BlackJack Data Link Protocol

(JPL-D-20675)

Interface And Implementation Description

Allen H. Farrington
Allen.Farrington@jpl.nasa.gov

GPS Systems Group
Advanced Radiometric Instruments Group
Tracking Systems and Applications Section (335)
Telecommunications Science and Engineering Division (330)
Engineering and Science Directorate (300)

Jet Propulsion Laboratory
California Institute of Technology

4800 Oak Grove Drive
Pasadena, California 91109

Acknowledgement

The BlackJack Data Link Protocol is a formalization of the Turbo-Rogue Space Receiver (TRSR) Protocol, originally developed at JPL by Charles Dunn of the Advanced Radiometric Instruments Group (Section 335). It has been flown or is about to be flown as the TRSR Protocol on several missions including GPSMET, SRTM, CHAMP, SAC-C, ICESat, FedSat, VCL, GRACE, and Jason-1.

1 Overview

1.1 Introduction

The BlackJack Data Link Protocol is a “wrapper” protocol (similar to HDLC) that ensures the framing and integrity of data transported on byte-oriented links. The format also maintains packet boundaries of data within any serial access device such as data files.

The BlackJack Data Link Protocol only carries the user information. While it only “wraps” user information, it is sensitive to the actual user information being transported; see section *3.1.2, Byte Escapes*, below.

1.2 Rationale

In some cases, data transported onboard spacecraft is subject to unreliable transmission channels. This unreliability expresses itself at all points from data source to data sink. Some examples include:

radiation effects in space-based buffers,
on-board protocol conversion anomalies,
flight computer processing load issues,
noise in space-ground transmissions, and
file transfer protocol anomalies in the ground segment.

The BlackJack Data Link Protocol mitigates this unreliability by providing mechanisms that maintain Framing and Integrity. Framing allows for the detection of dropped or added bytes and Integrity allows for the detection of altered bits within transmitted bytes.

In addition to this Framing and Integrity capability, the BlackJack Data Link Protocol also provides some rudimentary routing support. This support intends to provide the ability for multiple instruments to inter-operate using the Link Protocol to provide source and destination information.

1.3 Heritage

The BlackJack Data Link Protocol is based almost entirely on the Turbo-Rogue Space Receiver (TRSR) protocol. This protocol has been used on flight missions ranging from GPSMet to GRACE. Some of the rarely used capabilities of the TRSR protocol were removed to make the BlackJack Data Link Protocol more effective and easier to implement.

This document, however, does not depend on any previous documents, such as mission-specific documentation of the TRSR Protocol.

2 References

There are no references at this time.

3 Link Protocol Description

In this description, the input data sequence is referred to as the "packet" and the output data sequenced is referred to as the "protocol packet".

Packets provided to the BlackJack Data Link Protocol are processed according to this description prior to transmission. This process is called "wrapping". Protocol Packets provided to the BlackJack Data Link Protocol are processed after reception in a process called "stripping".

This section provides all the required information for implementing the wrapping and stripping functions of the protocol.

There is no defined Link Control Protocol for this protocol.

3.1 Framing

Packets of data are presented to the Protocol and wrapped to achieve framing. Framing allows the receiving mechanism to establish the boundaries of a protocol packet so that integrity checks can be performed. There are four steps to framing:

indicate the start of the protocol packet,
indicate the end of the protocol packet,
sequence protocol packets for re-assembly (optional), and
insure that packet data cannot be misinterpreted as the start or end.

A special single-byte flag called the Open Flag indicates the start of a packet.

The end of protocol packets is indicated by providing the length of the protocol packet. This length is indicated by a single-byte count of header bytes and by a two-byte count of data bytes within the protocol packet.

An optional mechanism is added to allow for spanning packets, that is packets that must reassembled in a particular order. This mechanism is the protocol packet sequence number.

To insure that no packet data or protocol packet header information is misinterpreted as the Open Flag, all the bytes of the protocol packet are subject to being "escaped". These escape sequences turn some single bytes into two-byte patterns that can be uniquely determined on receipt.

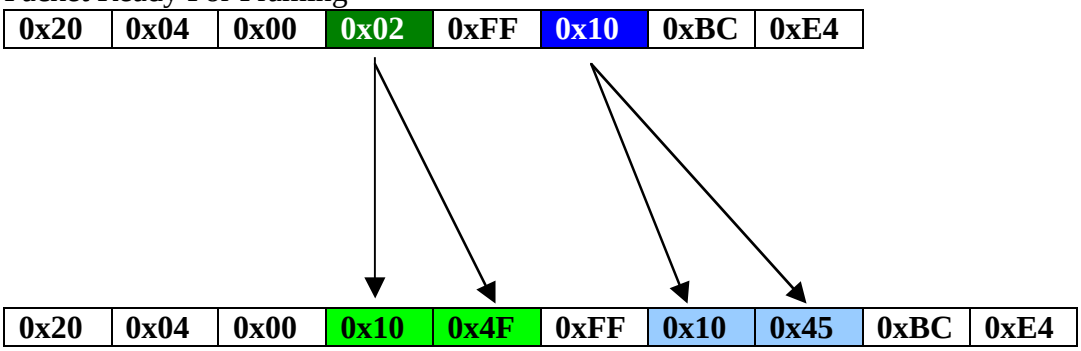
3.1.1 The Open Flag

The Open Flag is the first byte of any protocol packet. It is always the ASCII STX (start of transmission) character. The ASCII STX character has the value of 2 (0x02).

