ENSC 351: Real-time and Embedded Systems

chap 3

Craig Scratchley, Fall 2021

Multipart Project Part 2

Please continue working with a partner.

In part 1 of the project you programmed some code for the YMODEM protocol that generated blocks with CRC16s and responded to (imagined) 'C' characters, ACKs and NAKs. In part 2 we will expand on that code and write additional code to receive blocks and code to hook in a simulator for a communication medium comprising a pair of telephone modems and the telephone system between them.

In the function *Ensc351Part2()* of the provided code, I create at real-time priority 70 a thread identified via *term2Thrd*. That created thread will represent a communication terminal program identified as *term2* (terminal 2). The primary thread will serve as *term1* (terminal 1), and runs at real-time priority 60. Create another thread, identified via *mediumThrd*, for a "kind" medium and have it execute the function *mediumFunc*(). The kind medium should run at real-time priority 40.

We want to send one or more files using the CRC16 YMODEM protocol from *term2Thrd* via the kind medium to the primary thread. That is, *term2Thrd* will generate blocks and send the blocks to the kind medium for forwarding on to the receiver running in the context of the primary thread. I am providing a template project, which contains or will contain elements of the part 1 solution, to use for this part 2 of the project.

The template code as I am giving it to you should compile, run, and actually transfer a file. However, not much checking is done to see what bytes are actually being received on each end. Once the so-called "kind medium" is hooked up, the receiver will not correctly receive the file being sent.

Your mission, should you choose to accept it, is to improve the code so that files can be correctly transferred even when the medium is hooked up. You will need to improve the YMODEM code so that characters like 'C', ACKs and NAKs are properly sent, received, and processed. In general, changes will need to be made where indicated by the markers "*****" in comments in the code.

For now, don't worry about elements of the YMODEM protocol that are not needed for this part of the project. For example, don't worry about timeouts, user cancellation of a session, and purging at the receiver, etc. We will deal with them in later parts of the project. The "kind medium" is kind but not error free. The kind medium complements certain bytes that pass from term2 to term1. The way the medium is written, the first byte of each block for the file we are transmitting should not get complemented. So this means that the SOH byte in our block headers will not be corrupted in this part of the project. The kind medium also introduces some errors when transferring data from terminal 1 to terminal 2: it changes every 9thth or so ACK from the YMODEM receiver into a NAK. The kind medium does not drop (i.e. discard) any characters. In any case we consider this medium to be kind and we shouldn't need to worry about timeouts in this part of the project.

In future parts of the project we will have an "evil medium", which may alter the SOH header byte and do other mean things and that will necessitate that further mechanisms be employed in the code to be able to correctly transfer files on most occasions.

When writing the receiver code, consider the StateChart that I will provide for the simplified receiver from this part of the project. There is a document describing StateCharts on Canvas. Before writing the sender code, draw a StateChart for our simplified sender for this part of the project. This term, mostly later in the term, we will be using StateChart software called SmartState. The state *ConditionalTransient* and its outgoing transitions implement in SmartState a "conditional pseudostate" as described on page 7 of the StateCharts document. The entry code 'POST("*",CONT)' in the *ConditionalTransient* state immediately posts a continue (*CONT*) event that immediately kicks the StateChart out of that state (because both transitions out of the state are triggered by that event). You will need to hand in a drawing of your simplified sender StateChart at the time of your Part2 submission. For Part 2 you do not need to use the SmartState software, though it is on Canvas in the Software folder in the Files area if you want to install it. You can draw on paper and scan your page in if you want, or use any drawing program that you may prefer.

eg: browser sockets to the server.

To communicate between the threads, we will use socketpairs as indicated in the provided code and in the following diagrams:

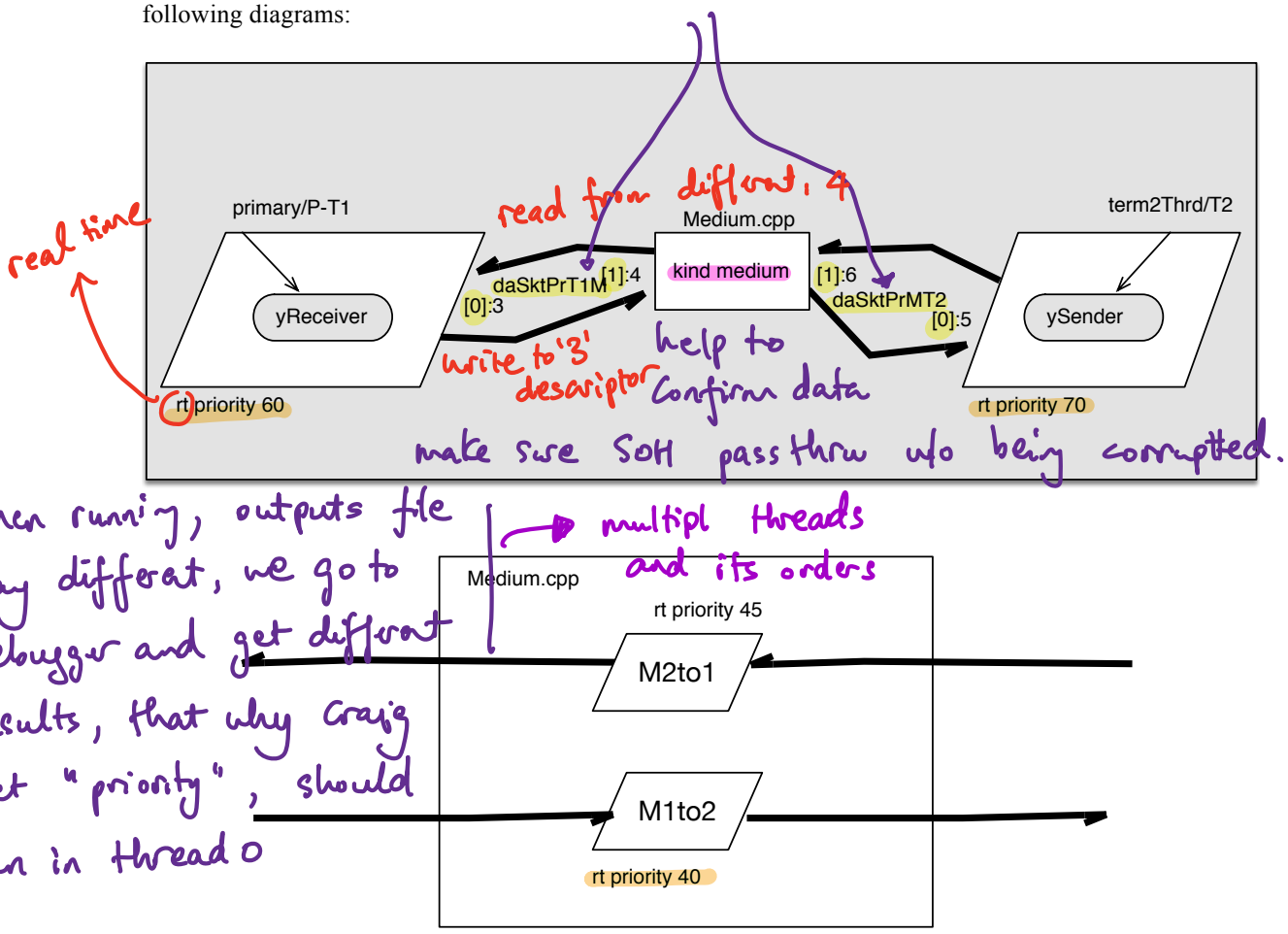


Figure 1: Collaboration graph showing asynchronous communication between threads provided by socketpairs

des [0] : input
des [1] : output
des [2] : error

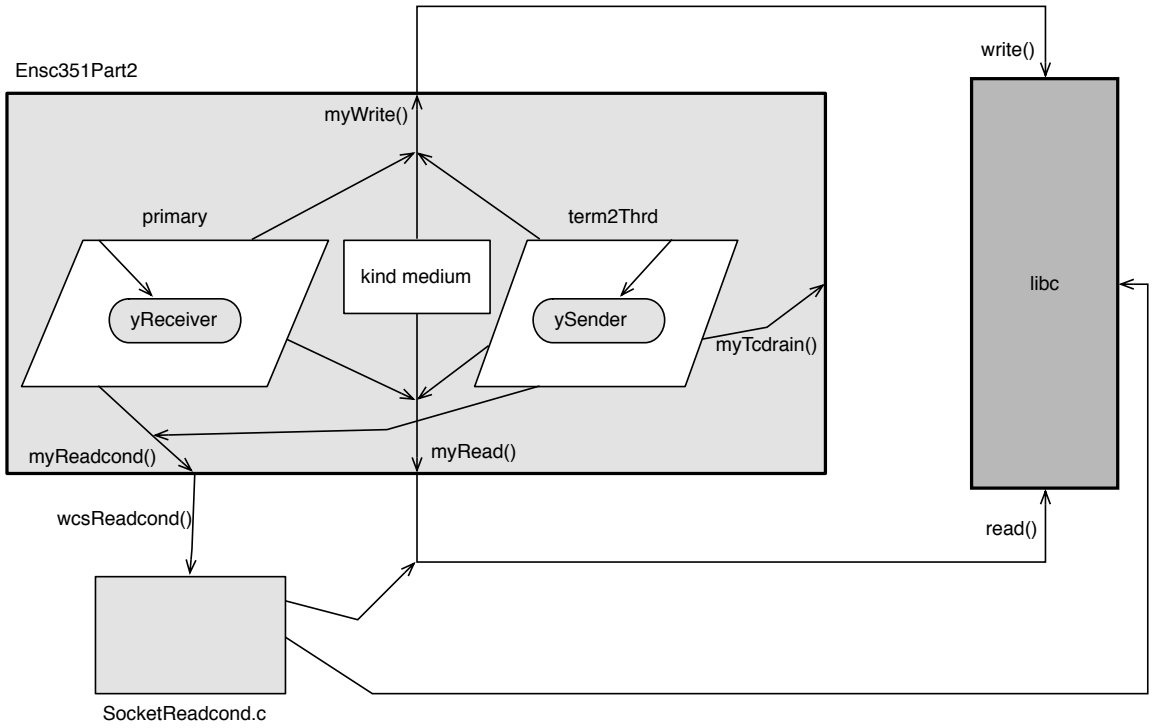


Figure 2: Collaboration Graph showing function calls for socketpair communication

