Lab 11 Binary Synchronous Communication

This laboratory assignment accompanies the book, Embedded Microcomputer Systems: Real Time Interfacing, by Jonathan W. Valvano, published by Brooks-Cole, copyright © 2000.

Goals • Design the high-level communication interface between two or more microcomputers;
• Study the binary synchronous communication (BSC) protocol;
• Investigate the use of a longitudinal redundancy check (LRC) for error detection;
• Implement the "stop and wait automatic repeat request" for error recovery.

Review • Valvano Chapter 14.

Starter files • list from lab 10 plus OC3TEST.C, OC3.C, OC3.H

Background

The half-duplex communication system between two or more microcomputers will be designed in two layers. The first layer, the physical layer performed in a previous lab, allowed single characters to be transmitted. The second layer, to be designed, implemented and tested in this lab, will consist of a simplified binary synchronous communication protocol (BSC). At this level message packets will be transmitted between the two machines. The packet is shown in Figure 8.1. Each packet contains a variable length ASCII string.

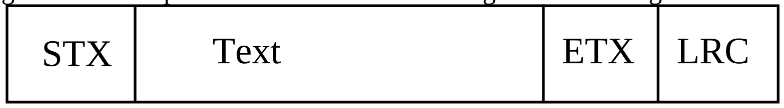


Figure 11.1. Data message packet.

The low-level interface (previous lab) transmits individual bytes, but at this level the hardware/software system transmits packets. The control characters are described below:

- STX start of text: precedes text block (ASCII $02)
- ETX end of text block: ETX is followed by the error checking byte (ASCII $03)
- ACK affirmative acknowledge: last block received correctly (ASCII $06)
- NAK negative acknowledge: last block was received in error (ASCII $15)

There are two mechanisms that allow the transmission of variable amounts of data. This BSC protocol uses start (STX) and stop (ETX) characters to surround a variable amount of data. In this lab, we will limit data transfers to a maximum of 40 characters. The disadvantage of this "termination code" method is that binary data cannot be sent because a data byte might match the termination character (ETX). It is not a problem if we will be sending ASCII characters. The other method uses a byte count to specify the length of a message. Many protocols use a byte count. The S19 records, for example, have a byte count in each line.

There are many ways to check for transmission errors. In this high-level, we will use a longitudinal redundancy check (LRC) or horizontal even parity. The error check byte is simply the ***exclusive-or*** of all the data bytes (not including the STX and ETX). The receiver also performs an ***exclusive-or*** on the data as well as the error check byte. The result will equal zero if the block has been transmitted successfully. Another popular method is checksum, that is simply the modulo$_{256}$ (8 bit) or modulo$_{65536}$ (16 bit) sum of the data packet. The LRC is chosen because of its simplicity. In addition, each byte could have (but doesn't) include even parity. The receiver will respond with an ACK if the message was received properly, and will respond with a NAK if there are any framing, overrun, noise, or LRC errors. The transmitter will send a message and "stop and wait" for either an ACK or a NAK. If an ACK is received, then it can continue. If a NAK is received, or if no response is received after a reasonable delay, then the message is re-transmitted. Because there is a half-duplex physical channel, there is a possibility of a collision. Your low-level software will be able to detect a collision at the transmitter, but now rather than reporting an error to the operator, it will implement retransmission so both transmit processes will complete (obviously one at a time.) You should choose an upper limit (e.g., 3) on the number of times a packet is retransmitted. After 3 tries an error is reported to the operator. Also remember the TA will short the network to ground (preventing transmission). A little harder to deal with is when the TA disconnects the other computer. To solve this fault you will need some time out mechanism (using output compare) to retransmit the packet if an ACK is not received in some reasonable time.

A fun but optional addition to this lab is to implement a chat room with more than 2 nodes. The packet will have to be augmented by adding destination and source addresses. Decide as a group on that protocol. Obviously the receiving node with the matching address will ACK the packet (not everybody). For example,

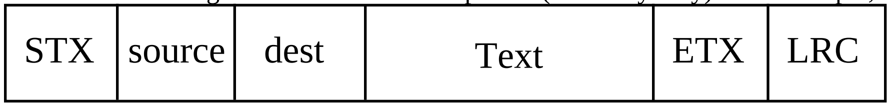


Figure 11.2. Data message packet with the first 2 ASCII characters being the source and destination addresses.