3.1.2 Byte Escapes

To prevent occurrences of the STX character from appearing in the protocol packet, the packet data and associated protocol packet header is "escaped". Any occurrence of the STX character is converted to the two-byte sequence, [DLE (0x10)]["O" (0x4F)]. Additionally, any occurrence of the DLE character is converted to the two-byte sequence, [DLE (0x10)]["E" (0x45)]. No other escape sequences are currently defined and are considered errors in the protocol.

Packet Ready For Framing



Packet After Escaping Operation, prior to adding the Open Flag (STX).

3.1.3 Data And Header Counts

The actual header of a protocol packet is variable length, depending on the features selected for a particular transmission. The length of the header, therefore, is provided as a part of the header. The current definition of the header provided in this document indicates a maximum header length of 10 bytes, but striper implementations should properly ignore unknown header bytes provided that the header length field is assumed correct.

Within the protocol packet, the length of the packet data is indicated by a two-byte header field. The size of the packet data is restricted to 65,535 bytes, the amount that can be carried by a 16-bit size field.

3.1.4 Sequence Numbering

Protocol packets can be optionally sequence numbered so that spanning packets can be wrapped and then reassembled after stripping. The sequence number is a sequentially increasing number that has an undefined epoch. Generally, the epoch is mission specific and will reset on a mission-specific basis. The sequence number is an optional part of the protocol packet header.

Re-assembly of spanning packets is the responsibility of the protocol user, not the protocol itself and is therefore a non-standard feature that must be further specified by mission-specific documentation.

3.2 Routing

To facilitate some routing capabilities during post-processing, two optional indicators are provided in the protocol header. Source Indicator provides a mission-specific value (within some limits) which identifies the source of the protocol packet. Likewise, the Destination Indicator provides the intended sink of the protocol packet. The number format for these indicators is required to be identical across a single mission.

3.2.1 Source Indicator

The source indicator is a two-byte number and appears in the protocol packet header.

3.2.2 Destination Indicator

The destination indicator is a two-byte number and appears in the protocol packet header. The destination value of 0xFFFF represents a broadcast address for systems that implement multiple access communications.

3.3 Integrity

The Flag Byte, the optional Ack/NAck bits, and the optional CRC confirm protocol packet integrity. Taken together, these capabilities allow for the detection and reporting of bit-errors in the protocol packet due to transmission errors.

3.3.1 Flag Byte

The Flag Byte contains bit definitions as follows:

Bit 7 (MSB) **Reserved**
This bit should be set to zero by wrappers. Strippers should ignore this bit.

Bit 6 **RequestAcknowledge**
This bit indicates that a link-level acknowledge of this packet should be generated. This bit must be set to zero for the Acknowledge protocol packets.

Bits 5,4 **CRC Code**
11 – Illegal Value
10 – CRC Present
01 – No CRC Present
00 – Illegal Value
Bits 5 & 4 are mutually exclusive and over-specified to prevent a single-bit error from making a bad packet look good.

Bits 3,2 **Type Code**
11 – Negative Acknowledge Type
10 – Positive Acknowledge Type
01 – Reserved
00 – Normal Link Packet
For the Negative and Positive Acknowledge type codes, there is no data in the data part of the packet, therefore the data length field (see below) is set to zero (0x0000).

Bits 1, 0 **Packet Priority**
A two-bit number from 0 to 3 indicating priority of the packet. Higher numbers indicates higher priority packets with 3 being the highest. The handling of different priority protocol packets is the responsibility of the wrapping and stripping functionality.

3.3.2 CRC-16 Generation

Adding new bytes to the code using the following formula generates the protocol packet Cyclical Redundancy Code (CRC-16):

CRC = (CRC << 8) XOR TableLookup[(CRC >> 8) XOR NewByte]

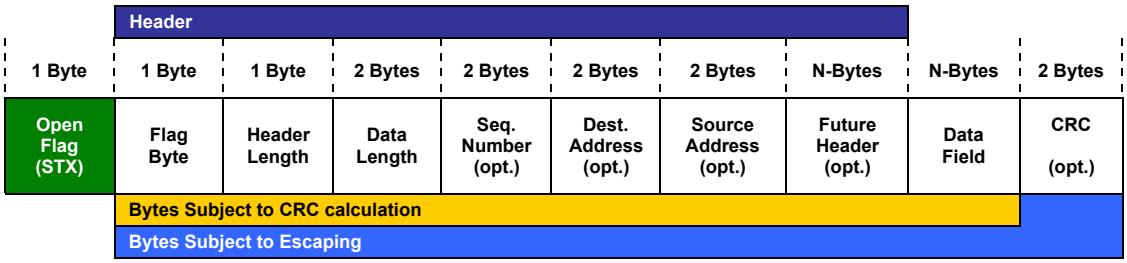
All operations are on unsigned types.