Note that the template code writes to and reads from the socketpairs indirectly using functions *myWrite(), myRead()* and *myReadcond()* found in the file *myIO.cpp*. All calls to *myReadcond()*, which provides the functionality of the *readcond()* function ("READ from CONsole Device") available natively in the QNX Real-time Operating System, have the *time* and *timeout* arguments set to 0 for this part of the project. We will need and deal with timeouts later in the course. The supplied code will also call a function *myTcdrain(),* which will in Project Part 3 for socketpairs simulate (and later for serial ports use) the *tcdrain()* function ("Terminal Control: DRAIN"). For this part of the project, myTcdrain() does nothing. myTcdrain() is used so that the YMODEM sender will, in future parts of the project, be able to wait for the medium to have received all the bytes of a block already written to the medium before sending the last byte of the block. That will allow the sender to dump any glitches received before sending that last byte of the block. More on that later.

The YMODEM receiver should create an output file for each file being transferred. The kind medium copies all data it sends in both directions to a special output file ("ymodemData.dat").

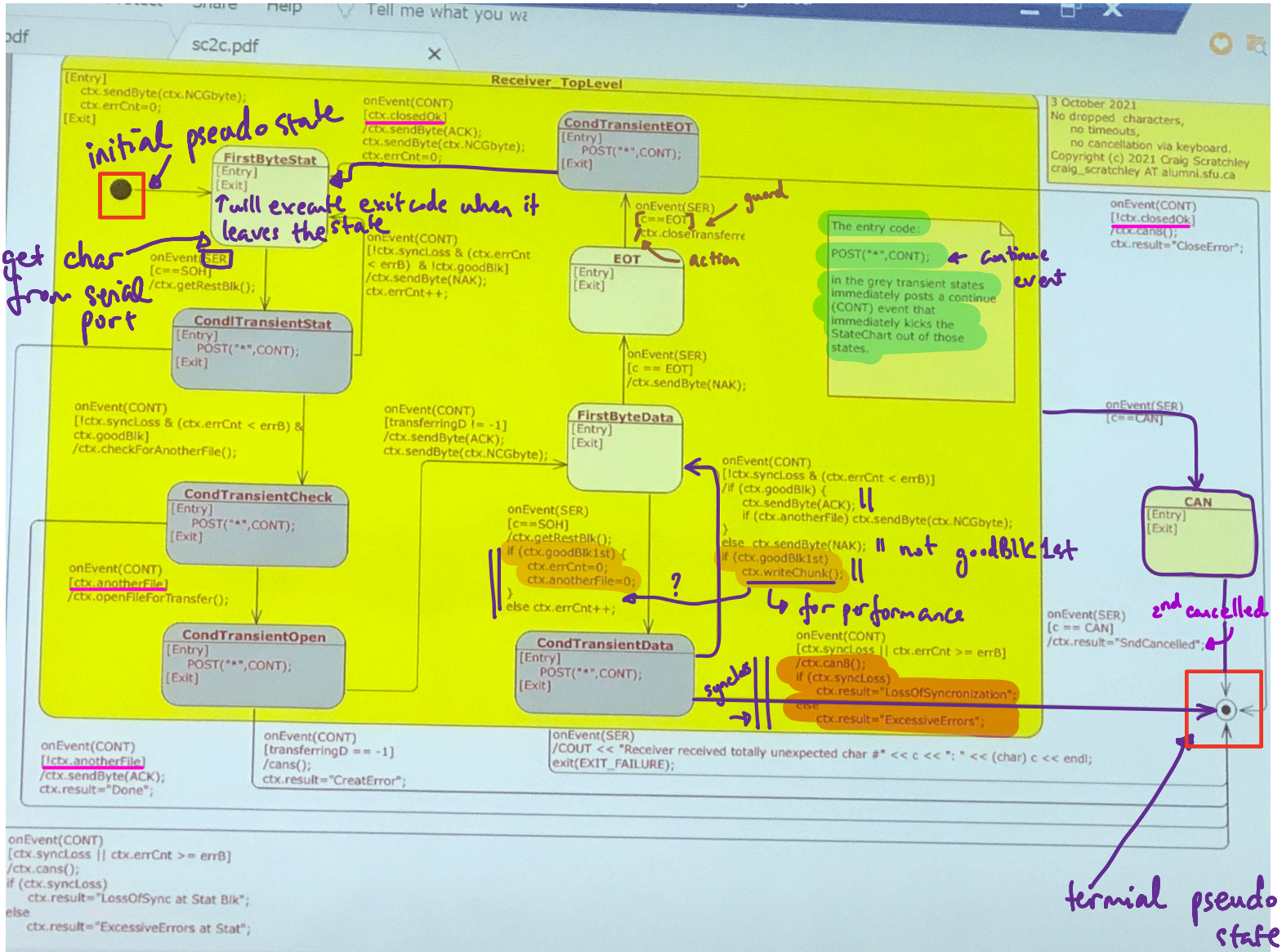
You will have two weeks to work on this part of the multipart project. The exact time will be announced shortly on CourSys (courses.cs.sfu.ca).

```
Ensc351Part2.cpp:
1) //should have term1Thrd??? where is primary thread?
    termFunc(Term1); //Receiver
    2 threads. medium thread is receiver

SenderY.cpp:
1) why we need to seperate most block, last byte?
    - last byte is to dump glitches
    - during file trannsfer, there will be some noise and we need to resend the data.
        Those noise are trash data that were sent incorrectly. We want to call the dumpGlitch()
        to clean those trash data before sending the last byte
2) What do we need to modify? how does this func affect to other funcs?
    void SenderY::sendBlkPrepNext()
    {
        // **** this function will need to be modified ****
        blkNum ++; // 1st block about to be sent or previous block ACK'd
        uint8_t lastByte = sendMostBlk(blkBuf);
        genBlk(blkBuf); // prepare next block
        sendLastByte(lastByte);
    }
    sendBlkPrepNext will need when hook up to kind Medium

ReceiverY.cpp:
1) explain while syntax?
    while (
            sendByte(ctx.NCGbyte),
            PE_NOT(myRead(mediumD, rcvBlk, 1), 1),// Should be SOH
            ctx.getRestBlk(), // get block 0 with fileName and filesize
            rcvBlk[DATA_POS])
    {}
    read comma operator
```

dead_lock: all threads try to read, but no data to read.

similar to finite state machine.