The software for this lab will be divided into three parts. The low-level drivers were completed in the last lab. The high-level "device driver" software will provide support for initialization, transmitting and receiving individual

BSC packets across the network. The packet transmission routines must be built on top of the low-level routines developed in the previous lab. You are free to provide additional low-level support (e.g., routines that access the SCI) that you add to the files **Network.h** and **Network.c**. You will add a periodic interrupt (output compare) to handle any background tasks at the high-level. Two more fifo queues can be used link the high-level background threads (OC interrupts) and foreground thread (the main program and the functions it calls). BSC handshake (ACK and NAK) and packet retransmission is built into this higher layer. At this level, collision is detected either by the low-level software, or by an incorrect LRC. Organize this high-level network software in the **BSC.h** and **BSC.c** files to illustrate the layered nature of the software system. Again, use a naming convention to identify all components of the high-level device driver (e.g., start all function and variable names with BSC). One way to tell if you have divided the software properly into low-level (**Network.c**) and high-level (**BSC.c**), is to consider switching the network from the SCI to a different serial port. This conversion should be possible by modifying only **Network.c** without any changes to **Network.h** and **BSC.c**. Conversely, consider switching the network from “stop and wait” to “go-back-N”. This conversion should be possible by modifying **BSC.c** without any changes to the main program, **Network.c**, **Network.h** and **BSC.h**.

The third part is the main program. In this lab, it will accept keyboard input from the PC. Each character will be echoed immediately on the PC terminal window. When a CR is typed, variable length string is transmitted as a packet using the high-level network interface. Incoming packets received from the network (using your packet receive function) will also be displayed in the PC terminal window. The high-level network interface should strip of the echoed data so that the transmitter window does not see two copies of the operator keyboard input. If both computers attempt to transmit packets simultaneously a collision error will occur. The objective of this part of the lab is to send packets between computers. Again, please note that both machines are not driven by identical software. The main program will look something like the following (you are free to modify as long as the effect is similar.)

```
void main(void){ unsigned char TxString[40],RxString[40],ErrorCode;
   BSCInitialize(“j”);  // address is “j” and calls NetworkInitialize
   while(1){
     if( InStatus() ) {                 // first letter of the TxString is destination
        InString(TxString);             // receive/echo input from PC keyboard
        BSCSendPacket(TxString);}       // begin process of transmitting data
     if(BSCRecvPacket(RxString))        // skip if no incoming data
        OutString(RxString);            // display on PC terminal window
     if( ErrorCode=BSCError()){         // check for network errors
        OutString(“The error code is ” ); OutCh(13);
        OutUdec(ErrorCode);OutChar(13); // message on PC terminal window
        BSCReset();}}}                  // recover from error
```

Note that the `BSCSendPacket()` and `BSCRecvPacket()` do not perform I/O directly to the SCI interface (because they are part of the foreground thread), rather they communicate with the low-level drivers. You could implement the system such that `BSCSendPacket()` and `BSCRecvPacket()` return error codes rather than having a separate `BSCError()` check error function. Once again it is very important that no I/O to the PC (e.g., `InStatus()`, `InChar()`, `OutChar()`, `OutUdec()`, `printf()`, `OutString()`, etc.) occur in any of the `BSCxxx()` functions. In other words, I/O to the PC exists only in the highest most `main()` program of the foreground thread. This modularity provides for easier reuse of the device driver routines. Since the high-level network supports error detection and packet retransmission, a BSC network error, `BSCError()`, will occur only after 3 tries to transmit the packet all fail.

Preparation

Since the hardware was built and tested in the previous lab, no additional hardware is required. Show the syntax-free software for both the high-level device driver and the foreground main program that tests the network.