The seed value is zero.

The TableLookup function is based the table presented in Appendix A.

3.4 Complete Packet Description

The protocol packet header is made up of a collection of the fields described previously. Some of the fields are optional. A complete description of the packet is as follows:



The Open Flag is always transmitted first and is not considered a part of the header nor does it participate in the header count. The next byte transmitted, the Flag Byte is considered the first byte of the header and counts towards the header length.

If the Header length is greater than four, then optional portions of the header are present. The optional fields will always appear in the order shown here, Sequence Number, Destination Address, and Source Address, followed by any future extensions of the header. With this version, there are no future header extensions described, however, protocol strippers can be written to account for this future expansion and remain robust while ignoring the future header bytes.

Following the header is the Data Field. There may be up to the number of bytes indicated in the Data Length header field. This size need not correspond to any notion of what the internal Data Field format may be, i.e. it may be padded for telemetry alignment purposes. This type of padding is outside the scope of this definition document.

Following the Data Field is the optional CRC. The presence of a CRC is indicated by the Flag Byte.

All bytes in the packet with the exception of the Open Flag are subject to escaping. This renders decoding by counting bytes unreliable.

3.5 Order Of Operations

The order of the operations must be consistent between the two operations of wrapping and stripping. Section 4, Implementation, contains further details of these operations as summarized here.

3.5.1 Wrapping

The header fields are determined first. The Flag Byte is set to indicate the particular requirements of the transmission. The Header Length field is set to a length, depending on the type of header the transmission requires. The size of the packet data is set into the Data Length field of the header. The rest of the optional fields are set, if required.

The packet data is then appended to the header and if required, the CRC is calculated and appended to the end of the packet data to form the proto-packet for transmission.

Next, the proto-packet is scanned for occurrences of the STX (0x02) and DLE (0x10) characters. Each occurrence is escaped into a two-byte sequence as described previously. The protocol packet is completed for transmission by placing a single STX (0x02) character at the beginning of the packet.

3.5.2 Stripping

For the most part, the operations are reversed from the wrapping operations. Logically, the protocol packet must be de-escaped prior to processing any fields. In implementation, however, it is possible to de-escape the bytes "on-the-fly" while processing the fields.

After bytes are de-escaped, the header length must be processed to find the start of the data section. If the header length exceeds ten, then extended header bytes may be ignored (but included in the CRC calculation), but they must not be counted as data bytes. The Data Length field is used to find the start of the optional CRC.

If at any time during the stripping process an STX character appears in the byte stream before the de-escape operation is applied, then the packet is in error. If this occurs prior to reaching the end of the Data Field or the optional CRC, then an error condition exists.

After stripping, the Flag Byte bit 6 will indicate whether or not an acknowledge has been requested for this protocol packet. If bit 6 is true (1), then a data-less acknowledge packet should be constructed and sent as a response. Acknowledge packets always carry a CRC and a priority of (3) three. The Acknowledge packet Flag Byte is set according to those requirements by the library. In addition, the Type Code fields of the Flag Byte should be set to indicate the status result from the stripping operation, Negative or Positive.

Multi-packet byte streams are processed by the library example provided in this document. The return code from the stripping operation will return how many packets a particular pass completed. In multi-packet streams, retrieving buffers with data will return the higher priority messages first.

4 Link Protocol Implementation Guide

4.1 Overview

This implementation guide is provided to support implementers in the development of BlackJack Data Link Protocol code. It is not the only way to implement the protocol, but represents the "reference" implementation as used on the BlackJack GPS receivers and associated NavaTyrr-class Transceivers.

This implementation guide is provided primarily as C++ source code. Conversion to C source code is possible without changing the actual algorithms.

The actual code is found as an appendix to this document, Appendix B.

4.2 Support Objects

Several objects support both the stripping and wrapping of BJDLP packets.

4.2.1 FlagByte Source Code

The FlagByte object encodes the flag byte of the protocol. The flag byte is a single 8-bit field in the protocol header. This class represents that byte by providing several access, setting, and Boolean check methods. The bit-fields within the byte are all represented by method to set the specific fields as well as to determine their settings within the entire byte.

When the FlagByte object is constructed, it's default value is set to indicate CRC present and that the associated packet is a Link packet.

An operator is provided, operator unsigned char, to allow the FlagByte object to be accessed as a byte as necessary.