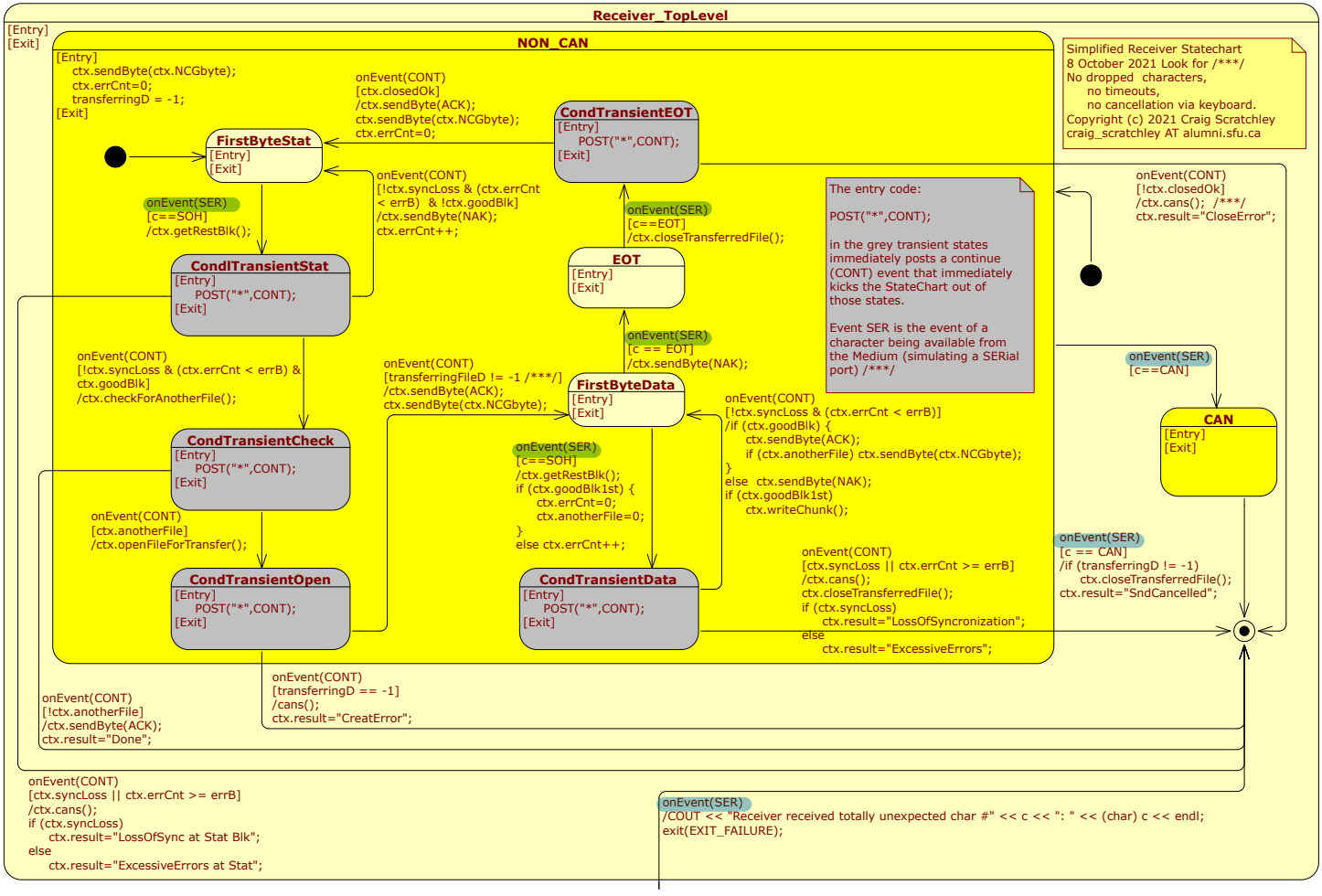
statechart

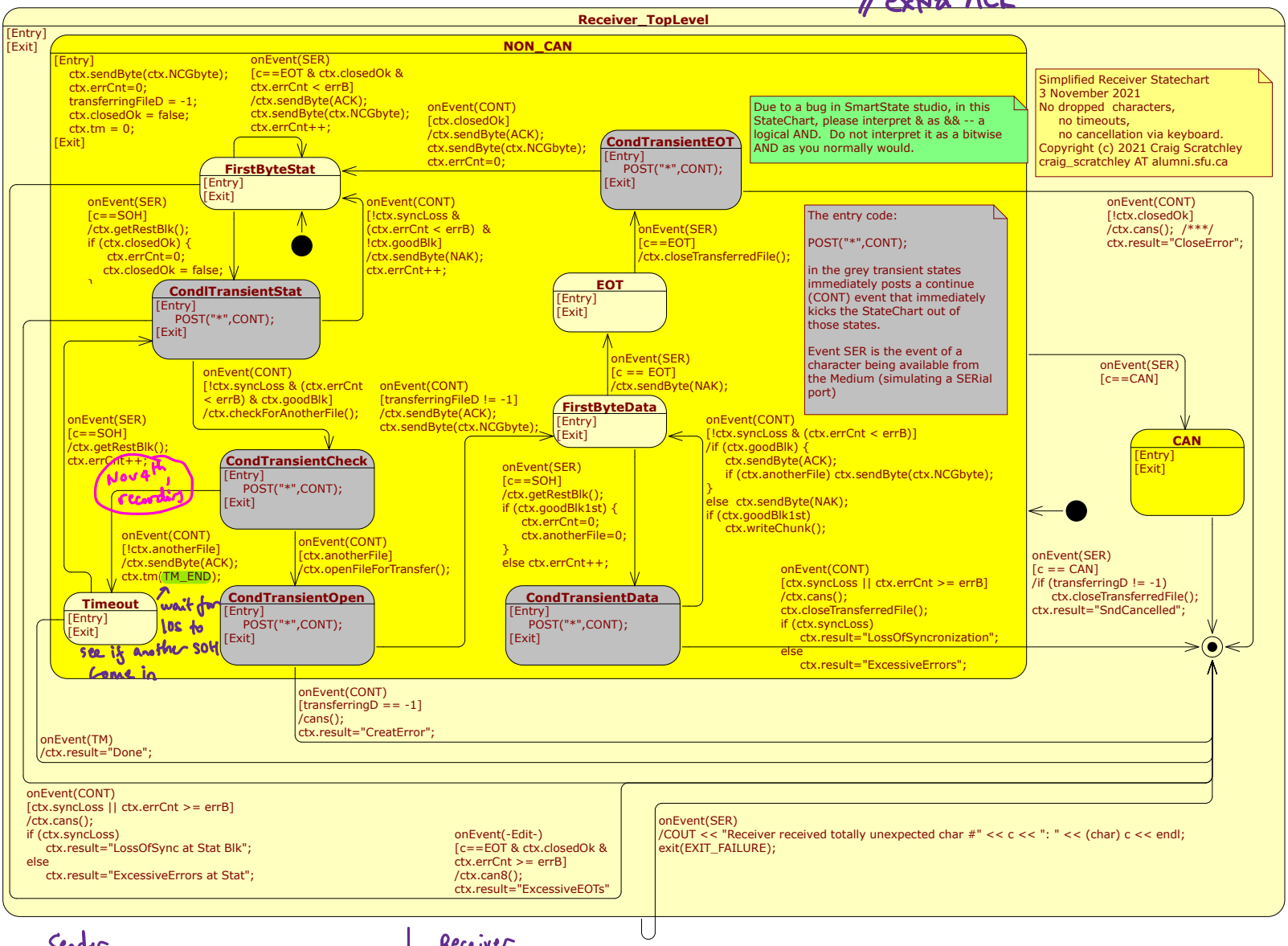


Convert statechart to C++ code:

Smart state:

