ENSC 351: Embedded and Real-time Software Systems

Craig Scratchley, Fall 2021

Multipart Project Part 4: Complete State Chart -- Draft

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 YMODEM protocol. You can use the simple receiver state chart that is being provided to you as your starting point (the .pdf and .smc files are on Canvas). 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 the minimum number of required characters within a specified time period, the receiver state chart can get a timeout event denoted by TM. Note that the myReadcond() function is called with a minimum number of required characters and also with timeout information. When the receiver gets a (keyboard) cancel command via its standard input, or it detects a fatal loss of synchronization or an excessive sequence of errors (including timeouts and an excessively repeated block), it should request the cancelation of the YMODEM session. The request for cancellation should be made when the YMODEM sender should actually be able to receive and process it. The event for a keyboard cancel command is denoted KB_C, and should have priority over a SER event if both are detected at essentially the same time.

Your complete receiver state chart should be able to co-ordinate with the complete sender state chart that I will be provided to you. If enough characters are dropped (or corrupted), the sender can timeout waiting for an expected control character like a 'C' character or NAK character (imagine that a serial cable becomes unplugged for 60 seconds or more). The sender will request the canceling of a session 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 at the beginning of the transfer).

For reference, below is a description of the public member functions and variables in class *ReceiverY*, the receiver context (an object of which is referenced as *ctx* in the receiver state chart actions and guard conditions). If you need to add additional functions and/or variables to this class, the *SenderY* class, or the *PeerY* parent class, please check with Craig first.

Function or Variable	**Description**
void cans()	Send CAN_LEN copies of CAN characters in a row to the peer, to inform it of the cancelling of a session.
void getRestBlk()	Call only after an SOH character has been received and posted to the receiver state chart. The function tries to read at least 132 more characters to form a complete block. The function will set or reset a Boolean variable, *goodBlk*. This variable will be made false if either • 132 bytes 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, 132 bytes (or more) are received and the block completed using the first 132 received bytes has something wrong with it, like the crc16 being incorrect. When it is released, see getRestBlk() in Part 2 solution for details. The function will also set or reset another Boolean member variable, *syncLoss*. *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 YMODEM specification. 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 132 bytes 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 up to (CAN_LEN – 2) 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 PeerY*/	A variable that counts a sequence of 'C' or NAK characters sent, 10-second timeouts, or ACK characters sent due to repeated blocks. The initial 'C' does not add to the count. The reception of a good block for the first time resets the count (see *goodBlk1st* below).
int closeProb	0 indicates that there was no problem closing the file just transferred. A positive value is the errno encountered when trying to close the file. -1 indicates that the program has not yet tried to close the current file (or there is no current file).
bool goodBlk	A Boolean variable that indicates whether the last block received was good (or deemed good) or whether it had problems (and was not deemed to be good).

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 *ReceiverY* class, like the *SenderY* class described below, is a class derived from *PeerY*. Here is a description of the public member functions and variables you can use in *PeerY*:

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 YMODEM 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.
std::string result	Points to a string giving the "result" when the session ends.
int transferringFileD	Descriptor for file being read from or written to.
int errCnt	See description in derived classes for interpretation for each class.