4.2.2 Header Source Code

The Header object implements the entire header for the protocol packet. It contains the actual buffer that holds the bytes of the header, output formatting methods, and the storage for each header field.

This class defines a set of access methods for each header field such as the type flag (class FlagByte) and the length fields.

Two operators, operator unsigned char* and operator[] are provided to allow the header's contents to be accessed either as a series of bytes or individually, respectively.

4.2.3 CRC Source Code

The CRC object implements the CRC data object that is transmitted at the end of the protocol packet. It also contains the algorithm for calculating the CRC-16 and the CRC-16 lookup table as static data.

When the CRC is constructed, it is initialized (seeded) to zero (0). A reset method is provided to reset the CRC to the seed value without destroying the object and constructing a new object.

A set of operators is provided for the CRC object to allow it to be manipulated. Four operators, operator(), provide various ways to add bytes to the CRC. Both signed and unsigned data bytes may be added to the CRC, however, all CRC-16 calculations are performed as unsigned operations.

An operator is provided to allow the CRC object to be used as an unsigned short (16-bits).

4.3 Stripping Objects

Two objects are primarily used in stripping the protocol from data after receipt. The RcvPacket object actually strips the protocol packets and places the contained data into a queue of RcvBuffer objects. RcvBuffer objects carry both the data and status information reflecting the results of the stripping operation.

4.3.1 RcvBuffer Source Code

The RcvBuffer object carries both the data extracted from a protocol packet as well as status information reflecting the results of the stripping operation. The status indications include three major categories of status, no status (due to no current packet), oddities, and errors.

No status, or “NoPacketYet” is an indicator that the RcvPacket object uses to indicate that it has not received enough bytes to completely strip a packet yet.

Oddities are a class of results that mean that the stripped protocol packet had an anomaly that may not have been fatal. These anomalies include BigHeader, an indication that the header is more than 10 bytes; LargeData, an indication that the data portion of the packet is more than 2048 bytes; SmallData, an indication that the data portion of the packet is zero; and CRCDidNotCheck, an indication that the CRC did not check.

The CRCDidNotCheck condition is actually considered an error, however, it is grouped with the oddities because the data may be recoverable.

Error conditions include ShortPacket, an indication that an OpenFlag (STX) was received prior to the completion of the processing of a protocol packet; UnknownEscape, an indication that the escape sequence is not one of the two defined; BadCRCBits, an indication that the CRC bits in the FlagByte are an illegal combination; and ConfusedMachine, an indication that the state machine has entered an illegal state (usually SEU induced).

The RcvBuffer object contains methods to access and set the status as well as some Boolean methods used to quickly determine the status settings.

Additionally, the RcvBuffer will assess the settings of the FlagByte and indicate whether or not a response is required. It is up to the application using this object to actually construct the response packet depending on the status indication of the RcvBuffer.

A copy constructor and operators are provided to allow RcvBuffers to be used as STL container objects.

Operators are provided to allow the buffer's data to be accessed as an indexed array or as a pointer to the data buffer itself.

4.3.2 RcvPacket Source Code

The RcvPacket object is the primary algorithm object for stripping protocol packets. This object implements a state machine to decode protocol packets "on the fly". Generally, the state machine transitions from one state to another without moving back. During byte processing, state transitions are minimized.

The states of the machine while processing protocol packets are:

WaitForOpen	- wait for the <STX> indicating the start of a packet,
ReadType	- read the type flag field,
ReadHeaderLen	- read the header length field,
ReadDataLen	- read the two-byte data length field,
ReadSeqNum	- read the optional two-byte sequence number field,
ReadDest	- read the optional two-byte destination address field,
ReadSrc	- read the optional two-byte source address field,
CollectingData	- read the data bytes,
ReadCRC	- read the optional two-byte CRC field, and
GotDLE	- got an escape, process the escape.

For all states except WaitForOpen and GotDLE, the state machine pre-processes each byte looking for two pre-conditions:

The <STX> character, indicating a new packet starting, and
The <DLE> character, indicating an escape sequence start.

The first condition is an error and will be flagged as a short packet error (see RcvBuffer above). The second condition causes a transition into the GotDLE state.

The GotDLE state simply saves the previous state and then processes the next byte by substituting the appropriate byte for the second part of the escape sequence and then transitioning back to the previous state for regular processing. Only two escapes are permitted in this version; other escapes are flagged as errors.

The state machine initializes into the WaitForOpen state and ignores (i.e. drops) any bytes processed until the <STX> character is received. Once the <STX> character is received, it transitions into the ReadType state. Throughout this processing, the bytes processed by the particular states are added to a running CRC to be checked at the end of the packet, if specified.

The ReadType state takes the next byte and interprets it as the TypeFlag. This indicates to the state machine whether or not to look for CRC's at the end of the data (see CollectingData below). After processing one byte, the machine transitions into the ReadHeaderLen state.