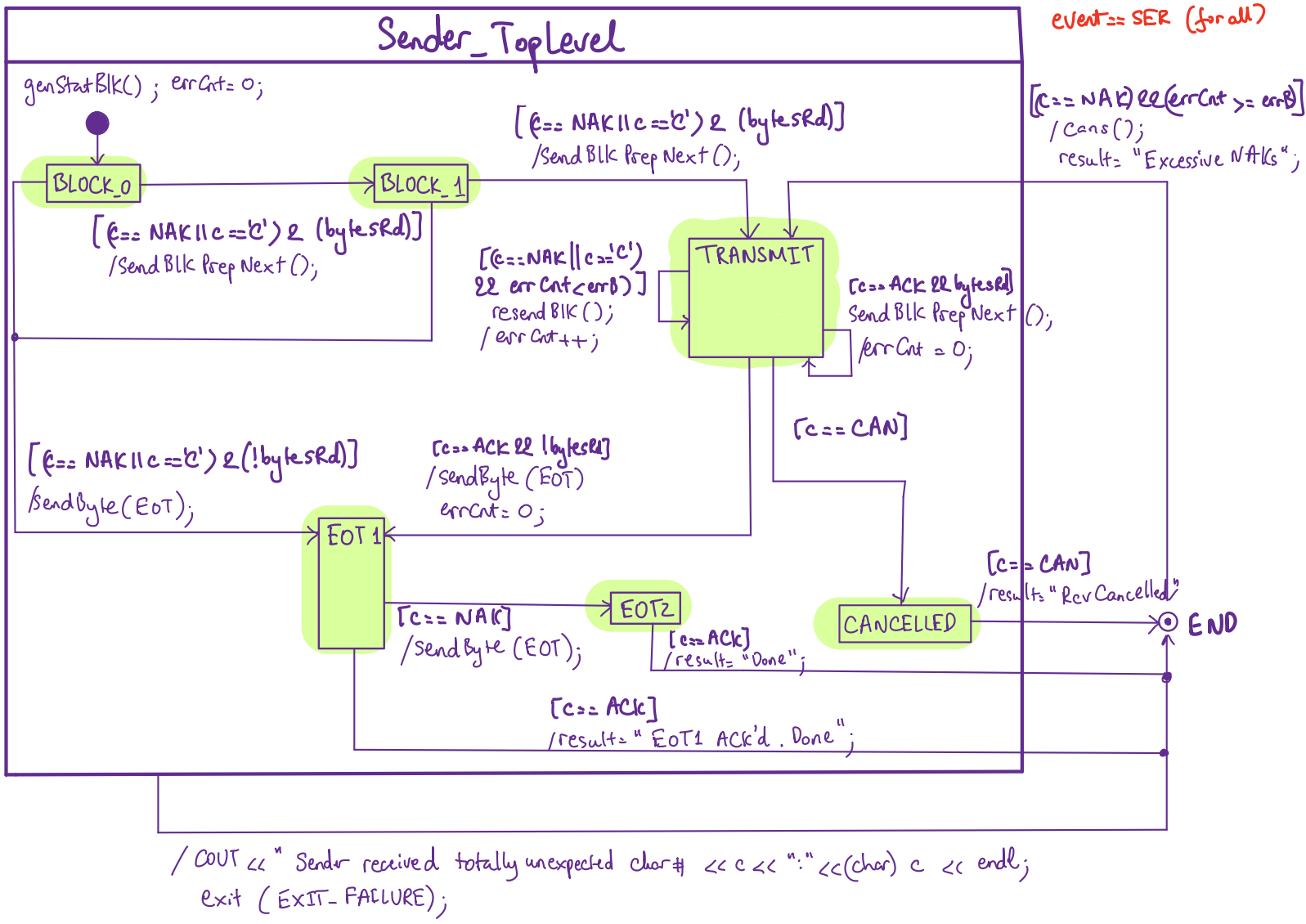
CodeGen → generate C++ file


Receiver_TopLevel
[Entry]
[Exit]
NON_CAN
[Entry]
ctx.sendByte(ctx.NCGbyte);
ctx.errCnt=0;
transferringD = -1;
[Exit]
FirstByteStat
[Entry]
[Exit]
onEvent(CONT)
[ctx.closedOk]
/ctx.sendByte(ACK);
ctx.sendByte(ctx.NCGbyte);
ctx.errCnt=0;
CondTransientEOT
[Entry]
POST("*",CONT);
[Exit]
onEvent(CONT)
[!ctx.syncLoss & (ctx.errCnt
< errB) & !ctx.goodBlk]
/ctx.sendByte(NAK);
ctx.errCnt++;
onEvent(SER)
[c==SOH]
/ctx.getRestBlk();
CondITransientStat
[Entry]
POST("*",CONT);
[Exit]
onEvent(SER)
[c==EOT]
/ctx.closeTransferredFile();
EOT
[Entry]
[Exit]
The entry code:
POST("*",CONT);
in the grey transient states
immediately posts a continue
(CONT) event that immediately
kicks the StateChart out of
those states.
Event SER is the event of a
character being available from
the Medium (simulating a SERial
port) /***/
onEvent(CONT)
[!ctx.closedOk]
/ctx.cans(); /***/
ctx.result="CloseError";
onEvent(CONT)
[!ctx.syncLoss & (ctx.errCnt < errB) &
ctx.goodBlk]
/ctx.checkForAnotherFile();
onEvent(CONT)
[transferringFileD != -1 /***/]
/ctx.sendByte(ACK);
ctx.sendByte(ctx.NCGbyte);
FirstByteData
[Entry]
[Exit]
onEvent(SER)
[c == EOT]
/ctx.sendByte(NAK);
onEvent(SER)
[c==CAN]
CAN
[Entry]
[Exit]
CondTransientCheck
[Entry]
POST("*",CONT);
[Exit]
onEvent(SER)
[c==SOH]
/ctx.getRestBlk();
if (ctx.goodBlk1st) {
ctx.errCnt=0;
ctx.anotherFile=0;
}
else ctx.errCnt++;
onEvent(CONT)
[!ctx.syncLoss & (ctx.errCnt < errB)]
/if (ctx.goodBlk) {
ctx.sendByte(ACK);
if (ctx.anotherFile) ctx.sendByte(ctx.NCGbyte);
}
else ctx.sendByte(NAK);
if (ctx.goodBlk1st)
ctx.writeChunk();
onEvent(CONT)
[ctx.anotherFile]
/ctx.openFileForTransfer();
CondTransientOpen
[Entry]
POST("*",CONT);
[Exit]
CondTransientData
[Entry]
POST("*",CONT);
[Exit]
onEvent(CONT)
[ctx.syncLoss || ctx.errCnt >= errB]
/ctx.cans();
ctx.closeTransferredFile();
if (ctx.syncLoss)
ctx.result="LossOfSyncronization";
else
ctx.result="ExcessiveErrors";
onEvent(SER)
[c == CAN]
/if (transferringD != -1)
ctx.closeTransferredFile();
ctx.result="SndCancelled";
onEvent(CONT)
[transferringD == -1]
/cans();
ctx.result="CreatError";
onEvent(CONT)
[!ctx.anotherFile]
/ctx.sendByte(ACK);
ctx.result="Done";
onEvent(CONT)
[ctx.syncLoss || ctx.errCnt >= errB]
/ctx.cans();
if (ctx.syncLoss)
ctx.result="LossOfSync at Stat Blk";
else
ctx.result="ExcessiveErrors at Stat";
onEvent(SER)
/COUT << "Receiver received totally unexpected char #" << c << ": " << (char) c << endl;
exit(EXIT_FAILURE);
Simplified Receiver Statechart
8 October 2021 Look for /***/
No dropped characters,
no timeouts,
no cancellation via keyboard.
Copyright (c) 2021 Craig Scratchley
craig_scratchley AT alumni.sfu.ca



Sender	Receiver
- empty stat Block (EOT)	- ACK - timeout

- you asked for 133 bytes, but medium drops some byte during transmission (says 128 bytes) ⇒ dead lock ⇒ timeout is used


Sender_TopLevel
event == SER (for all)
genStatBlk() ; errCnt = 0;
BLOCK_0
BLOCK_1
[(c == NAK || c == 'C') && (bytesRd)]
/Send Blk Prep Next();
[(c == NAK) && (errCnt >= errB)]
/ Cans();
result = "Excessive NAKs";
[(c == NAK || c == 'C') && (bytesRd)]
/Send Blk Prep Next();
TRANSMIT
[(c==NAK || c=='C') && errCnt<errB)]
resend Blk();
/ errCnt++;
[c == ACK && bytesRd]
Send Blk Prep Next();
/errCnt = 0;
[c == CAN]
[(c == NAK || c == 'C') && (!bytesRd)]
/SendByte(EOT);
[c == ACK && !bytesRd]
/sendByte(EOT)
errCnt = 0;
EOT 1
[c == NAK]
/SendByte(EOT);
EOT2
[c == ACK]
/result = "Done";
CANCELLED
[c == CAN]
/result = "RcvCancelled"
END
[c == ACK]
/result = "EOT1 ACK'd. Done";
/ COUT << " Sender received totally unexpected char# << c << ":" <<(char) c << endl;
exit (EXIT_FAILURE);

main.cpp

```
#include "posixThread.hpp"

using namespace std;
using namespace pthreadSupport;

int Ensc351Part2(); // should we include a header file?

int main()
{
   // Pre-allocate some memory for the process.
   // Seems to make priorities work better, at least
   // when using gdb.
   // For some programs, the amount of memory allocated
   // here should perhaps be greater.
   void* ptr = malloc(5000);
   free(ptr);

   try {      // ... realtime policy.
      // program should be starting off at realtime priority 98 or 99
      if (getSchedPrio() < 98)
         std::cout << "**** If you are debugging, debugger is not running at a high priority. ****\n"
<<
                  " **** This could cause problems with debugging.  Consider debugging\n" <<
                  " **** with the proper debug launch configuration ****" << endl;

      setSchedPrio(60); // drop priority down somewhat.  FIFO?
      return Ensc351Part2();
   }
   catch (system_error& error) {
      cout << "Error: " << error.code() << " - " << error.what() << '\n';
      cout << "Have you launched process with a realtime scheduling policy?" << endl;
      return error.code().value();
   }
   catch (...) { throw; }
}
```

ENSC Part 2.cpp

```
//
//============================================================================
//
//% Student Name 1: student1
//% Student 1 #: 123456781
//% Student 1 userid (email): stu1 (stu1@sfu.ca)
//
//% Student Name 2: student2
//% Student 2 #: 123456782
//% Student 2 userid (email): stu2 (stu2@sfu.ca)
//
//% Below, edit to list any people who helped you with the code in this file,
//%      or put 'None' if nobody helped (the two of) you.
//
// Helpers: _everybody helped us/me with the assignment (list names or put 'None')__
//
// Also, list any resources beyond the course textbooks and the course pages on Piazza
// that you used in making your submission.
//
// Resources:  ___________
//
//%% Instructions:
//% * Put your name(s), student number(s), userid(s) in the above section.
//% * Also enter the above information in other files to submit.
//% * Edit the "Helpers" line and, if necessary, the "Resources" line.
//% * Your group name should be "P2_<userid1>_<userid2>" (eg. P2_stu1_stu2)
//% * Form groups as described at:  https://courses.cs.sfu.ca/docs/students
//% * Submit files to courses.cs.sfu.ca
//
// Version     : September, 2021
// Copyright   : Copyright 2021, Craig Scratchley
// Description : Starting point for ENSC 351 Project Part 2
//
//============================================================================

#include <stdlib.h> // EXIT_SUCCESS
#include <sys/socket.h>
#include <pthread.h>
#include <thread>

#include "myIO.h"
#include "Medium.h"

#include "VNPE.h"
#include "AtomicCOUT.h"
#include "posixThread.hpp"

#include "ReceiverY.h"
```

```
#include "SenderY.h"

using namespace std;
using namespace pthreadSupport;

enum  {Term1, Term2};
enum  {TermSkt, MediumSkt};

static int daSktPr[2];   //Socket Pair between term1 and term2
//static int daSktPrT1M[2];          //Socket Pair between term1 and medium
//static int daSktPrMT2[2];          //Socket Pair between medium and term2

void testReceiverY(int mediumD)
{
   COUT << "Will try to receive file(s) with CRC" << endl;
   ReceiverY yReceiverCRC(mediumD);
   yReceiverCRC.receiveFiles();
   COUT << "yReceiver result was: " << yReceiverCRC.result << endl  << endl;
}

void testSenderY(vector<const char*> iFileNames, int mediumD)
{
   SenderY ySender(iFileNames, mediumD);
   COUT << "test sending" << endl;
   ySender.sendFiles();
   COUT << "Sender finished with result: " << ySender.result << endl << endl;
}

void termFunc(int termNum)
{
       // ***** modify this function to communicate with the "Kind Medium" *****

       if (termNum == Term1) {
//     testReceiverY(daSktPr[Term1]);       // file does not exist
       testReceiverY(daSktPr[Term1]);       // empty file and normal file.
       }
       else { // Term2
             PE_0(pthread_setname_np(pthread_self(), "T2")); // give the thread (terminal 2) a name
          // PE_0(pthread_setname_np("T2")); // Mac OS X

          vector<const char*> iFileNamesA = {"/doesNotExist.txt"};
          vector<const char*> iFileNamesB = {"/home/osboxes/.sudo_as_admin_successful", "/
home/osboxes/hs_err_pid11431.log"};

//     testSenderY(iFileNamesA, daSktPr[Term2]);   // file does not exist
       testSenderY(iFileNamesB, daSktPr[Term2]);   // empty file and normal file.
       }
       pthreadSupport::setSchedPrio(20); // drop priority down somewhat.  FIFO?
       PE(myClose(daSktPr[termNum]));
}
```

```
int Ensc351Part2()
{
	// ***** Modify this function to create the "Kind Medium" threads and communicate with it *****

	PE_0(pthread_setname_np(pthread_self(), "P-T1")); // give the Primary thread (Terminal 1) a name
   // PE_0(pthread_setname_np("P-T1")); // Mac OS X

	// ***** switch from having one socketpair for direct connection to having two socketpairs
	//                  for connection through medium threads *****
	PE(mySocketpair(AF_LOCAL, SOCK_STREAM, 0, daSktPr));
	//daSktPr[Term1] =  PE(/*myO*/open("/dev/ser2", O_RDWR));
  posixThread term2Thrd(SCHED_FIFO, 70, termFunc, Term2);

   // ***** create thread with SCHED_FIFO priority 40 for medium *****
   //    have the thread run the function found in Medium.cpp:
   //       void mediumFunc(int T1d, int T2d, const char *fname)
   //       where T1d is the descriptor for the socket to Term1
   //       and T2d is the descriptor for the socket to Term2
   //       and fname is the name of the binary medium "log" file
   //       ("ymodemData.dat").
   //     Make sure that thread is created at SCHED_FIFO priority 40

	termFunc(Term1);
   term2Thrd.join();
   // ***** join with thread for medium *****

	return EXIT_SUCCESS;
}
```