Procedure

You will probably need two computers (and two boards) for most of this lab. You should find another lab group and test your software system with the other group. IT IS VERY IMPORTANT NOT TO SHARE IMPLEMENTATION DETAILS (just specifications.) YOU MAY NOT SHARE SOFTWARE. The two software solutions must be very different in style and approach. You are not to work as a group of four, and achieve one optimal solution. You can communicate with other groups about channel specifications and network policies (no sharing source code.)

Checkout

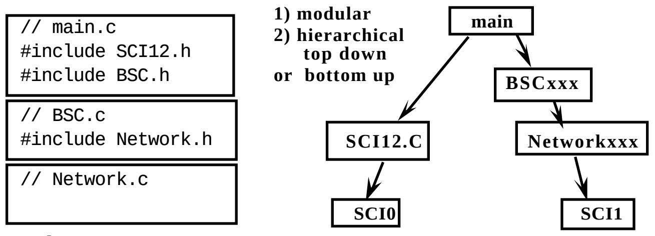
Demonstrate the transmission of character strings between two computer systems. You can cause errors by using a third open collector driver on the transmission line. Also test what happens if the receiving computer is disabled (not connected or not running its software.) You must demonstrate the detection of various errors and the retry feature to your TA. Connect a scope to the channel so that the collusion detection and retry feature can be demonstrated.

Hints

1) Check for overrun errors. If you do get overrun errors, you can add a delay between character transmissions.
2) When testing the BSC layer use an odd number of ASCII digits so that the LRC will be a printing character. For example, the LRC for ASCII "1,2,4" will be $31 xor $32 xor $34 equals $37 which is of course an ASCII "7".
3) If you use the 6812 RAF flag to avoid collisions, then it needs to be built into the low-level Network layer.
4) There are two techniques to developing large complex software systems. The first is to develop the system in a bottom up manner, making VERY small incremental steps. Test the system thoroughly at each step. The second technique is to develop a rich set of debugging instruments that help you visualize the control and data flow within the system.
5) If you change STX to 'S', ETX to 'E', ACK to 'A' and NAK to 'N' then the Lab11 software on one computer can communicate with the Lab10 software on the other computer.

These flowcharts are only to illustrate some of the issues involved in this lab. You are clearly allowed to change any and all aspects of its operation, as long as

1) information is transmitted in packets
2) error detection/recovery is performed
3) the implementation has three layers (main, BSC, previous lab Network)
4) PC I/O occurs only in the main layer.



```
// main.c
#include SCI12.h
#include BSC.h
```

```
// BSC.c
#include Network.h
```

```
// Network.c
```

Figure 11.3. Layered Hierarchy.

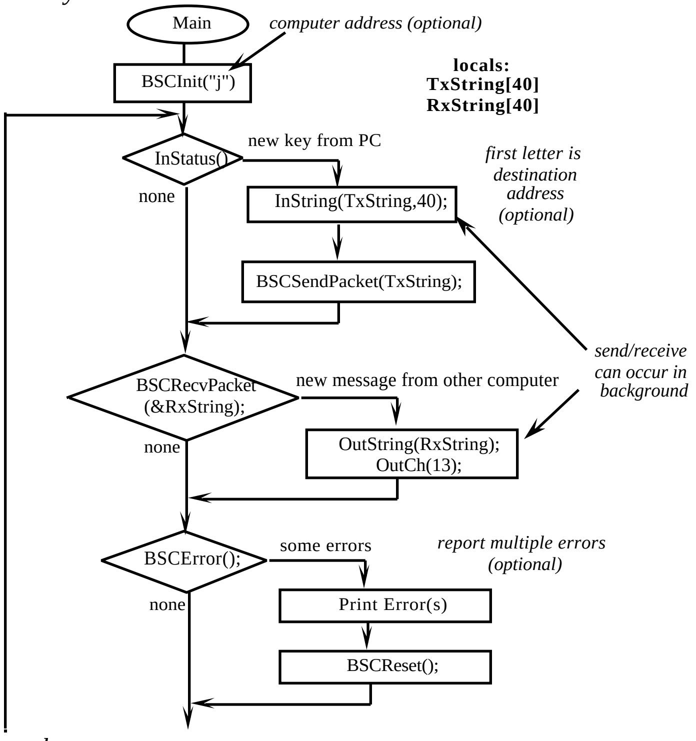

Main
computer address (optional)
BSCInit("j")
locals:
TxString[40]
RxString[40]
InStatus()
new key from PC
none
InString(TxString,40);
first letter is destination address (optional)
BSCSendPacket(TxString);
send/receive can occur in background
BSCRecvPacket (&RxString);
new message from other computer
none
OutString(RxString); OutCh(13);
BSCError();
some errors
report multiple errors (optional)
none
Print Error(s)
BSCReset();

Figure 11.4. Main program layer.