The ReadHeaderLen state reads one byte and takes that as the length of the header. Nominally, this is a number between 6 and 10, but may range higher. This version only knows how to process headers up to 10 bytes long, but will ignore larger headers making it backwards compatible in a robust way. After processing one byte, the machine transitions into the ReadDataLen state.

The ReadDataLen state collects two bytes and determines the size of the data field. This size is used later in the CollectingData state. The machine then sets up the return RcvBuffer for this packet and checks the size limits. If the header length has been reached after the processing of these two bytes, the machine transitions into the CollectingData state, otherwise, it transitions into the ReadSeqNum state.

The ReadSeqNum state collects two bytes and stores the sequence number into the packet storage RcvBuffer. If the header length has been reached after the processing of these two bytes, the machine transitions into the CollectingData state, otherwise, it transitions into the ReadDest state.

The ReadDest state collects two bytes and stores the destination address into the packet storage RcvBuffer. If the header length has been reached after the processing of these two bytes, the machine transitions into the CollectingData state, otherwise, it transitions into the ReadSrc state.

The ReadSrc state collects two bytes and stores the source address into the packet storage RcvBuffer. The machine transitions into the CollectingData state after the receipt of these bytes since this version does not understand anymore header bytes.

The CollectingData state processes bytes by adding them to the running CRC until it has collected all the rest of the header bytes. When the number of bytes processed exceeds the value of the HeaderLen field, then the machine starts storing bytes into the RcvBuffer setup in the ReadDataLen state. When the amount of bytes equal to DataLen has been collected, the machine checks the state of the CRC bits within the FlagByte and then, if indicated, transitions to the ReadCRC state. If no CRC is indicated, the machine sets the return value so that the calling process can know that a complete packet was processed. It then returns to the WaitForOpen state.

The ReadCRC state is used to compare the transmitted CRC with the calculated running CRC. The result of that check is stored in the RcvBuffer for this packet and then the machine returns to the WaitForOpen state.

If for any reason during processing the state variable is not one of the above values, a ConfusedMachine return value is indicated and the WaitForOpen state is entered to try and pick up on the next packet.

The RcvPacket object contains a queue that holds RcvBuffers that have been processed out of protocol packets. This queue is automatically fed by the

operator() method which processes the input bytes. The queue is a priority queue and will sort decoded packets according to their priority.

Two operators are provided as the primary interface to this class. The operator() is used to input a buffer of bytes (including single-byte buffers) that is processed for packet contents. The fragmentation of the input buffers is not important to the internal state machine's operation. The operator() returns the number of complete packets in the storage queue so that calling processes can extract all the contents of the queue.

The conversion operator RcvBuffer() is used to extract completed RcvBuffer's from the internal queue. This operator returns the actual RcvBuffer and removes it from the internal queue.

4.4 Wrapping Objects

The primary object used in wrapping packets is the SendPacket object. This object packages buffers of data up into protocol packets. The format of the data to be wrapped is not important to this object. It does, however, produce content-sensitive protocol packets due to the escape sequence generation.

4.4.1 SendPacket Source Code

The SendPacket object contains members used to represent all the components of the protocol packet header and CRC as well as internal buffers used in the construction and manipulation of the protocol packets.

The SendPacket object contains a static member to generate the sequence number, if included in the header. This ensures that the sequence numbers are monotonically increasing regardless of how many SendPacket objects are constructed.

The SendPacket object contains a copy constructor, operator=, and operator< to support STL containers.

The normal constructor includes the input of the data buffer, the size of the buffer, and the priority of the packet being constructed.

After construction, data can be appended to the packet using the AppendPacket() method.

The conversion operator char*() is used to extract the formatted packet from the object to inject into an output byte stream.

Several methods are provided to set the header parameters of the packet:

SetTypeFlag	- allows for setting the entire TypeFlag,
SetTypeToReqAck	- allows only setting ReqAck,
SetTypeToLink	- allows only setting the packet type bits,
SetTypeToNAQ	- allows the construction of Negative Acks,
SetTypeToACK	- allows the construction of Positive Acks,
SetPriority	- allows the setting of the packet priority,
SetToUseSequenceNumber	- turns on Sequence Number generation,
SetDestinationAddress	- sets the Destination Address, and
SetSourceAddress	- sets the Source Address.

The private methods PrepareBuffer() and EscapeAndWrite() are used internally to format the output buffers from the input data.