// Craig's gift

check if error occurs

term2 is parameter of termFunc

to get the same order of the output. It helps students compare .dat file

// thread2 exists (looks for Deby (e-side))

```
//                                                                    SenderY.cpp
//============================================================================
//
//% Student Name 1: student1
//% Student 1 #: 123456781
//% Student 1 userid (email): stu1 (stu1@sfu.ca)
//
//% Student Name 2: student2
//% Student 2 #: 123456782
//% Student 2 userid (email): stu2 (stu2@sfu.ca)
//
//% Below, edit to list any people who helped you with the code in this file,
//%      or put 'None' if nobody helped (the two of) you.
//
// Helpers: _everybody helped us/me with the assignment (list names or put 'None')__
//
// Also, list any resources beyond the course textbooks and the course pages on Piazza
// that you used in making your submission.
//
// Resources:  ___________
//
//%% Instructions:
//% * Put your name(s), student number(s), userid(s) in the above section.
//% * Also enter the above information in other files to submit.
//% * Edit the "Helpers" line and, if necessary, the "Resources" line.
//% * Your group name should be "P2_<userid1>_<userid2>" (eg. P2_stu1_stu2)
//% * Form groups as described at:  https://courses.cs.sfu.ca/docs/students
//% * Submit files to courses.cs.sfu.ca
//
// File Name   : SenderY.cpp
// Version     : September 23rd, 2021
// Description : Starting point for ENSC 351 Project Part 2
// Original portions Copyright (c) 2021 Craig Scratchley  (wcs AT sfu DOT ca)
//
//============================================================================

#include "SenderY.h"

#include <iostream>
#include <experimental/filesystem> // for C++14
#include <filesystem>
#include <stdio.h> // for snprintf()
#include <stdint.h> // for uint8_t
#include <string.h> // for memset(), and memcpy() or strncpy()
#include <errno.h>
#include <fcntl.h>     // for O_RDWR
#include <sys/stat.h>
```

```
#include "myIO.h"
#include "VNPE.h"

using namespace std;
using namespace std::filesystem; // C++17
//using namespace experimental::filesystem; // C++14

SenderY::
SenderY(vector<const char*> iFileNames, int d)
:PeerY(d),
 bytesRd(-1),
 fileNames(iFileNames),
// fileNameIndex(0),
 blkNum(0)
{
}

//------------------------------------------------------------------------

// get rid of any characters that may have arrived from the medium.
void SenderY::dumpGlitches()
{
        const int dumpBufSz = 20;
        char buf[dumpBufSz];
        int bytesRead;
        while (dumpBufSz == (bytesRead = PE(myReadcond(mediumD, buf, dumpBufSz, 0, 0, 0))));
}

// Send the block, less the block's last byte, to the receiver.
// Returns the block's last byte.

uint8_t SenderY::sendMostBlk(blkT blkBuf)
//uint8_t SenderY::sendMostBlk(uint8_t blkBuf[BLK_SZ_CRC])
{
        const int mostBlockSize = (BLK_SZ_CRC) - 1;
        PE_NOT(myWrite(mediumD, blkBuf, mostBlockSize), mostBlockSize);
        return *(blkBuf + mostBlockSize);
}

// Send the last byte of a block to the receiver
void
SenderY::
sendLastByte(uint8_t lastByte)
{
        PE(myTcdrain(mediumD)); // wait for previous part of block to be completely drained from
the descriptor
        dumpGlitches();                 // dump any received glitches

        PE_NOT(myWrite(mediumD, &lastByte, sizeof(lastByte)), sizeof(lastByte));
}
```

waits for medium descriptor to be drained

→ terminal control; wait until all output written to the object referred to by descriptor has been terminated

```
/* generate a block (numbered 0) with filename and filesize */
void SenderY::genStatBlk(blkT blkBuf, const char* fileName)
//void SenderY::genStatBlk(uint8_t blkBuf[BLK_SZ_CRC], const char* fileName)
{
   blkBuf[SOH_OH] = 0;
   blkBuf[SOH_OH + 1] = ~0;
   int index = DATA_POS;
   if (*fileName) { // (0 != strcmp("", fileName)) { // (strlen(fileName) > 0)
      const auto myBasename = path( fileName ).filename().string();
      auto c_basename = myBasename.c_str();
      int fileNameLengthPlus1 = strlen(c_basename) + 1;
      // check for fileNameLengthPlus1 greater than 127.
      if (fileNameLengthPlus1 + 1 > CHUNK_SZ) { // need at least one decimal digit to store
st.st_size below
         cout /* cerr */ << "Ran out of space in file info block!  Need block with 1024 bytes of
data." << endl;
         exit(-1);
      }
      // On Linux: The maximum length for a file name is 255 bytes. The maximum combined
length of both the file name and path name is 4096 bytes.
      memcpy(&blkBuf[index], c_basename, fileNameLengthPlus1);
      //strncpy(&blkBuf[index], c_basename, 12X);
      index += fileNameLengthPlus1;
      struct stat st;
      PE(stat(fileName, &st));
      int spaceAvailable = CHUNK_SZ + DATA_POS - index;
      int spaceNeeded = snprintf((char*)&blkBuf[index], spaceAvailable, "%ld", st.st_size); //
check the value of CHUNK_SZ + DATA_POS - index
      if (spaceNeeded > spaceAvailable) {
         cout /* cerr */ << "Ran out of space in file info block!  Need block with 1024 bytes of
data." << endl;
         exit(-1);
      }
      index += spaceNeeded + 1;

      blkNum = 0 - 1; // initialize blkNum for the data blocks to come.
   }
   uint8_t padSize = CHUNK_SZ + DATA_POS - index;
   memset(blkBuf+index, 0, padSize);

   // check here if index is greater than 128 or so.
   blkBuf[0] = SOH; // can be pre-initialized for efficiency if no 1K blocks allowed

   /* calculate and add CRC in network byte order */
   crc16ns((uint16_t*)&blkBuf[PAST_CHUNK], &blkBuf[DATA_POS]);
}

/* tries to generate a block.  Updates the
variable bytesRd with the number of bytes that were read
```

```
from the input file in order to create the block. Sets
bytesRd to 0 and does not actually generate a block if the end
of the input file had been reached when the previously generated block
was prepared or if the input file is empty (i.e. has 0 length).
*/
//void SenderY::genBlk(blkT blkBuf)
void SenderY::genBlk(uint8_t blkBuf[BLK_SZ_CRC])
{
        //read data and store it directly at the data portion of the buffer
        bytesRd = PE(myRead(transferringFileD, &blkBuf[DATA_POS], CHUNK_SZ ));
        if (bytesRd>0) {
                blkBuf[0] = SOH; // can be pre-initialized for efficiency
                //block number and its complement
                uint8_t nextBlkNum = blkNum + 1;
                blkBuf[SOH_OH] = nextBlkNum;
                blkBuf[SOH_OH + 1] = ~nextBlkNum;

                        //pad ctrl-z for the last block
                        uint8_t padSize = CHUNK_SZ - bytesRd;
                        memset(blkBuf+DATA_POS+bytesRd, CTRL_Z, padSize);

                /* calculate and add CRC in network byte order */
                crc16ns((uint16_t*)&blkBuf[PAST_CHUNK], &blkBuf[DATA_POS]);
        }
}

//Send CAN_LEN copies of CAN characters in a row to the YMODEM receiver, to inform it of
//      the cancelling of a file transfer
void SenderY::cans()
{
        // No need to space in time CAN chars for Part 2.
        // This function will be more complicated in later parts.
    char buffer[CAN_LEN];
    memset( buffer, CAN, CAN_LEN);
    PE_NOT(myWrite(mediumD, buffer, CAN_LEN), CAN_LEN);
}

/* While sending the now current block for the first time, prepare the next block if possible.
*/
void SenderY::sendBlkPrepNext()
{
    // **** this function will need to be modified ****
    blkNum ++; // 1st block about to be sent or previous block ACK'd
    uint8_t lastByte = sendMostBlk(blkBuf);
    genBlk(blkBuf); // prepare next block
    sendLastByte(lastByte);
}

// Resends the block that had been sent previously to the xmodem receiver.
void SenderY::resendBlk()
```

```
{
        // resend the block including the checksum or crc16
        //  ***** You will have to write this simple function *****
}

int
SenderY::
openFileToTransfer(const char* fileName)
{
    transferringFileD = myOpen(fileName, O_RDONLY);
    return transferringFileD;
}

int
SenderY::
closeTransferredFile()
{
    return PE(myClose(transferringFileD));
}

void SenderY::sendFiles()
{
    char byteToReceive;
    SenderY& ctx = *this; // needed to work with SmartState-generated code

    // ***** modify the below code according to the protocol *****
    // below is just a starting point.  You can follow a
    //  different structure if you want.

    //for (auto fileName : fileNames) {
    for (unsigned fileNameIndex = 0; fileNameIndex < fileNames.size(); ++fileNameIndex) {
        const char* fileName = fileNames[fileNameIndex];

        ctx.openFileToTransfer(fileName);

        if(ctx.transferringFileD != -1) {
            ctx.genStatBlk(blkBuf, fileName); // prepare 0eth block
        }
        PE_NOT(myRead(mediumD, &byteToReceive, 1), 1); // assuming get a 'C'
//      ctx.Crcflg = true;

        if(ctx.transferringFileD == -1) {
            ctx.cans();
            ctx.result = "OpenError"; // include errno and so forth in here.
        }
        else {
            ctx.sendBlkPrepNext();

            PE_NOT(myRead(mediumD, &byteToReceive, 1), 1); // assuming get an ACK
            PE_NOT(myRead(mediumD, &byteToReceive, 1), 1); // assuming get a 'C'
```

```
        while (ctx.bytesRd) {
            ctx.sendBlkPrepNext();
            // assuming on next line we get an ACK
            PE_NOT(myRead(mediumD, &byteToReceive, 1), 1);
        }
        ctx.sendByte(EOT); // send the first EOT
        PE_NOT(myRead(mediumD, &byteToReceive, 1), 1); // assuming get a NAK
        ctx.sendByte(EOT); // send the second EOT
        PE_NOT(myRead(mediumD, &byteToReceive, 1), 1); // assuming get an ACK

        ctx.closeTransferredFile();
      }
    }
    // indicate end of the batch.
    ctx.genStatBlk(blkBuf, ""); // prepare 0eth block
    PE_NOT(myRead(mediumD, &byteToReceive, 1), 1); // assuming get a 'C'
    ctx.sendLastByte(ctx.sendMostBlk(blkBuf));
    PE_NOT(myRead(mediumD, &byteToReceive, 1), 1); // assuming get an ACK

    ctx.result = "Done";
}
```