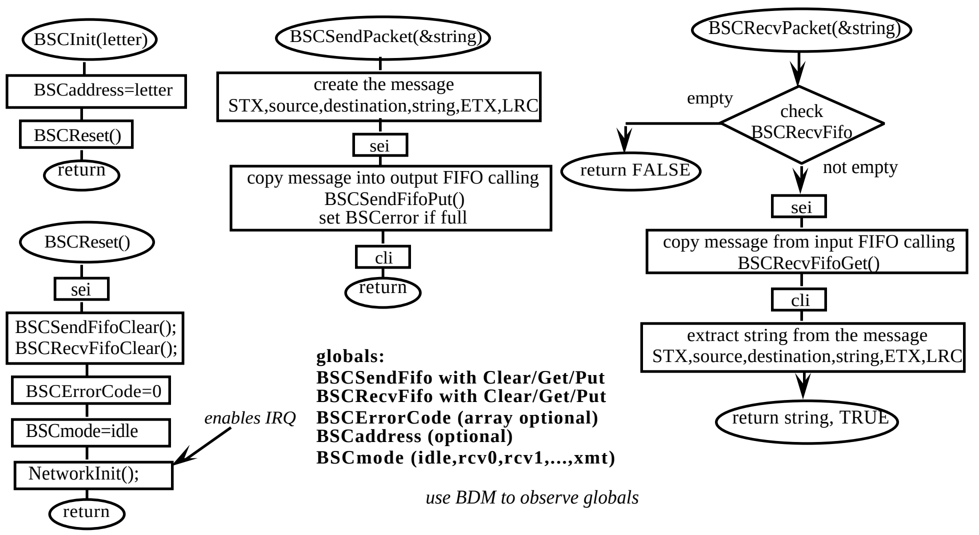


Figure 11.5. BSC programs in the foreground.

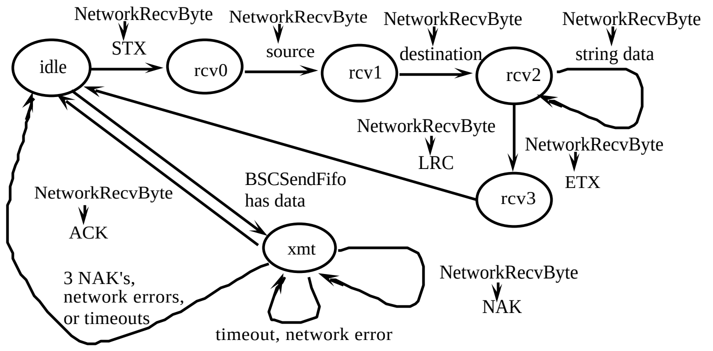


Figure 11.6. BSC state transition graph.