Appendix A:
CRC Generation Table Lookup Values

Example: If the lookup key, ((CRC >> 8) XOR NewByte), is 0x68, the lookup value is 0xEDAE.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0X	0000	1021	2042	3063	4084	50A5	60C6	70E7	8108	9129	A14A	B16B	C18C	D1AD	E1CE	F1EF
1X	1231	0210	3273	2252	52B5	4294	72F7	62D6	9339	8318	B37B	A35A	D3BD	C39C	F3FF	E3DE
2X	2462	3443	0420	1401	64E6	74C7	44A4	5485	A56A	B54B	8528	9509	E5EE	F5CF	C5AC	D58D
3X	3653	2672	1611	0630	76D7	66F6	5695	46B4	B75B	A77A	9719	8738	F7DF	E7FE	D79D	C7BC
4X	48C4	58E5	6886	78A7	0840	1861	2802	3823	C9CC	D9ED	E98E	F9AF	8948	9969	A90A	B92B
5X	5AF5	4AD4	7AB7	6A96	1A71	0A50	3A33	2A12	DBFD	CBDC	FBBF	EB9E	9B79	8B58	BB3B	AB1A
6X	6CA6	7C87	4CE4	5CC5	2C22	3C03	0C60	1C41	EDAE	FD8F	CDEC	DDCD	AD2A	BD0B	8D68	9D49
7X	7E97	6EB6	5ED5	4EF4	3E13	2E32	1E51	0E70	FF9F	EFBE	DFDD	CFFC	BF1B	AF3A	9F59	8F78
8X	9188	81A9	B1CA	A1EB	D10C	C12D	F14E	E16F	1080	00A1	30C2	20E3	5004	4025	7046	6067
9X	83B9	9398	A3FB	B3DA	C33D	D31C	E37F	F35E	02B1	1290	22F3	32D2	4235	5214	6277	7256
AX	B5EA	A5CB	95A8	8589	F56E	E54F	D52C	C50D	34E2	24C3	14A0	0481	7466	6447	5424	4405
BX	A7DB	B7FA	8799	97B8	E75F	F77E	C71D	D73C	26D3	36F2	0691	16B0	6657	7676	4615	5634
CX	D94C	C96D	F90E	E92F	99C8	89E9	B98A	A9AB	5844	4865	7806	6827	18C0	08E1	3882	28A3
DX	CB7D	DB5C	EB3F	FB1E	8BF9	9BD8	ABBB	BB9A	4A75	5A54	6A37	7A16	0AF1	1AD0	2AB3	3A92
EX	FD2E	ED0F	DD6C	CD4D	BDAA	AD8B	9DE8	8DC9	7C26	6C07	5C64	4C45	3CA2	2C83	1CE0	0CC1
FX	EF1F	FF3E	CF5D	DF7C	AF9B	BFBA	8FD9	9FF8	6E17	7E36	4E55	5E74	2E93	3EB2	0ED1	1EF0