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

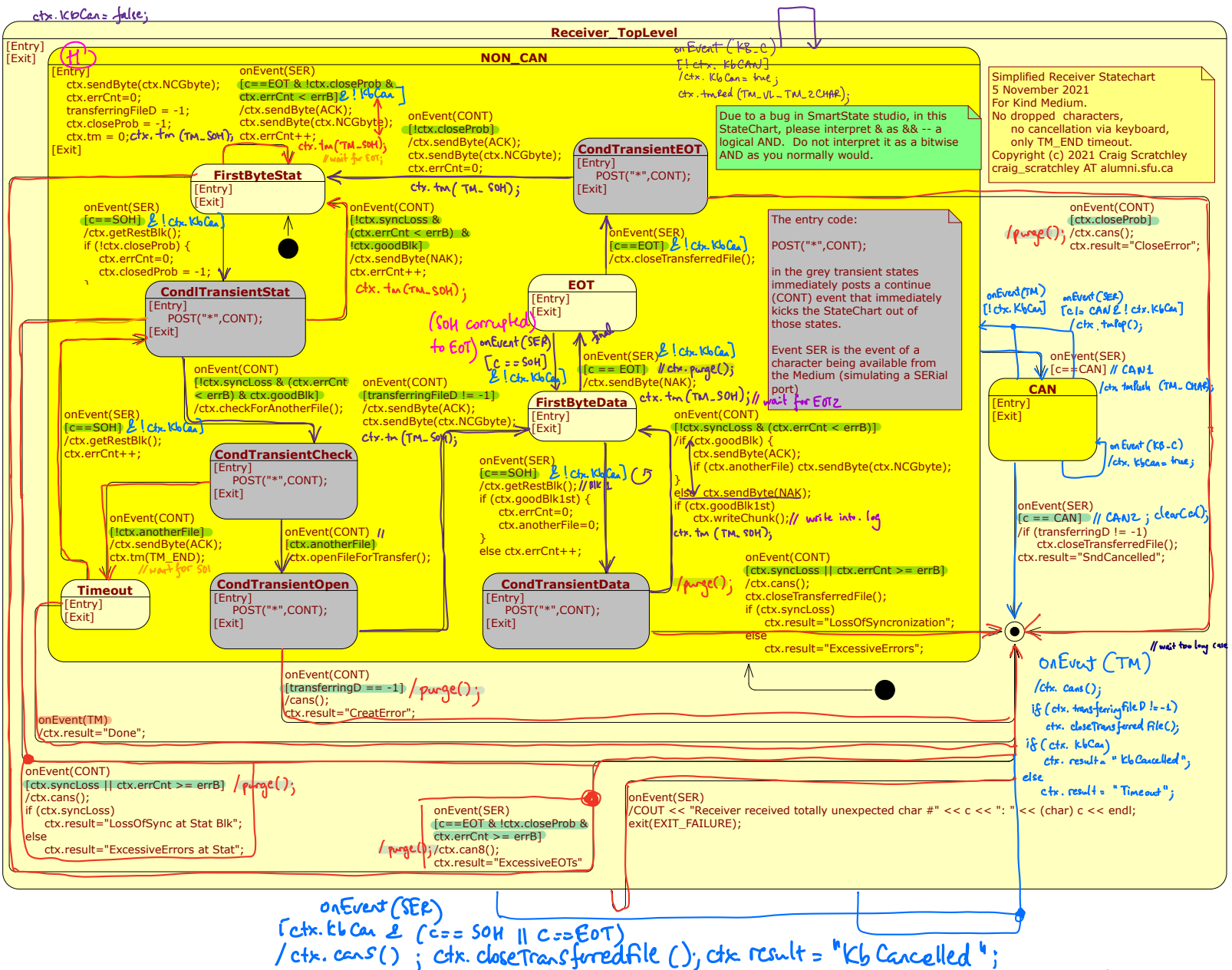
As written above, if you need to add additional functions and/or variables to this base class, the *SenderY* class mentioned on the next page, or the *ReceiverY* class, please check with Craig first. A solution, as will be provided by Craig, uses only the provided functions and variables.