ReceiverY.cpp

```
//
//============================================================================
//
//% Student Name 1: student1
//% Student 1 #: 123456781
//% Student 1 userid (email): stu1 (stu1@sfu.ca)
//
//% Student Name 2: student2
//% Student 2 #: 123456782
//% Student 2 userid (email): stu2 (stu2@sfu.ca)
//
//% Below, edit to list any people who helped you with the code in this file,
//%      or put 'None' if nobody helped (the two of) you.
//
// Helpers: _everybody helped us/me with the assignment (list names or put 'None')__
//
// Also, list any resources beyond the course textbooks and the course pages on Piazza
// that you used in making your submission.
//
// Resources:  ___________
//
//%% Instructions:
//% * Put your name(s), student number(s), userid(s) in the above section.
//% * Also enter the above information in other files to submit.
//% * Edit the "Helpers" line and, if necessary, the "Resources" line.
//% * Your group name should be "P2_<userid1>_<userid2>" (eg. P2_stu1_stu2)
//% * Form groups as described at:  https://courses.cs.sfu.ca/docs/students
//% * Submit files to courses.cs.sfu.ca
//
// File Name   : ReceiverY.cpp
// Version     : September 24th, 2021
// Description : Starting point for ENSC 351 Project Part 2
// Original portions Copyright (c) 2021 Craig Scratchley  (wcs AT sfu DOT ca)
//
//============================================================================

#include "ReceiverY.h"

#include <string.h> // for memset()
#include <fcntl.h>
#include <stdint.h>
#include "myIO.h"
#include "VNPE.h"

//using namespace std;

ReceiverY::
ReceiverY(int d)
```

```
:PeerY(d),
NCGbyte('C'),
goodBlk(false),
goodBlk1st(false),
syncLoss(false), // transfer will end when syncLoss becomes true
numLastGoodBlk(0)
{
}

/* Only called after an SOH character has been received.
The function receives the remaining characters to form a complete
block.
The function will set or reset a Boolean variable,
goodBlk. This variable will be set (made true) only if the
calculated checksum or CRC agrees with the
one received and the received block number and received complement
are consistent with each other.
Boolean member variable syncLoss will only be set to
true when goodBlk is set to true AND there is a
fatal loss of syncronization as described in the XMODEM
specification.
The member variable goodBlk1st will be made true only if this is the first
time that the block was received in "good" condition. Otherwise
goodBlk1st will be made false.
*/
void ReceiverY::getRestBlk()
{
   // ********* this function must be improved ***********
   PE_NOT(myReadcond(mediumD, &rcvBlk[1], REST_BLK_SZ_CRC, REST_BLK_SZ_CRC, 0, 0),
REST_BLK_SZ_CRC);
   goodBlk1st = goodBlk = true;
}

//Write chunk (data) in a received block to disk
void ReceiverY::writeChunk()
{
      PE_NOT(myWrite(transferringFileD, &rcvBlk[DATA_POS], CHUNK_SZ), CHUNK_SZ);
}

int
ReceiverY::
openFileForTransfer()
{
   mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
   transferringFileD = myCreat((const char *) &rcvBlk[DATA_POS], mode);
   return transferringFileD;
}

int
ReceiverY::
```

```
closeTransferredFile()
{
   return myClose(transferringFileD);
}

//Send CAN_LEN CAN characters in a row to the XMODEM sender, to inform it of
//      the cancelling of a file transfer
void ReceiverY::cans()
{
       // no need to space in time CAN chars coming from receiver
   char buffer[CAN_LEN];
   memset( buffer, CAN, CAN_LEN);
   PE_NOT(myWrite(mediumD, buffer, CAN_LEN), CAN_LEN);
}

void ReceiverY::receiveFiles()
{                    context                    help communicate w/ stateChart
  ReceiverY& ctx = *this; // needed to work with SmartState-generated code

   // ***** improve this member function *****

   // below is just an example template.  You can follow a
   //  different structure if you want.

   while (
         sendByte(ctx.NCGbyte),
         PE_NOT(myRead(mediumD, rcvBlk, 1), 1), // Should be SOH
         ctx.getRestBlk(), // get block 0 with fileName and filesize
         rcvBlk[DATA_POS]) {

      if(openFileForTransfer() == -1) {
         cans();
         result = "CreatError"; // include errno or so
         return;
      }
      else {
         sendByte(ACK); // acknowledge block 0 with fileName.

         // inform sender that the receiver is ready and that the
         //      sender can send the first block
         sendByte(ctx.NCGbyte);

         while(PE_NOT(myRead(mediumD, rcvBlk, 1), 1), (rcvBlk[0] == SOH))
         {
            ctx.getRestBlk();
            ctx.sendByte(ACK); // assume the expected block was received correctly.
            ctx.writeChunk();
         };
         // assume EOT was just read in the condition for the while loop
         ctx.sendByte(NAK); // NAK the first EOT
```

```
        PE_NOT(myRead(mediumD, rcvBlk, 1), 1); // presumably read in another EOT

        // Check if the file closed properly.  If not, result should be "CloseError".
        if (ctx.closeTransferredFile()) {
            ; // ***** fill this in *****
        }
        else {
            ctx.sendByte(ACK);  // ACK the second EOT
            ctx.result="Done";
        }
      }
    }
    sendByte(ACK); // acknowledge empty block 0.
}
```

```
/*
 * Medium.cpp
 *
 *      Author: Craig Scratchley
 *      Version: September 10, 2021
 *      Copyright(c) 2021 Craig Scratchley
 */

#include <fcntl.h>
#include <unistd.h> // for write()
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <pthread.h>
#include "Medium.h"
#include "myIO.h"
#include "VNPE.h"
#include "AtomicCOUT.h"
#include "posixThread.hpp"

#include "PeerY.h"

// Uncomment the line below to turn on debugging output from the medium
//#define REPORT_INFO

//#define SEND_EXTRA_ACKS

//This is the kind medium.

#define T2toT1_CORRUPT_BYTE        395

using namespace std;
using namespace pthreadSupport;

Medium::Medium(int d1, int d2, const char *fname)
:Term1D(d1), Term2D(d2), logFileName(fname)
{
   byteCount = 0;
       ACKforwarded = 0;
       ACKreceived = 0;
       sendExtraAck = false;
   crcMode = false;

       mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
       logFileD = PE2(myCreat(logFileName, mode), logFileName);
}

Medium::~Medium() {
}
```

```
// this function will return false when it detects that the Term2 (sender) socket has closed.
bool Medium::MsgFromTerm2()
{
	blkT bytesReceived; // ?
	int numOfBytesReceived;
	int byteToCorrupt;

	if (!(numOfBytesReceived = PE(myRead(Term2D, bytesReceived, 1)))) {
		COUT << "Medium thread: TERM2's socket closed, Medium terminating" << endl;
		return false;
	}
   byteCount += numOfBytesReceived;

	PE_NOT(myWrite(logFileD, bytesReceived, numOfBytesReceived), numOfBytesReceived);
	//Forward the bytes to Term1 (usually RECEIVER),
	PE_NOT(myWrite(Term1D,   bytesReceived, numOfBytesReceived), numOfBytesReceived);

	if(bytesReceived[0] == CAN) {
		numOfBytesReceived = PE_NOT(myRead(Term2D, bytesReceived, CAN_LEN - 1),
CAN_LEN - 1);
		// byteCount += numOfBytesReceived;
		PE_NOT(myWrite(logFileD, bytesReceived, numOfBytesReceived),
numOfBytesReceived);
		//Forward the bytes to Term1 (usually RECEIVER),
		PE_NOT(myWrite(Term1D,   bytesReceived, numOfBytesReceived),
numOfBytesReceived);
	}
   else if (bytesReceived[0] == SOH) {
	   if (sendExtraAck) {
   #ifdef REPORT_INFO
		COUT << "{" << "+A" << "}" << flush;
   #endif
		uint8_t buffer = ACK;
		PE_NOT(myWrite(logFileD, &buffer, 1), 1);
		//Write the byte to term2,
		PE_NOT(myWrite(Term2D, &buffer, 1), 1);

		sendExtraAck = false;
	   }

	numOfBytesReceived = PE(myRead(Term2D, bytesReceived, (crcMode ? BLK_SZ_CRC :
BLK_SZ_CS) - numOfBytesReceived));

	byteCount += numOfBytesReceived;
	if (byteCount >= T2toT1_CORRUPT_BYTE) {
	   byteCount = byteCount - T2toT1_CORRUPT_BYTE;  // how large could byteCount end
up? (CB - 1) + 133 - CB = 132
	   byteToCorrupt = numOfBytesReceived - byteCount; // how small could byteToCorrupt
be?
	   if (byteToCorrupt < numOfBytesReceived) {
```

```
            bytesReceived[byteToCorrupt] = (255 - bytesReceived[byteToCorrupt]);
        #ifdef REPORT_INFO
            COUT << "<" << byteToCorrupt << "x>" << flush;
        #endif
            }
        }

        PE_NOT(myWrite(logFileD, &bytesReceived, numOfBytesReceived), numOfBytesReceived);
        //Forward the bytes to Term1 (RECEIVER),
        PE_NOT(myWrite(Term1D, &bytesReceived, numOfBytesReceived), numOfBytesReceived);
    }
        return true;
}

bool Medium::MsgFromTerm1()
{
        uint8_t buffer[CAN_LEN];
        int numOfByte = PE(myRead(Term1D, buffer, CAN_LEN));
        if (numOfByte == 0) {
            COUT << "Medium thread: TERM1's socket closed, Medium terminating" << endl;
            return false;
        }

    /*note that we record the corrupted ACK in the log file so that we can for it*/
        switch(buffer[0]) {
        case 'C':
            crcMode = true;
            break;
        case CAN:
            crcMode = false;
            break;
        case EOT:
            crcMode = false;
            break;
        case ACK: {
            ACKreceived++;

            if((ACKreceived%10)==0)
            {
                ACKreceived = 0;
                buffer[0]=NAK;
    #ifdef REPORT_INFO
                COUT << "{" << "AxN" << "}" << flush;
    #endif
            }
    #ifdef SEND_EXTRA_ACKS
            else/*actually forwarded ACKs*/
            {
                ACKforwarded++;
```

```
            if((ACKforwarded%6)==0)/*Note that this extra ACK is not an ACK forwarded from
receiver to the sender, so we don't increment ACKforwarded*/
            {
               ACKforwarded = 0;
               sendExtraAck = true;
            }
         }
   #endif
      }
      }

      PE_NOT(write(logFileD, buffer, numOfByte), numOfByte);