Appendix B:
Source Code

```
// FlagByte.h
//
// Flag Byte
//
// Interface for the FlagByte class.
// This class defines the packet header flag byte type.


// only include this file once per compilation
#ifndef        _FLAGBYTE_H_
#define        _FLAGBYTE_H_


class FlagByte
{
     private:
          // the actual type byte storage
          unsigned char  Val;

     public:
          // constants to support bit decoding
          enum
          {
               ReqACK         = 0x40,             // Acknowledge

               CRCMask        = 0x30,             // CRC bits Mask
               CRCPresent     = 0x20,             // CRC Present
               NoCRC          = 0x10,             // CRC Not Present

               PrioMask  = 0x03              // Priority bits Mask
          };

          // distinct packet types
          typedef   enum
          {
               TypeMask  = 0x0C              // Type bits Mask
          };

          typedef   enum
          {
               NAQPkt         = 0x0C,             // Negative Acknowledge Packet Type
               ACKPkt         = 0x08,             // Positive Acknowledge Packet Type
               ReservedPkt    = 0x04,             // Reserved Packet Type
               TransportPkt   = 0x00              // Normal Data Transport Packet Type
          }
          PktTypes;

          // constructors
          // the default construction provides a CRC and sets the type as a DataPkt
          FlagByte( unsigned char theFlagValue  = CRCPresent | TransportPkt );

          // a placeholder, it does nothing
          ~FlagByte();

          // operators
          // this is the unsigned char conversion operator
          // it returns the type byte
          operator unsigned char();

          // get/set methods
          // the Set method allows for the setting of the flag
          // byte directly, without regard to the contents or
          // meaning of the bits of the flag
```

```
        // the Get method returns the flag byte, similar to the operator above
        void SetFlag(  unsigned char theFlag );
        unsigned char  GetFlag() const;

        // these two methods allow you to check and set (respectively)
        // the acknowledge bit of the flag byte
        bool isReqACK() const    { return Val & ReqACK;   }                          // true if packet is an acknowledge packet
        void SetReqACK(     bool yesno );                                           // argument, yesno, true = Ack packet, false = normal

        // these three methods allow you to check, set, and test the validity (respectively)
        // of the CRC bit of the flag byte
        bool hasCRC()  const     { return !( ( Val & CRCMask ) == NoCRC );    }     // true if CRC is present
                                                                                     // illegal bit combinations are set to have the CRC so that
                                                                                     // malformed packets will be tossed along with the next packet
        void SetCRC(        bool yesno );                                            // argument, yesno, true = CRC on, false = CRC off
        bool ValidCRCBits() const;                                                   // true if CRC bits are a valid combination

        // these methods allow you to manipulate and check the type bits of the flag byte
        int       Type()    const     { return Val & TypeMask;          }          // returns the type of packet, properly masked
        bool isType(        int  aType )   const     { return ( Val & TypeMask ) == aType;   }   // compares the type with an integer type (operator==)
        void SetType(  int theType );                                                // allows you to set the type, properly masked

        // these methods allow for the setting and checking of the type bits of the packet
        bool isNAQPkt()     const     { return Val & NAQPkt;   }                     // returns true if the packet is a NAQ type
        void SetToNAQ();                                                             // sets the packet to be NAQ type

        bool isACKPkt()     const     { return Val & ACKPkt;   }                     // returns true if the packet is a ACK type
        void SetToACK();                                                             // sets the packet to be ACK type

        bool isTransportPkt()    const     { return Val & TransportPkt;  }           // returns true if the packet is a Transport type
        void SetToTransportPkt();                                                    // sets the packet to be Transport type

        // these methods allow for the setting and checking of the priority bits of the packet
        int       Prio()    const     { return Val & PrioMask; }                     // returns the priority, properly masked
        void SetPrio( unsigned char Priority );                                      // allows you to set the priority, properly masked
};

#endif    _FLAGBYTE_H_
```

```
// FlagByte.cpp
//
// Flag Byte
//
// Implementation for the FlagByte class.
// This class defines the packet header flag byte type.


#include "FlagByte.h"

#pragma export on

// constructors
// default constructor
// construct with a value
FlagByte::FlagByte( unsigned char theFlagValue ) :
     Val( theFlagValue )
{}


// default destructor
FlagByte::~FlagByte()
{}


// operators
FlagByte::operator unsigned char()
{
      return Val;
}


// get/set methods
void FlagByte::SetFlag( unsigned char theFlag )
{
     Val = theFlag;
}


unsigned char FlagByte::GetFlag() const
{
     return Val;
}


// bit manipulation methods
// set the Request Acknowledge Bit
void FlagByte::SetReqACK( bool yesno )
{
     if( yesno )
     {
          Val |= ReqACK;
     }
     else
     {
          Val &= ~ReqACK;
     }
}


// set the CRC Bit
void FlagByte::SetCRC( bool yesno )
{
```

```
     Val &= ~CRCMask;                           // clear the CRC Bits
     Val |= ( yesno ) ? CRCPresent : NoCRC;  // set the appropriate bits
}


// check the validity of the CRC bits
bool FlagByte::ValidCRCBits() const
{
     bool retVal = false;

     switch( Val & CRCMask )
     {
          case CRCPresent:
          case NoCRC:
               retVal = true;
               break;

          default:
               retVal = false;
               break;
     }

     return retVal;
}


// set the Type to a value
void FlagByte::SetType( int theType )
{
     Val &= ~FlagByte::TypeMask;                    // clear the type bits
     Val |= ( theType & FlagByte::TypeMask );     // set only the type bits
}


// set the type to Negative Response (NAQ) packet type
void FlagByte::SetToNAQ()
{
     Val &= ~FlagByte::TypeMask;                    // clear the type bits
     Val |= NAQPkt;                                 // set only the type bits to NAQ
}


// set the type to Positive Response (ACK) packet type
void FlagByte::SetToACK()
{
     Val &= ~FlagByte::TypeMask;                    // clear the type bits
     Val |= ACKPkt;                                 // set only the type bits to ACK
