ENSC 351: Real-time and Embedded Systems

Craig Scratchley, Fall 2021

Multipart Project Part #1

Details on how and when to submit this part of the project will be given sometime in the next week. You will have about two weeks to work on this part. I will probably give out the next part of the project before this part is due, so don't delay in getting started on this part.

Unless I have instructed you or agreed with you otherwise, please choose one partner to work with. Choose your partner carefully. Try to choose a partner that will work with you on the various parts of the multipart project with as much care as your own degree of care. For purposes like this course, pair programming can be very useful.

http://en.wikipedia.org/wiki/Pair_programming

This part of the project does not require understanding real-time or embedded systems. It is a standard programming assignment and allows you to practice programming using relatively simple C++.

I have listed a good C++ book in the course syllabus. The book for ENSC 251 might also be helpful.

The first part of the project focuses on parts of the YMODEM-batch file transfer protocol.

Modify and complete the designated function members in the SenderY class in SenderY.cpp and in the PeerY class in PeerY.cpp. For this first part of the project, we are just simulating the sending of a couple files, including "/home/osboxes/hs_err_pid11506.log", using 16-bit CRC to detect errors and having all blocks hold 128 bytes of data. The protocol is described in the "ymodem.txt" file at

http://pauillac.inria.fr/~doligez/zmodem/ymodem.txt

and that file will be our primary reference. I've noticed a few ways to improve that file and so may create my own edited version of it and post it on Canvas. Other reference documents may be provided as well.

For the first part of the project, instead of sending the blocks and any other characters over a serial port or similar, the blocks and other characters should simply be written to an output file, "ymodemSenderData.dat". This file should only contain bytes that would normally be sent from the sender to the receiver. We are just imagining that a receiver exists and are assuming that it will correctly start the sender, positively acknowledge every block that it receives, and properly handle the termination of each transfer after it has received all the blocks for each file in the batch. A YMODEM sender implementation is allowed to send blocks that can hold either 128 bytes or 1024 bytes of data. At least for this first part of the project, we will generate only blocks that can hold just 128 bytes of data.

I have written a rather complete template for you to modify. You will only need to modify the member functions genStatBlk, genBlk(), sendBlk(), cans(), sendFiles(), and crc16ns(), calling any additional functions that you might want. See the sendFile() member function as given to you for a simulation of the main part of the protocol in operation and for the use of the genBlk() member function (you *will* need to finish the sendFile() member function). Use the myCreat(), myOpen(), myRead(), myWrite(), and myClose() wrapper functions for the POSIX functions creat(), open(), read(), write(), and close() to create, open, read from, write to, and close files (and, later in the course, devices like serial ports). You can find

documentation on POSIX functions online and on the Xubuntu Virtual Machine that I created for the class. We will be using Linux this term, and I recommend that everybody use the Xubuntu VM that I have created.

The template already makes some use of the wrappers for these POSIX functions. Mac OS X and Linux support POSIX functions "out-of-the-box". Notice that in my template all direct or indirect calls to the POSIX functions open(), create(), read(), write(), and close() are checked for a return value like –1, indicating that an error occurred or something else went wrong during execution of the function. For any direct or indirect calls to write() and/or read() that you need to add to the template, you must handle a return value of -1 in a suitable way, and also handle an unexpected return value where reasonable to do so. Do not modify lines of code in my template unless modifications to those lines are indicated or you explain in a comment why you have decided to make such modifications. Modifications that you feel improve the code are encouraged but otherwise modifications are discouraged as, among other things, they may affect marking. Discuss with me if you want to make modifications where not indicated.

Assume that the imaginary receiver sends a 'C' to begin the transfer of file information or file contents and always sends an ACK for each block. After the last block of a file is ACKed, send an EOT, which will be NAKed, and then repeat the EOT. So for a single file that needs 4 blocks and has no transmission errors an imaginary receiver requesting CRCs (and being provided information on the file) would send 'C', ACK, 'C', ACK, ACK, ACK, ACK, NAK, ACK, 'C', ACK

Except where indicated in the code, for now don't worry about references to the CAN byte in the specification, and don't worry about timing. Just assume that, for example, an ACK will be quickly received after each block is sent.

We plan to grade your code on both a little-endian system and a big-endian system. So be very careful when writing a CRC16 in Network Byte Order. More on that in class.

[redacted]

September 2021