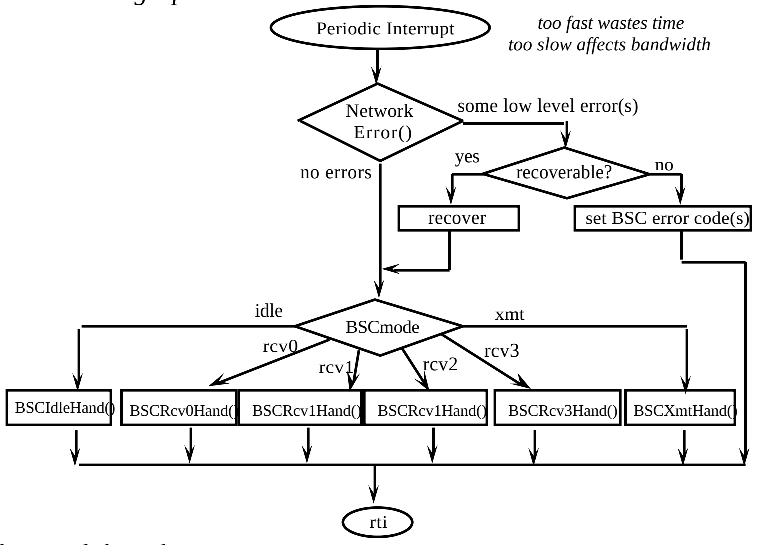
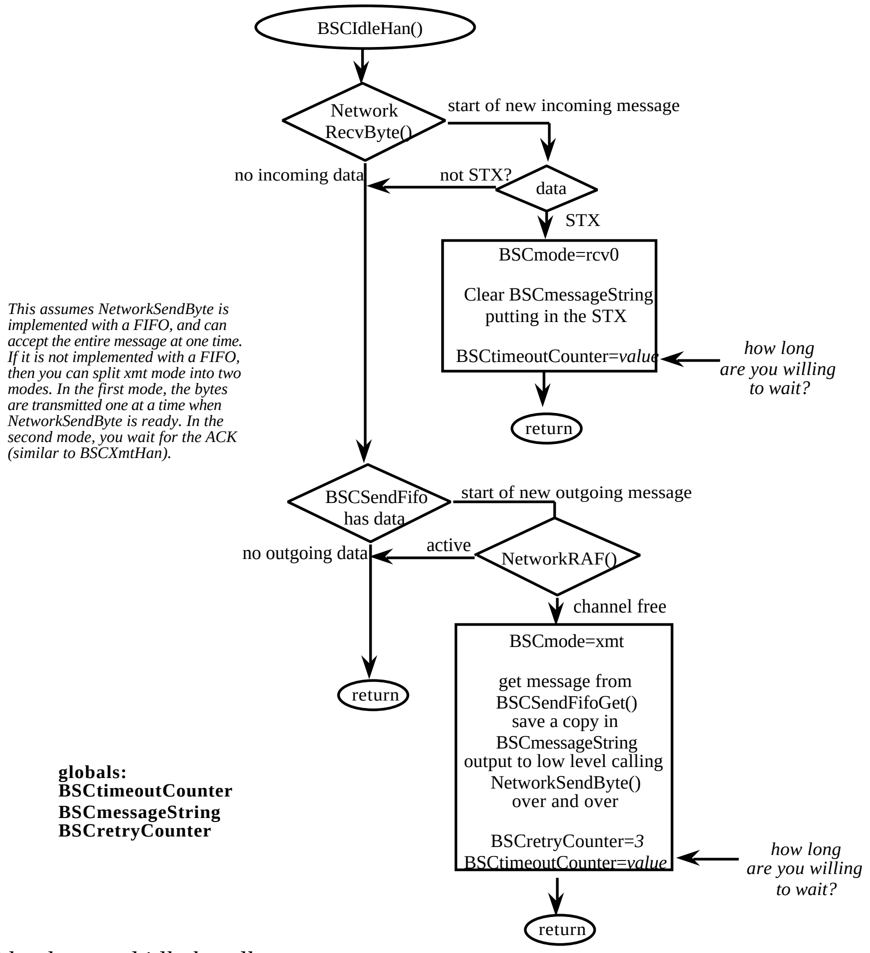


Figure 11.7. BSC background thread.


BSCIdleHan()
Network RecvByte()
start of new incoming message
no incoming data
not STX?
data
STX
BSCmode=rcv0
Clear BSCmessageString putting in the STX
BSCtimeoutCounter=*value*
how long are you willing to wait?
return
BSCSendFifo has data
start of new outgoing message
no outgoing data
active
NetworkRAF()
channel free
return
BSCmode=xmt
get message from BSCSendFifoGet()
save a copy in BSCmessageString
output to low level calling NetworkSendByte() over and over
BSCretryCounter=*3*
BSCtimeoutCounter=*value*
how long are you willing to wait?
return

This assumes NetworkSendByte is implemented with a FIFO, and can accept the entire message at one time. If it is not implemented with a FIFO, then you can split xmt mode into two modes. In the first mode, the bytes are transmitted one at a time when NetworkSendByte is ready. In the second mode, you wait for the ACK (similar to BSCXmtHan).

globals:
BSCtimeoutCounter
BSCmessageString
BSCretryCounter

Figure 11.8. BSC background idle handler.

