard_);
    connected_ = success;
    CreateThread(NULL, 0, Serial::thread, this, 0, NULL);
    return success;
}

bool Serial::disconnect() {
    HANDLE die = CreateEvent(NULL, FALSE, FALSE, SERIAL_THREAD_DIE_EVENT);
    SetEvent(die);
    ENSURE_BOOL(connected_);
    PurgeComm(hComm_, PURGE_FLAGS);
    CloseHandle(hComm_);
    connected_ = false;
```

```
        return true;
    }

    bool Serial::connected() {
        return connected_;
    }
```