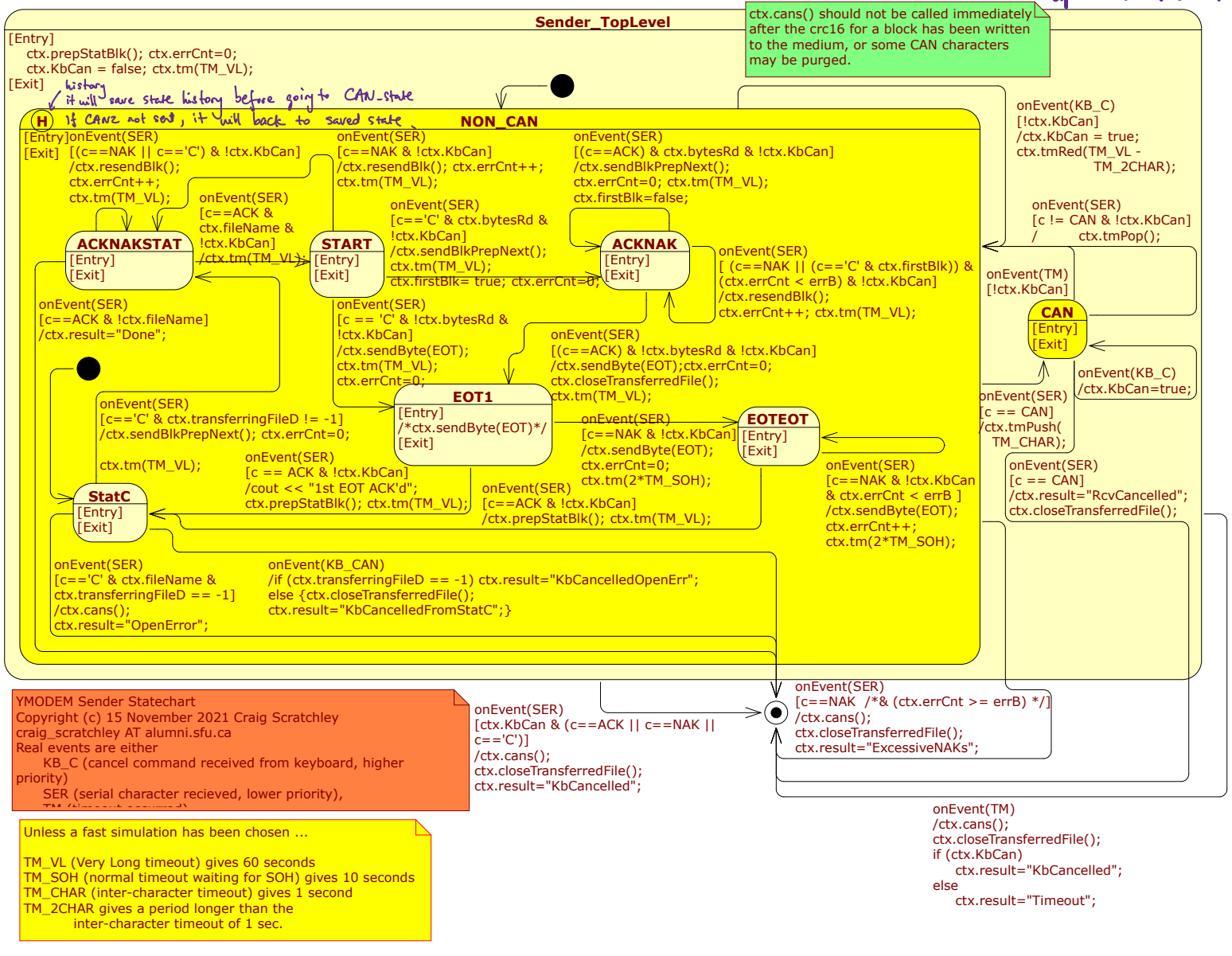
For your information, here is a description of the public member functions and variables used in *SenderY*, the sender context class (an instance of which is referenced as *ctx* in the sender statechart actions and guard conditions):

Member Function or Variable	**Description**
void can8()	Send to the peer CAN_LEN copies of 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 prepared if the input file is empty or the end of the input file was reached when the block recently prepared was filled in. Finally, after the rest of the block has been sent, dumps any received glitches and sends the last byte of block being sent (the last byte of crc16).
void resendBlk()	Resends the block that had been sent most recently or an empty zero-numbered "stat" block (less its last byte in both cases), and then dumps any received glitches and sends the last byte.
clearCan()	Read and discard contiguous CAN characters. Read up to (CAN_LEN – 2) 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 PeerY*/	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.
bool firstBlk	True if first block of file content data has not yet been acknowledged with an ACK.

