```
/*
*****************************************************************************
* CommOut.cpp                                                               *
* Purpose: Sending completed packets from the buffer to the serial port.    *
* Additionally, CommOut handles the RVI process of receiving data and       *
* sending that to the GUI.                                                  *
* Author: Max Wardell                                                       *
* Version: 1.0                                                              *
*****************************************************************************
*/

#include "CommOut.h"

/*Purpose: Constructor for the CommOut object.
* Parameters: Buffer *buffer: A pointer to the Buffer object.
*             Serial *serial: A pointer to the Serial object.
*/
CommOut::CommOut(Buffer *buffer, Serial *serial, Controller *gui): buffer_(buffer),
                                                                   serial_(serial),
                                                                   gui_(gui) {
}

////Outbuff gets populated, this function is called.
///*Purpose: Connects to the receiver*/
void CommOut::ConnectClient() {
    //TODO: Try this up here, then delete it.
    serial_->sendPacket(Packet(ENQ));
    Sleep(100);
    SendPacket(); //Send the packet
}

//Outbuff gets populated, this function is called.
/*Purpose: Connects to the receiver*/
////void CommOut::ConnectClient() {
////    //TODO: Try this up here, then delete it.
////    serial_->sendPacket(Packet(ENQ));
////
////    while(true) {
////        //serial_->sendPacket(Packet(ENQ));
////        try {
////            //packet_ = serial_->getPacket(TIMEOUT_TIME);//Grab the packet from the
serial port.
////            packet_ = serial_->getPacket(100);
////        }
////        catch (const int i) {         //Never received a packet, timeout.
////            //SetEvent(gotoIdle);
////            ENSURE_EXCEPTION(i, TIMEOUT_EXCEPTION);
////            throw GOTO_IDLE_EXCEPTION;
////        }
```

```
////        if(!packet_.valid())
////            continue; //Packet was invalid, wait for another
////        if(packet_.flags() == ACK0)    //Packet is an ACK0,
////            break; //Break out of loop and send packet.
////        if(packet_.flags() == ENQ) {
////            Sleep(1000);
////            throw GOTO_IDLE_EXCEPTION;
////        }
////    }
////    SendPacket(); //Send the packet
////}

/*Purpose: Sends the packet*/
void CommOut::SendPacket() {
    while(!buffer_->empty()) { //TODO: Change this condition
        serial_->sendPacket(buffer_->peek()); //SEND PACKET
        buffer_->pop();
    }
    Sleep(100);
    serial_->sendPacket(Packet(NTS)); //Send a NTS packet and
}

/*Purpose: Sends the packet*/
//void CommOut::SendPacket() {
//  int nackCount = 0, torCount = 0;
//  while(true) { //TODO: Change this condition
//      if(buffer_->empty()) {  //If the buffer is empty,
//          serial_->sendPacket(Packet(NTS)); //Send a NTS packet and
//          throw GOTO_IDLE_EXCEPTION; //break out of the Sending process
//      }
//      serial_->sendPacket(buffer_->peek()); //SEND PACKET
//      try {
//          /*Grab the control packet from the serial port. Possibilities:
//            ACK1 - Got the packet, keep sending them.
//            ACK0 - Got the packet, return to IDLE
//            NACK - Didn't receive the packet, resend. */
//          packet_ = serial_->getPacket(TIMEOUT_TIME);
//      }
//      catch (...) { //Never received a packet, timeout.
//          torCount++;
//          continue;
//          //ENSURE_EXCEPTION(i, TIMEOUT_EXCEPTION);
//          //throw GOTO_RESET_EXCEPTION;
//      }
//      if(packet_.valid()) {
//          if(packet_.flags() == NACK) {
//              nackCount++;
//          } else if(nackCount >= MAX_NCOUNT) { //NACK count maxed out or received an
ACK0; go back to IDLE.
```

```
//              throw GOTO_IDLE_EXCEPTION;
//          } else if(packet_.flags() == ACK0) {
//              buffer_->pop();
//              throw GOTO_IDLE_EXCEPTION;
//          } else if(torCount >= MAX_TCOUNT) {            //Timouts maxed out, reset
connection.
//              throw GOTO_RESET_EXCEPTION;
//          } else if(packet_.flags() == ACK1) {
//              buffer_->pop();
//              nackCount = 0;
//              torCount = 0;
//          } else if(packet_.flags() == RVI) { //Received an RVI; switch roles!
//              RVIProcess();
//          }
//      }
//  }
//}

/*Purpose: Receives packets when in RVI mode; displays to GUI.*/
void CommOut::RVIProcess() { //Occurs when an RVI is sent
    int timeoutCount = 0, nackCount = 0;
    buffer_->send(ACK1); //Send an acknoledgement, letting the sender know you received
the RVI
    //Connection Confirmed
    while(true) {
        try {
            packet_ = serial_->getPacket(TIMEOUT_TIME); //Grab the packet sent
        } catch (const int i) {
            ENSURE_EXCEPTION(i, TIMEOUT_EXCEPTION);
            if(++timeoutCount >= MAX_TCOUNT)
                throw GOTO_IDLE_EXCEPTION;
            continue;
        }
        if(packet_.valid()) { //If the packet is valid,
            // TODO: update function
            gui_->DisplayReceivedText(packet_.data().c_str());
            //serial_->sendPacket(Packet(ACK1));
            if(packet_.flags() == RVI) //If the packet is an RVI (second one)
                throw GOTO_IDLE_EXCEPTION;
            nackCount = 0; //Received a valid packet, reset NACK Counter
        }
        else { //If the packet is invalid,
            serial_->sendPacket(Packet(NACK));
            if(++nackCount >= MAX_NCOUNT) { //Keep count of how many NACKs you receive
in a row.
                throw GOTO_IDLE_EXCEPTION;
            }
        }
    }
```

```
}
```