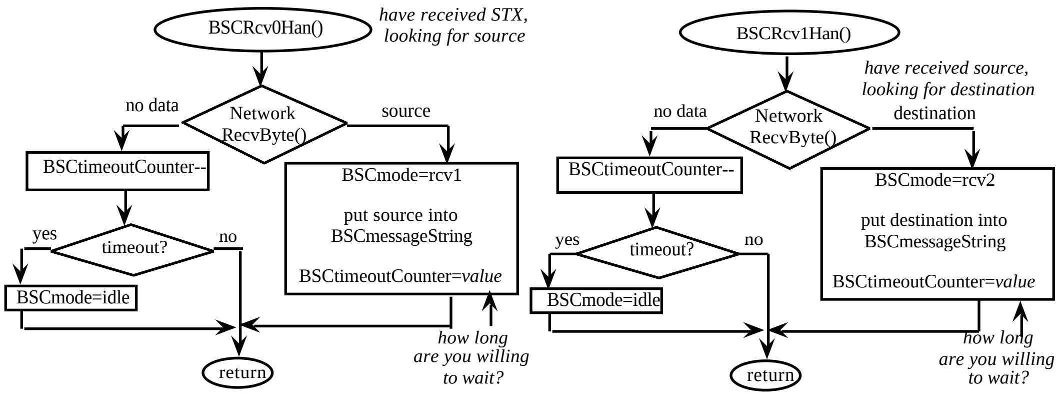

BSCRcv0Han()
have received STX, looking for source
no data
Network RecvByte()
source
BSCtimeoutCounter--
BSCmode=rcv1
put source into BSCmessageString
BSCtimeoutCounter=*value*
yes
timeout?
no
BSCmode=idle
how long are you willing to wait?
return
BSCRcv1Han()
have received source, looking for destination
no data
Network RecvByte()
destination
BSCtimeoutCounter--
BSCmode=rcv2
put destination into BSCmessageString
BSCtimeoutCounter=*value*
yes
timeout?
no
BSCmode=idle
how long are you willing to wait?
return

Figure 11.9. BSC background receiver handler for source and destination information.

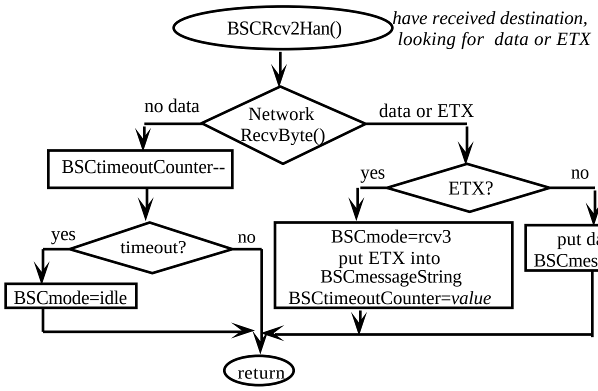

BSCRcv2Han()
have received destination, looking for data or ETX
Network RecvByte()
no data
data or ETX
BSCtimeoutCounter--
ETX?
yes
no
timeout?
yes
no
BSCmode=idle
BSCmode=rcv3 put ETX into BSCmessageString BSCtimeoutCounter=value
put data into BSCmessageString
return

Figure 11.10. BSC background receiver handler for data, and looking for ETX.

