ENSC 351: Embedded and Real-time Software Systems

Craig Scratchley, Fall 2017

Multipart Project Part 4: Complete State Charts

Please continue working with a partner.

Each pair of students should prepare a receiver state chart that takes care of all possibilities that can arise with the XMODEM protocol. You can use the simple receiver state chart that was included in a solution to Part 2A of the multipart project as your starting point (the .pdf and .smc files are in the *CodedBySmartState* Eclipse application project). With respect to data reception, your receiver state chart should specifically take care of the following possibilities: dropped characters (characters sent but not received), glitch characters (characters received but not sent), and corrupted characters (a character is corrupted if a different character is received than the one that was sent). Recall that SER is used to denote the event of receiving a character from the attached serial port, i.e. the medium. If, perhaps due to dropped characters, the receiver does not receive any characters within a specified time period, the receiver state chart can get a timeout event denoted by TM. When the receiver gets a cancel command via its standard input, or it detects a fatal loss of synchronization or an excessive sequence of errors or timeouts, it should request the cancelation of the file transfer in progress. The request for cancellation should be made when the XMODEM sender should actually be able to receive and process it. The event for a keyboard cancel command is denoted KB_C.

Your complete receiver state chart should be able to co-ordinate with the complete sender state chart that I have provided to you. If enough characters are dropped (or corrupted), the sender can timeout (imagine that a serial cable becomes unplugged for 60 seconds or more). The sender will request the canceling of a transmission because of such a timeout or because of getting a cancel command via the sender's keyboard input or because it received an excessive number of NAKs (or 'C's and NAKs).

For reference, below is a description of the public member functions and variables in *ReceiverX*, the receiver context (referenced as *ctx* in the *Receiver_SS* statechart guard conditions and actions). Feel free if needed to add additional functions and variables to this class, the *SenderX* class, or the *PeerX* superclass.

Function or Variable	**Description**
void can8()	Send 8 CAN characters in a row to the peer, to inform it of the canceling of a file transfer
void getRestBlk()	Call only after an SOH character has been received and posted to the *Receiver_SS* statechart. The function tries to read at least 131 or 132 more characters (depending on whether a checksum or crc16, respectively, is expected) to form a complete block. The function will set or reset a Boolean variable, *goodBlk*. This variable will be made false if either • 131 or 132 bytes, depending, have not yet been received and another byte does not arrive within 1 second of the last byte (or within 1 second of the function being called). • if a good copy of the block has not already been received, 131 or 132 bytes, depending, (or more) are received and the block created using the first 131 or 132 bytes has something wrong with it, like the checksum or crc16 being incorrect. See getRestBlk() in Part 2 solution for details. The function will also set or reset another Boolean variable, *syncLoss*. *syncLoss* will only be set to true when there is a fatal loss of synchronization as described in the XMODEM specification. If *goodBlk* has not already been made false and if *syncLoss* is false, then *goodBlk* will be set to true. The first time each block is received and is good, *goodBlk1st* will be set to true. This is an indication of when the data in a block should be written to disk. If *goodBlk1st* is false, or in any case if more than 131 or 132 bytes (depending) were received, then the *purge*() function (see below) will be called before returning from *getRestBlk*().
void writeChunk()	Write the data in a received block to disk.
void clearCan()	Read and discard contiguous CAN characters. Read characters one-by-one in a loop until either nothing is received over a 2-second period or a character other than CAN is received. If received, send a non-CAN character to the console.
void purge()	The *purge*() subroutine will read and discard characters until nothing is received over a 1-second period.
int errCnt (in PeerX)	A variable that counts a sequence of 10-second timeouts, ACK characters sent due to repeated blocks, or NAK characters sent. An initial NAK does not add to the count. The reception of a good block for the first time resets the count.
bool goodBlk	A Boolean variable that indicates whether the last block received was good or whether it had problems.
bool goodBlk1st	A Boolean variable that indicates that a good copy of a block being sent has been received for the first time. It is an indication that the data in the block can be written to disk.
bool syncLoss	A Boolean variable that indicates whether or not a fatal loss of synchronization has been detected.

The ReceiverX class, like the SenderX class described below, is a subclass of PeerX. Here is a description of the public member functions and variables you can use in PeerX:

Function or Variable	**Description**
void sendByte(uint8_t)	Send a byte to the remote peer across the medium
void tm(int tmSeconds)	set a timeout time at an absolute time *tmSeconds* into the future. That is, determine an absolute time to be used for the next XMODEM timeout by adding *tmSeconds* to the current time.
void tmPush(int tmSeconds)	Store the current absolute timeout, and create a temporary absolute timeout *tmSeconds* into the future.
void tmPop()	Discard the temporary absolute timeout and revert to the stored absolute timeout
void tmRed(int secsToReduce)	Make the absolute timeout time earlier by *secsToReduce* seconds.
const char* result	Points to a c-string giving the "result" when the file transfer ends.
bool Crcflg	Specifies whether CRC16 is being used, or otherwise checksums are used.

Note that the *errCnt* variable described in the derived classes actually lives here in *PeerX*.

As written above, feel free if needed to add additional functions and variables to this base class, the *SenderX* class mentioned on the next page, or the *ReceiverX* class.

For your information, here is a description of the public member functions and variables you can use in *SenderX*, the sender context (referenced as *ctx* in the *Sender_SS* statechart guard conditions and actions):

Member Function or Variable	**Description**
void can8()	Send to the peer 8 CAN characters in pairs, each pair separated by slightly over 1 second, to inform it of the canceling of a file transfer.
void sendBlkPrepNext()	Sends for the first time the block recently prepared (less the last byte), then updates the variable bytesRd and tries to prepare the next block. bytesRd is set to 0 and a block is not actually created if the end of the input file was reached when the current block was created. Finally, after most of the block has been sent, dumps any received glitches and sends the checksum or last byte of crc16, depending.
void resendBlk()	Resends the block that had been sent most recently (less its last byte), and then dumps any received glitches and sends the last byte.
void prep1stBlk()	Tries to prepare the first block. Updates the variable *bytesRd* with the number of bytes that were read from the input file before trying to create the block. Sets *bytesRd* to 0 and does not actually create a block if the input file is empty (i.e. has 0 length).
clearCan()	Read and discard contiguous CAN characters. Read characters one-by-one in a loop until either nothing is received over a 1-second period or a character other than CAN is received. If received, send a non-CAN character to the console.
int errCnt (in PeerX)	Counts the number of times that a block or EOT is resent.
ssize_t bytesRd	The number of bytes last read from the input file.

Since we don't ask you, at this point, to test your Statechart by generating code from it and running the code, you can get full marks for your work as long as you have made a sincere attempt to handle all the issues identified above.

Please have a draft finished and submitted by 8:30 am on Wednesday, November 1st. When you submit your draft, please email me if you would be willing to share your draft with the class for discussion during the lecture that day. If an insufficient number of you volunteer, I may just choose some Statechart submissions for class discussion. After the class discussion, you will have until 12:30 pm on Monday, November 5th to polish up your Statechart for final submission.