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 in a reasonable way all the issues identified at the top of this document.

Since this part of the project will not require you to test your work, it should take less time than recent parts of the project.


ctx.kbCan = false;
Receiver_TopLevel
[Entry]
[Exit]
NON_CAN
[Entry]
ctx.sendByte(ctx.NCGbyte);
ctx.errCnt=0;
transferringFileD = -1;
ctx.closeProb = -1;
ctx.tm = 0;ctx.tm (TM_SOH);
[Exit]
onEvent(SER)
[c==EOT & !ctx.closeProb &
ctx.errCnt < errB] & ! kbCan
/ctx.sendByte(ACK);
ctx.sendByte(ctx.NCGbyte);
ctx.errCnt++;
ctx.tm(TM_SOH);
// wait for EOT;
onEvent(CONT)
[!ctx.closeProb]
/ctx.sendByte(ACK);
ctx.sendByte(ctx.NCGbyte);
ctx.errCnt=0;
ctx.tm(TM_SOH);
FirstByteStat
[Entry]
[Exit]
onEvent(SER)
[c==SOH] & ! ctx.kbCan
/ctx.getRestBlk();
if (!ctx.closeProb) {
ctx.errCnt=0;
ctx.closedProb = -1;
}
CondITransientStat
[Entry]
POST("*",CONT);
[Exit]
onEvent(CONT)
[!ctx.syncLoss &
(ctx.errCnt < errB) &
!ctx.goodBlk]
/ctx.sendByte(NAK);
ctx.errCnt++;
ctx.tm(TM_SOH);
CondTransientEOT
[Entry]
POST("*",CONT);
[Exit]
onEvent(SER)
[c==EOT] & ! ctx.kbCan
/ctx.closeTransferredFile();
EOT
[Entry]
[Exit]
(SOH corrupted to EOT)
onEvent(SER)
[c == SOH]
& ! ctx.kbCan
onEvent(SER)
[c == EOT] // ctx.purge();
/ctx.sendByte(NAK);
ctx.tm(TM_SOH); // wait for EOT2
FirstByteData
[Entry]
[Exit]
onEvent(CONT)
[transferringFileD != -1]
/ctx.sendByte(ACK);
ctx.sendByte(ctx.NCGbyte);
ctx.tm(TM_SOH);
onEvent(CONT)
[!ctx.syncLoss & (ctx.errCnt
< errB) & ctx.goodBlk]
/ctx.checkForAnotherFile();
onEvent(SER)
[c==SOH] & ! ctx.kbCan
/ctx.getRestBlk();
ctx.errCnt++;
CondTransientCheck
[Entry]
POST("*",CONT);
[Exit]
onEvent(CONT)
[!ctx.anotherFile]
/ctx.sendByte(ACK);
ctx.tm(TM_END);
// wait for SOT
onEvent(CONT) //
[ctx.anotherFile]
/ctx.openFileForTransfer();
Timeout
[Entry]
[Exit]
CondTransientOpen
[Entry]
POST("*",CONT);
[Exit]
onEvent(SER)
[c==SOH] & ! ctx.kbCan
/ctx.getRestBlk(); // Blk 1
if (ctx.goodBlk1st) {
ctx.errCnt=0;
ctx.anotherFile=0;
}
else ctx.errCnt++;