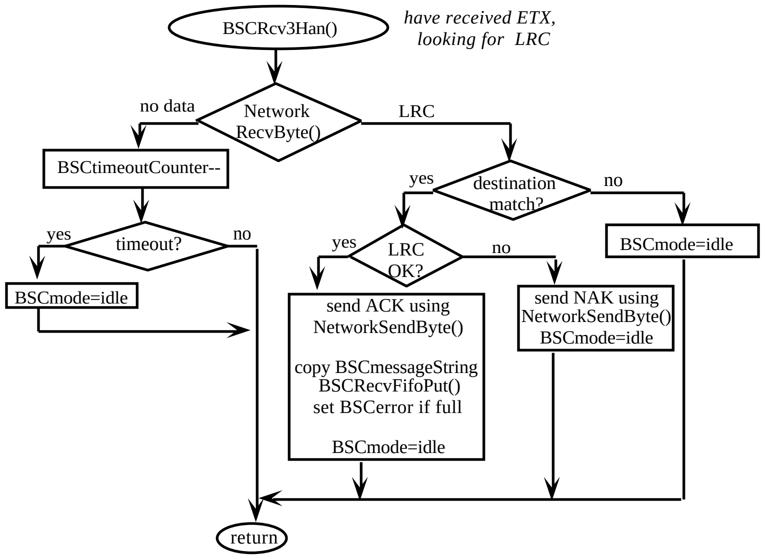

BSCRcv3Han()
have received ETX, looking for LRC
Network RecvByte()
no data
LRC
BSCtimeoutCounter--
destination match?
yes
no
timeout?
yes
no
LRC OK?
yes
no
BSCmode=idle
BSCmode=idle
send ACK using NetworkSendByte() copy BSCmessageString BSCRecvFifoPut() set BSCerror if full BSCmode=idle
send NAK using NetworkSendByte() BSCmode=idle
return

Figure 11.11. BSC background receiver handler for LRC checking.

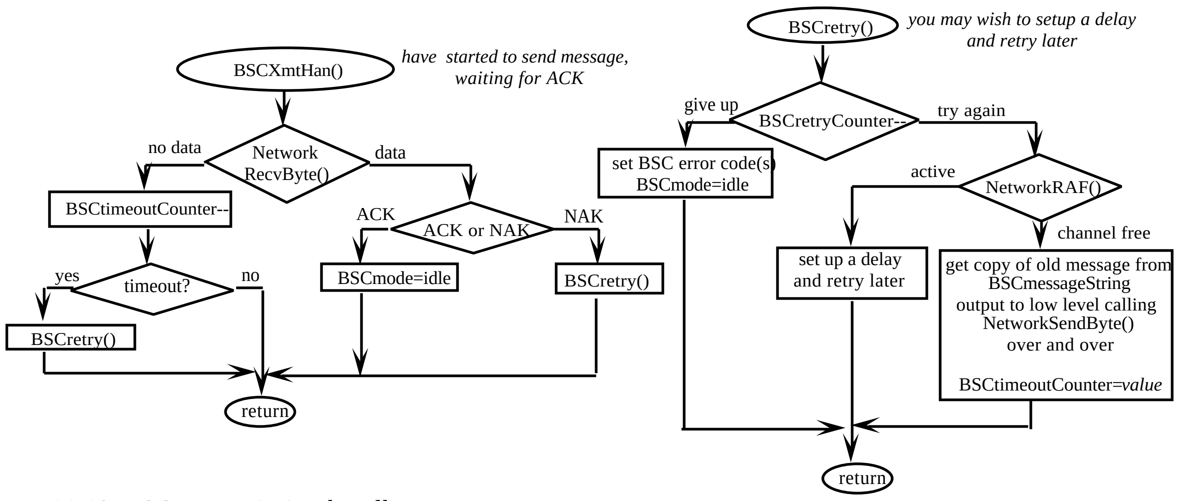

BSCXmtHan()
have started to send message, waiting for ACK
Network RecvByte()
no data
data
BSCtimeoutCounter--
ACK or NAK
ACK
NAK
timeout?
yes
no
BSCmode=idle
BSCretry()
BSCretry()
return
BSCretry()
you may wish to setup a delay and retry later
BSCretryCounter--
give up
try again
set BSC error code(s) BSCmode=idle
NetworkRAF()
active
channel free
set up a delay and retry later
get copy of old message from BSCmessageString output to low level calling NetworkSendByte() over and over BSCtimeoutCounter=value
return

Figure 11.12. BSC transmission handler.