      //Forward the buffer to term2,
      PE_NOT(myWrite(Term2D, buffer, numOfByte), numOfByte);
      return true;
}

void
Medium::
mediumFuncT1toT2()
{
   PE_0(pthread_setname_np(pthread_self(), "M1to2"));
   while (MsgFromTerm1());
}

void Medium::run()
{
//    posixThread mediumThrd1to2(SCHED_FIFO, 35, &Medium::mediumFuncT1toT2, this); //
gives error
   thread mediumThrd1to2(&Medium::mediumFuncT1toT2, this);
   pthreadSupport::setSchedPrio(45); // raise priority up slightly.  FIFO?

   //transfer data from Term2 (sender)
   while (MsgFromTerm2());

   mediumThrd1to2.join();
      PE(myClose(logFileD));
      PE(myClose(Term1D));
      PE(myClose(Term2D));
}


void mediumFunc(int T1d, int T2d, const char *fname)
{
   PE_0(pthread_setname_np(pthread_self(), "M2to1"));
   Medium medium(T1d, T2d, fname);
   medium.run();
}
```

```
//
=============================================================================
====
// File Name   : PeerX.cpp (interim version -- to be replaced with Part 1 submission and
eventually solution)
// Version     : September 24th, 2021
// Description : Starting point for ENSC 351 Project with Part 1 Solution
// Original portions Copyright (c) 2021 Craig Scratchley  (wcs AT sfu DOT ca)
//
=============================================================================
====

#include "PeerY.h"

#include <arpa/inet.h> // for htons() -- not available with MinGW

#include "VNPE.h"

#include "myIO.h"

uint16_t my_htons(uint16_t n)
{
   unsigned char *np = (unsigned char *)&n;

   return
      ((uint16_t)np[0] << 8) |
      (uint16_t)np[1];
}

/* update CRC */
/*
The following XMODEM crc routine is taken from "rbsb.c".  Please refer to
   the source code for these programs (contained in RZSZ.ZOO) for usage.
As found in Chapter 8 of the document "ymodem.txt".
   Original 1/13/85 by John Byrns
   */

/*
 * Programmers may incorporate any or all code into their programs,
 * giving proper credit within the source. Publication of the
 * source routines is permitted so long as proper credit is given
 * to Stephen Satchell, Satchell Evaluations and Chuck Forsberg,
 * Omen Technology.
 */

unsigned short
updcrc(register int c, register unsigned crc)
{
      register int count;
```

```
        for (count=8; --count>=0;) {
                if (crc & 0x8000) {
                        crc <<= 1;
                        crc += (((c<<=1) & 0400)  !=  0);
                        crc ^= 0x1021;
                }
                else {
                        crc <<= 1;
                        crc += (((c<<=1) & 0400)  !=  0);
                }
        }
        return crc;
}

// Should return via crc16nsP a crc16 in 'network byte order'.
// Derived from code in "rbsb.c" (see above).
// Line comments in function below show lines removed from original code.
void
crc16ns (uint16_t* crc16nsP, uint8_t* buf)
{
        register int wcj;
        register uint8_t *cp;
        unsigned oldcrc=0;
        for (wcj=CHUNK_SZ,cp=buf; --wcj>=0; ) {
                //sendline(*cp);

                /* note the octal number in the line below */
                oldcrc=updcrc((0377& *cp++), oldcrc);

                //checksum += *cp++;
        }
        //if (Crcflg) {
                oldcrc=updcrc(0,updcrc(0,oldcrc));
                /* at this point, the CRC16 is in oldcrc */

                /* This is where rbsb.c "wrote" the CRC16.  Note how the MSB
                 * is sent before the LSB
                 * sendline is a function to "send a byte over a telephone line"
                 */
                //sendline((int)oldcrc>>8);
                //sendline((int)oldcrc);

                /* in our case, we want the bytes to be in the memory pointed to by crc16nsP
                 * in the correct 'network byte order'
                 */
                *crc16nsP = my_htons((uint16_t) oldcrc);
                /* *crc16nsP = htons((uint16_t) oldcrc); */

        //}
        //else
```

```
            //sendline(checksum);
}

PeerY::
PeerY(int d)
:result("ResultNotSet"),
 errCnt(0),
 //Crcflg(useCrc),
 mediumD(d),
 transferringFileD(-1)  // will need to be updated
{
}

//Send a byte to the remote peer across the medium
void
PeerY::
sendByte(uint8_t byte)
{
        PE_NOT(myWrite(mediumD, &byte, sizeof(byte)), sizeof(byte));
}
```

```
/*
 * Medium.cpp
 *
 *      Author: Craig Scratchley
 *      Version: September 10, 2021
 *      Copyright(c) 2021 Craig Scratchley
 */

#include <fcntl.h>
#include <unistd.h> // for write()
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <pthread.h>
#include "Medium.h"
#include "myIO.h"
#include "VNPE.h"
#include "AtomicCOUT.h"
#include "posixThread.hpp"

#include "PeerY.h"

// Uncomment the line below to turn on debugging output from the medium
//#define REPORT_INFO

//#define SEND_EXTRA_ACKS

//This is the kind medium.

#define T2toT1_CORRUPT_BYTE        395

using namespace std;
using namespace pthreadSupport;

Medium::Medium(int d1, int d2, const char *fname)
:Term1D(d1), Term2D(d2), logFileName(fname)
{
   byteCount = 0;
      ACKforwarded = 0;
      ACKreceived = 0;
      sendExtraAck = false;
   crcMode = false;
```

```
    mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
    logFileD = PE2(myCreat(logFileName, mode), logFileName);
}

Medium::~Medium() {
}

// this function will return false when it detects that the Term2 (sender) socket has
closed.
bool Medium::MsgFromTerm2()
{
    blkT bytesReceived; // ?
    int numOfBytesReceived;
    int byteToCorrupt;

    if (!(numOfBytesReceived = PE(myRead(Term2D, bytesReceived, 1)))) {
        COUT << "Medium thread: TERM2's socket closed, Medium terminating" <<
endl;
        return false;
    }
  byteCount += numOfBytesReceived;

    PE_NOT(myWrite(logFileD, bytesReceived, numOfBytesReceived),
numOfBytesReceived);
    //Forward the bytes to Term1 (usually RECEIVER),
    PE_NOT(myWrite(Term1D,   bytesReceived, numOfBytesReceived),
numOfBytesReceived);

    if(bytesReceived[0] == CAN) {
        numOfBytesReceived = PE_NOT(myRead(Term2D, bytesReceived, CAN_LEN -
1), CAN_LEN - 1);
        // byteCount += numOfBytesReceived;
        PE_NOT(myWrite(logFileD, bytesReceived, numOfBytesReceived),
numOfBytesReceived);
        //Forward the bytes to Term1 (usually RECEIVER),
        PE_NOT(myWrite(Term1D,   bytesReceived, numOfBytesReceived),
numOfBytesReceived);
    }
  else if (bytesReceived[0] == SOH) {
     if (sendExtraAck) {
  #ifdef REPORT_INFO
         COUT << "{" << "+A" << "}" << flush;
  #endif
```

```
            uint8_t buffer = ACK;
            PE_NOT(myWrite(logFileD, &buffer, 1), 1);
            //Write the byte to term2,
            PE_NOT(myWrite(Term2D, &buffer, 1), 1);

            sendExtraAck = false;
        }

        numOfBytesReceived = PE(myRead(Term2D, bytesReceived, (crcMode ?
BLK_SZ_CRC : BLK_SZ_CS) - numOfBytesReceived));

        byteCount += numOfBytesReceived;
        if (byteCount >= T2toT1_CORRUPT_BYTE) {
            byteCount = byteCount - T2toT1_CORRUPT_BYTE;  // how large could
byteCount end up? (CB - 1) + 133 - CB = 132
            byteToCorrupt = numOfBytesReceived - byteCount; // how small could
byteToCorrupt be?
            if (byteToCorrupt < numOfBytesReceived) {
                bytesReceived[byteToCorrupt] = (255 - bytesReceived[byteToCorrupt]);
        #ifdef REPORT_INFO
                COUT << "<" << byteToCorrupt << "x>" << flush;
        #endif
            }
        }

        PE_NOT(myWrite(logFileD, &bytesReceived, numOfBytesReceived),
numOfBytesReceived);
        //Forward the bytes to Term1 (RECEIVER),
        PE_NOT(myWrite(Term1D, &bytesReceived, numOfBytesReceived),
numOfBytesReceived);
    }
    return true;
}

bool Medium::MsgFromTerm1()
{
    uint8_t buffer[CAN_LEN];
    int numOfByte = PE(myRead(Term1D, buffer, CAN_LEN));
    if (numOfByte == 0) {
        COUT << "Medium thread: TERM1's socket closed, Medium terminating" <<
endl;
        return false;
    }
```

```
   /*note that we record the corrupted ACK in the log file so that we can for it*/
      switch(buffer[0]) {
      case 'C':
         crcMode = true;
         break;
      case CAN:
         crcMode = false;
         break;
      case EOT:
         crcMode = false;
         break;
      case ACK: {
         ACKreceived++;

         if((ACKreceived%10)==0)
         {
            ACKreceived = 0;
            buffer[0]=NAK;
   #ifdef REPORT_INFO
            COUT << "{" << "AxN" << "}" << flush;
   #endif
         }
   #ifdef SEND_EXTRA_ACKS
         else/*actually forwarded ACKs*/
         {
            ACKforwarded++;

            if((ACKforwarded%6)==0)/*Note that this extra ACK is not an ACK forwarded
from receiver to the sender, so we don't increment ACKforwarded*/
            {
               ACKforwarded = 0;
               sendExtraAck = true;
            }
         }
   #endif
      }
      }

      PE_NOT(write(logFileD, buffer, numOfByte), numOfByte);

      //Forward the buffer to term2,
      PE_NOT(myWrite(Term2D, buffer, numOfByte), numOfByte);
```

```
      return true;
}

void
Medium::
mediumFuncT1toT2()
{
   PE_0(pthread_setname_np(pthread_self(), "M1to2"));
   while (MsgFromTerm1());
}

void Medium::run()
{
//    posixThread mediumThrd1to2(SCHED_FIFO, 35, &Medium::mediumFuncT1toT2,
this); // gives error
   thread mediumThrd1to2(&Medium::mediumFuncT1toT2, this);
   pthreadSupport::setSchedPrio(45); // raise priority up slightly.  FIFO?

   //transfer data from Term2 (sender)
   while (MsgFromTerm2());

   mediumThrd1to2.join();
      PE(myClose(logFileD));
      PE(myClose(Term1D));
      PE(myClose(Term2D));
}


void mediumFunc(int T1d, int T2d, const char *fname)
{
   PE_0(pthread_setname_np(pthread_self(), "M2to1"));
   Medium medium(T1d, T2d, fname);
   medium.run();
}
```