CondTransientData
[Entry]
POST("*",CONT);
[Exit]
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(); // write into .log
ctx.tm (TM_SOH);
onEvent(CONT)
[ctx.syncLoss || ctx.errCnt >= errB]
/ctx.cans();
ctx.closeTransferredFile();
if (ctx.syncLoss)
ctx.result="LossOfSyncronization";
else
ctx.result="ExcessiveErrors";
/purge();
onEvent(CONT)
[transferringD == -1] /purge();
/cans();
ctx.result="CreatError";
onEvent(TM)
/ctx.result="Done";
onEvent('KB_C')
[!ctx.kbCAN]
/ctx.kbCan = true;
ctx.tmRcd(TM_VL_TM_2CHAR);
Due to a bug in SmartState studio, in this StateChart, please interpret & as && -- a logical AND. Do not interpret it as a bitwise AND as you normally would.
Simplified Receiver Statechart
5 November 2021
For Kind Medium.
No dropped characters,
no cancellation via keyboard,
only TM_END timeout.
Copyright (c) 2021 Craig Scratchley
craig_scratchley AT alumni.sfu.ca
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.closeProb]
/purge();
/ctx.cans();
ctx.result="CloseError";
onEvent(TM)
[!ctx.kbCan]
onEvent(SER)
[c!= CAN & !ctx.kbCan]
ctx.tmTop();
onEvent(SER)
[c!=CAN] // CAN1
/ctx.tmFlush (TM_CHAR);
CAN
[Entry]
[Exit]
onEvent(KB_C)
/ctx.kbCan = true;
onEvent(SER)
[c== CAN] // CAN2 ; clearCo();
/if (transferringD != -1)
ctx.closeTransferredFile();
ctx.result="SndCancelled";
// wait too long case
onEvent(TM)
/ctx.cans();
if (ctx.transferringFileD != -1)
ctx.closeTransferredFile();
if (ctx.kbCan)
ctx.result = "KbCancelled";
else
ctx.result = "Timeout";
onEvent(CONT)
[ctx.syncLoss || ctx.errCnt >= errB]
/ctx.cans();
if (ctx.syncLoss)
ctx.result="LossOfSync at Stat Blk";
else
ctx.result="ExcessiveErrors at Stat";
/purge();
onEvent(SER)
[c==EOT & !ctx.closeProb &
ctx.errCnt >= errB]
/ctx.can8();
ctx.result="ExcessiveEOTs"
/purge();
onEvent(SER)
/COUT << "Receiver received totally unexpected char #" << c << ": " << (char) c << endl;
exit(EXIT_FAILURE);
onEvent(SER)
[ctx.kbCan & (c== SOH || c==EOT)
/ctx.cans() ; ctx.closeTransferredFile(); ctx.result = "Kb Cancelled";


