ENSC 351: Real-time and Embedded Systems

Craig Scratchley, Fall 2021

Multipart Project Part 5

Please continue working with a partner.

Part 2 of the multipart project included programming some code for the YMODEM protocol that allowed an YMODEM file transfer to complete in some situations – it could handle, for example, some bytes of the data in a block being corrupted, but could not handle dropped characters or some other sorts and combinations of the transmission errors that can occur. In Part 5 we will expand on code derived from the Part 2, 3, and 4 solutions by handling the case when the medium becomes more evil, and will in either direction corrupt or drop bytes being transmitted and also inject glitch bytes onto the medium.

We have introduced a new thread for switching console input and aggregating console output. See the "kvm" thread in the diagram. It doesn't support a mouse, so maybe we should have just called it the "kv" thread. And it aggregates console input, so perhaps we really should have chosen a better name. Oh well, kvm is the name I've used. KVM commands are:

~1<return> -- write future keyboard data to terminal 1
~2<return> -- write future keyboard data to terminal 2
~q!<return> -- quit the entire program

Lines of input that do not start with one of these commands are forwarded to terminal 1 or 2 depending on which of these is currently selected to receive keyboard data. Terminal 2 is initially selected to receive keyboard data. The terminals, in turn, have their own commands:

&s [<filepath>]<return>
-- send file with specified path (or use the default filepath)

&r <return>
-- receive file

&c<return>
-- cancel transmission in progress

When received by a terminal, lines of input that do not start with one of these commands are transmitted to the other terminal through the medium. See the diagram "Simulator in Part 5".

StateCharts

Because of the expanded requirements in this part of the course project, including the ability described above to cancel a transmission in progress, you will need StateCharts that handle all features of the YMODEM protocol. Following the StateChart that you prepared for Part 4, I am providing you with the solution complete StateCharts. (As I believe I have mentioned, SmartState unfortunately does not always generate 100% correct code, particularly with regard to entry and exit code, and so I have provided you with StateCharts that I have tested as properly working around such problems).

Medium

For data going from TERM2 to TERM1, the Part 5 (evil) medium will corrupt (complement) the 264th byte and every 790 bytes thereafter. Also, it will drop (not forward) the 659th byte received from TERM2 and every 790 bytes thereafter. Every second time the medium corrupts a byte going in this direction, it will also inject 25 or so glitch bytes at the end of the bytes provided by the medium's read() function call.

For data going from TERM1 to TERM2, the medium will corrupt (complement) every 6th byte it gets from TERM1 and drop every 7th byte it gets from TERM1 (a byte like the 42nd is dropped and so does not need to be corrupted too). Also, before actually forwarding every 3rd byte (i.e. don't count dropped bytes), this medium will send to TERM2 a glitch byte. The first glitch byte sent will have the numeric value 0, and each subsequent glitch byte will increment the value by 1. Lastly, the medium writes to TERM2 an extra ACK after every 50th byte it gets from TERM1. The extra ACK is written not immediately, but rather is written to TERM2 after the next byte is written to TERM1.

I am providing the medium code to you, so you don't need to code this. Just before writing any characters to either terminal, the medium will write the characters to a special output file called /tmp/ymodemData.dat

Sample console input/output should look something like the following (complementing, dropping, and insertion of glitch bytes, etc. is just to give you the idea, this is not what you should actually get). The italic lines in *green* are typed into the Eclipse console via your keyboard. Comments are in **bold blue**.

```
KVM FUNCTION BEGINS (INPUT ROUTED TO terminal 2)
&s              or…    &s /etc/protocols (/etc/protocols is the default for fast simulations)
TERM 2: Will request sending of '/etc/protocols'
~1
KVM SWITCHING TO terminal 1
&r
TERM 1: Will request receiving (some debug info will probably be displayed after this)
TERM1: xReceiver result was: Done
TERM2: xSender result was: Done
~2
&s
TERM 2: Will request sending of '/etc/protocols'
~1
KVM SWITCHING TO terminal 1
&r
TERM 1: Will request receiving to '/tmp/t-file'
TERM1: xReceiver result was: Done
TERM2: xSender result was: Done
Hi, how are you?
Hi, hw arGe yCu?
~2
KVM SWITCHING TO terminal 1
I'm fine, thanks. Please send me a file again.
I'm fne, thanks. Plese seCd me a file again.
~q!
KVM TERMINATING
TERM 1: inD Closed
TERM 2: inD Closed
Medium thread: TERM1's socket closed, Medium terminating
```

Note that in the above sample, all debugging output has been turned off. Some debugging output can be turned on by "#define"ing REPORT_INFO in Medium.cpp, ReceiverY.cpp, and/or SenderY.cpp. Medium.cpp is in the Ensc351Part5 application project. ReceiverY.cpp and SenderY.cpp are in the

Ensc351ymodLib library project. Both projects are in a common .zip file that you can download from Canvas. I have enabled some debugging output in the code I am providing to you. If desired, StateChart debugging output can be turned on for the StateCharts by commenting out the setDebugLog(nullptr) call in the transferCommon() function in PeerY.cpp.

Because during file transfer the YMODEM sender and YMODEM receiver must now monitor for input from both the keyboard (via the KVM thread) and the medium, the select() function must be used somewhere in PeerY.cpp. Simple examples of the use of the select() function can be seen in a number of places in the code, for example in function *Terminal*() in the file terminal.cpp (in the library project). Note that in PeerY.cpp the select() call may need to timeout, so the last argument of the select() call there should not necessarily be NULL. Please see the documentation for the select() function such as in the man page for select().

Starting from the code I am providing to you, add to and change the code and debug your program to get it working with the evil Part 5 medium described above. The program as I have given it to you uses a kind medium similar to Part 2 and can successfully send a file, at least in the direction from TERM2 to TERM1. To switch the behaviour of the medium in each direction, see the preprocessor definitions near the top of the Medium.h file. In terms of specific changes, you will need to add/change lines of code in function transferCommon() in PeerY.cpp and function purge() in the ReceiverY class. The number of lines of code that you need to add or change is actually quite limited, but the work should make you think and hopefully you will find it interesting.

Note that in the file socketReadcond.c in the ensc351 library project, as was the case with Part2 of the multipart project, I am providing you with a function *wcsReadcond*() that is similar to the QNX *readcond*() function, and that works with socket(pair)s too. We are accessing wcsReadcond() via the myReadcond() function in myIO.cpp. Read the documentation for readcond() to learn more about this function. A simple example of the use of myReadcond() is shown in the current version of the getRestBlk() function in ReceiverY.cpp. myIO.cpp is in the Ensc351ymodLib library project.

Details on exactly which files to submit will be provided separately on CourSys.