myIO.cpp

```
//
============================================================
==========
//
//% Student Name 1: student1
//% Student 1 #: 123456781
//% Student 1 userid (email): stu1 (stu1@sfu.ca)
//
//% Student Name 2: student2
//% Student 2 #: 123456782
//% Student 2 userid (email): stu2 (stu2@sfu.ca)
//
//% Below, edit to list any people who helped you with the code in this file,
//%      or put 'None' if nobody helped (the two of) you.
//
// Helpers: _everybody helped us/me with the assignment (list names or put 'None')__
//
// Also, list any resources beyond the course textbooks and the course pages on Piazza
// that you used in making your submission.
//
// Resources:  ___________
//
//%% Instructions:
//% * Put your name(s), student number(s), userid(s) in the above section.
//% * Also enter the above information in other files to submit.
//% * Edit the "Helpers" line and, if necessary, the "Resources" line.
//% * Your group name should be "P3_<userid1>_<userid2>" (eg. P3_stu1_stu2)
//% * Form groups as described at:  https://courses.cs.sfu.ca/docs/students
//% * Submit files to courses.cs.sfu.ca
//
// File Name   : myIO.cpp
// Version     : September 28, 2021
// Description : Wrapper I/O functions for ENSC-351
// Copyright (c) 2021 Craig Scratchley  (wcs AT sfu DOT ca)
//
============================================================
==========

#include <unistd.h>              // for read/write/close
#include <fcntl.h>               // for open/creat
#include <sys/socket.h>   // for socketpair
#include <stdarg.h>        // for va stuff
```

```
#include "SocketReadcond.h"

int myOpen(const char *pathname, int flags, ...) //, mode_t mode)
{
   mode_t mode = 0;
   // in theory we should check here whether mode is needed.
   va_list arg;
   va_start (arg, flags);
   mode = va_arg (arg, mode_t);
   va_end (arg);
      return open(pathname, flags, mode);
}

int myCreat(const char *pathname, mode_t mode)
{
      return creat(pathname, mode);
}

int mySocketpair( int domain, int type, int protocol, int des_array[2] )
{
   int returnVal = socketpair(domain, type, protocol, des_array);
   return returnVal;
}

ssize_t myRead( int des, void* buf, size_t nbyte )
{
      return read(des, buf, nbyte );
}

ssize_t myWrite( int des, const void* buf, size_t nbyte )
{
      return write(des, buf, nbyte );
}

int myClose( int des )
{
      return close(des);
}

int myTcdrain(int des)
{ //is also included for purposes of the course.
      return 0;
}
```

```
/* Arguments:
des
    The file descriptor associated with the terminal device that you want to read from.
buf
    A pointer to a buffer into which readcond() can put the data.
n
    The maximum number of bytes to read.
min, time, timeout
    When used in RAW mode, these arguments override the behavior of the MIN and
TIME members of the terminal's termios structure. For more information, see...
 *
 *  https://developer.blackberry.com/native/reference/core/com.qnx.doc.neutrino.lib_ref/
topic/r/readcond.html
 *
 *  */
int myReadcond(int des, void * buf, int n, int min, int time, int timeout)
{
      return wcsReadcond(des, buf, n, min, time, timeout );
}
```