updated - KbCan
Sender_TopLevel
[Entry]
ctx.prepStatBlk(); ctx.errCnt=0;
ctx.KbCan = false; ctx.tm(TM_VL);
[Exit]
history
it will save state history before going to CAN_state
If CANz not sent, it will back to saved state
ctx.cans() should not be called immediately after the crc16 for a block has been written to the medium, or some CAN characters may be purged.
NON_CAN
H
[Entry]onEvent(SER)
[Exit] [(c==NAK || c=='C') & !ctx.KbCan]
/ctx.resendBlk();
ctx.errCnt++;
ctx.tm(TM_VL);
onEvent(SER)
[c==NAK & !ctx.KbCan]
/ctx.resendBlk(); ctx.errCnt++;
ctx.tm(TM_VL);
onEvent(SER)
[(c==ACK) & ctx.bytesRd & !ctx.KbCan]
/ctx.sendBlkPrepNext();
ctx.errCnt=0; ctx.tm(TM_VL);
ctx.firstBlk=false;
onEvent(KB_C)
[!ctx.KbCan]
/ctx.KbCan = true;
ctx.tmRed(TM_VL -
TM_2CHAR);
onEvent(SER)
[c != CAN & !ctx.KbCan]
/ ctx.tmPop();
ACKNAKSTAT
[Entry]
[Exit]
onEvent(SER)
[c==ACK &
ctx.fileName &
!ctx.KbCan]
/ctx.tm(TM_VL);
START
[Entry]
[Exit]
onEvent(SER)
[c=='C' & ctx.bytesRd &
!ctx.KbCan]
/ctx.sendBlkPrepNext();
ctx.tm(TM_VL);
ctx.firstBlk= true; ctx.errCnt=0;
ACKNAK
[Entry]
[Exit]
onEvent(SER)
[(c==NAK || (c=='C' & ctx.firstBlk)) &
(ctx.errCnt < errB) & !ctx.KbCan]
/ctx.resendBlk();
ctx.errCnt++; ctx.tm(TM_VL);
onEvent(TM)
[!ctx.KbCan]
CAN
[Entry]
[Exit]
onEvent(KB_C)
/ctx.KbCan=true;
onEvent(SER)
[c == CAN]
/ctx.tmPush(
TM_CHAR);
onEvent(SER)
[c == CAN]
/ctx.result="RcvCancelled";
ctx.closeTransferredFile();
onEvent(SER)
[c==ACK & !ctx.fileName]
/ctx.result="Done";
onEvent(SER)
[c == 'C' & !ctx.bytesRd &
!ctx.KbCan]
/ctx.sendByte(EOT);
ctx.tm(TM_VL);
ctx.errCnt=0;
onEvent(SER)
[(c==ACK) & !ctx.bytesRd & !ctx.KbCan]
/ctx.sendByte(EOT);ctx.errCnt=0;
ctx.closeTransferredFile();
ctx.tm(TM_VL);
EOT1
[Entry]
/*ctx.sendByte(EOT)*/
[Exit]
EOTEOT
[Entry]
[Exit]
onEvent(SER)
[c=='C' & ctx.transferringFileD != -1]
/ctx.sendBlkPrepNext(); ctx.errCnt=0;
ctx.tm(TM_VL);
onEvent(SER)
[c == ACK & !ctx.KbCan]
/cout << "1st EOT ACK'd";
ctx.prepStatBlk(); ctx.tm(TM_VL);
onEvent(SER)
[c==NAK & !ctx.KbCan]
/ctx.sendByte(EOT);
ctx.errCnt=0;
ctx.tm(2*TM_SOH);
onEvent(SER)
[c==NAK & !ctx.KbCan
& ctx.errCnt < errB]
/ctx.sendByte(EOT);
ctx.errCnt++;
ctx.tm(2*TM_SOH);
onEvent(SER)
[c==ACK & !ctx.KbCan]
/ctx.prepStatBlk(); ctx.tm(TM_VL);
StatC
[Entry]
[Exit]
onEvent(SER)
[c=='C' & ctx.fileName &
ctx.transferringFileD == -1]
/ctx.cans();
ctx.result="OpenError";
onEvent(KB_CAN)
/if (ctx.transferringFileD == -1) ctx.result="KbCancelledOpenErr";
else {ctx.closeTransferredFile();
ctx.result="KbCancelledFromStatC";}
YMODEM Sender Statechart
Copyright (c) 15 November 2021 Craig Scratchley
craig_scratchley AT alumni.sfu.ca
Real events are either
KB_C (cancel command received from keyboard, higher priority)
SER (serial character recieved, lower priority),
onEvent(SER)
[ctx.KbCan & (c==ACK || c==NAK ||
c=='C')]
/ctx.cans();
ctx.closeTransferredFile();
ctx.result="KbCancelled";
onEvent(SER)
[c==NAK /*& (ctx.errCnt >= errB) */]
/ctx.cans();
ctx.closeTransferredFile();
ctx.result="ExcessiveNAKs";
onEvent(TM)
/ctx.cans();
ctx.closeTransferredFile();
if (ctx.KbCan)
ctx.result="KbCancelled";
else
ctx.result="Timeout";
Unless a fast simulation has been chosen ...
TM_VL (Very Long timeout) gives 60 seconds
TM_SOH (normal timeout waiting for SOH) gives 10 seconds
TM_CHAR (inter-character timeout) gives 1 second
TM_2CHAR gives a period longer than the
inter-character timeout of 1 sec.