}


// set the type to Transport packet type
void FlagByte::SetToTransportPkt()
{
     Val &= ~FlagByte::TypeMask;                    // clear the type bits
     Val |= TransportPkt;                                    // set only the type bits to NAQ
}


// set the priority
void FlagByte::SetPrio( unsigned char Priority )
{
     Val &= ~FlagByte::PrioMask;                    // clear the priority bits
     Val |= ( FlagByte::PrioMask & Priority );    // set the priority bits
```

```
}

#pragma export reset
```

```
// Header.h
//
// Header
//
// Interface for the Header class.
// This class defines the packet header and supports headers up to
// 10 Bytes in length to include
// <TypeFlag>                1 Byte         (required)
// <HeaderLength>            1 Byte         (required)
// <DataLength>                   2 Bytes        (required)
// <SequenceNumber>          2 Bytes        (optional)
// <DestinationAddress>      2 Bytes        (optional)
// <Source Address>          2 Bytes        (optional)


// only include this file once per compilation
#ifndef        _HEADER_H_
#define        _HEADER_H_


// include support for the type flags class
#include "FlagByte.h"


#define   MAX_HEADER     10


class Header
{
     private:
          // the buffer where the header is assembled
          // the maximum size of a TRSR Header is 10 Bytes
          unsigned char  hBuffer[ MAX_HEADER ];

     protected:
          // the type flag
          // see TRSRFlagByte.h for a complete description
          FlagByte       TypeFlag;

          // the length of this header
          // current range is 4-10 Bytes in increments of two
          // the length cooresponds to the inclusion of the optional Bytes described above
          // and does not include the optional CRC bytes at the end of the packet
          unsigned char  HeaderLength;

          // the length of the encapulated data, not including escape sequences
          unsigned short DataLength;

          // an optional sequence number on the output packet
          unsigned short SequenceNumber;

          // an optional destination receiver address
          unsigned short DestinationAddress;

          // an optional source receiver address
          unsigned short SourceAddress;

     public:
          // constants used in escaping packets
          enum
          {
               OpenFlag  = 0x02,        // this byte starts every packet
               DLE            = 0x10,        // this byte preceeds escape sequences
```

```
        OpenESC        = 0x4F,        // this is the second byte of the escape sequence for 0x02
        ESCESC         = 0x45         // this is the second byte of the escape sequence for 0x10

    };

    // default values for the source and destination receivers
    enum
    {
        Rcvr = 0x8003,
        Host = 0x8081
    };

    // constructors
    // makes a null packet with a header length of 4 and a data length of zero (0)
    Header();

    // makes a packet with a header length of 4 and the specified data length
    Header(   unsigned short theDataLength                                                                                    );

    // makes a packet with a header length of 6, the specified data length, and the specified SequenceNumber
    Header( unsigned short theDataLength, unsigned short theSequenceNumber                                                  );

    // makes a packet with a header length of 6, the specified data length, the specified SequenceNumber, and
    // the specified source and destination addresses
    Header( unsigned short theDataLength, unsigned short theSequenceNumber, unsigned short theDestinationAddress, unsigned short theSourceAddress );

    // this destructor is a place holder, it does nothing extra
    ~Header();

    // operators
    // this is the unsigned char, array access operator
    // it returns the byte of the header specified by the argument
    unsigned char operator[] (int i);

    // this is the the unsigned char* conversion operator
    // it returns a pointer to the buffer after the buffer is properly
    // filled-out from the internal representations of the buffer contents
    operator unsigned char*();

private:
    // this function provides a formatting function to place the internal
    // representations of the header contents into the header buffer
    void           Bufferize();

public:
    // set methods - allow manipulation of the internal representation of the header
    // sets the type flag with a specific value the value is not checked for correctness
    void SetTypeFlag(            unsigned char  theTypeFlag );

    // same as SetTypeFlag(), provided for backwards compatibility
    void SetType(                unsigned char  theType );

    // these methods set the internal representations of their coresponding elements
    void SetHeaderLength(        unsigned char  theHeaderLength );
    void SetDataLength(          unsigned short theDataLength );
    void SetSequenceNumber(      unsigned short theSequenceNumber );
    void SetDestinationAddress(  unsigned short theDestinationAddress );
    void SetSourceAddress(       unsigned short theSourceAddress );

    // get methods - return the internal representations of their coresponding elements
    unsigned char  GetTypeFlag() const;
    unsigned char  GetHeaderLength() const;
    unsigned short GetDataLength() const;
```

```
        unsigned short GetSequenceNumber() const;
        unsigned short GetDestinationAddress() const;
        unsigned short GetSourceAddress() const;
};

#endif    _HEADER_H_
```

```
// Header.cpp
//
// Header
//
// Implementation for the TRSRHeader class.
// This class defines the TRSR packet header.


#include <cstring>

#include "Header.h"


using namespace std;


// constructors
// default constructor
Header::Header() :
     HeaderLength( 4 ),
     DataLength( 0 ),
     SequenceNumber( 0 ),
     DestinationAddress( 0 ),
     SourceAddress( 0 )
{}


// constructor with only lengths
Header::Header(     unsigned short theDataLength ) :
     HeaderLength( 4 ),
     DataLength( theDataLength ),
     SequenceNumber( 0 ),
     DestinationAddress( 0 ),
     SourceAddress( 0 )
{}


// constructor with a sequence number
Header::Header( unsigned short theDataLength, unsigned short theSequenceNumber ) :
     HeaderLength( 6 ),
     DataLength( theDataLength ),
     SequenceNumber( theSequenceNumber ),
     DestinationAddress( 0 ),
     SourceAddress( 0 )
{}


// constructor with a destination address and source address
Header::Header( unsigned short theDataLength, unsigned short theSequenceNumber, unsigned short theDestinationAddress, unsigned short theSourceAddress ) :
     HeaderLength( 10 ),
     DataLength( theDataLength ),
     SequenceNumber( theSequenceNumber ),
     DestinationAddress( theDestinationAddress ),
     SourceAddress( theSourceAddress )
{}


// default destructor
Header::~Header()
{}


// operators
```

```
// return individual buffer locations
unsigned char Header::operator[] (int i)
{
     Bufferize();

     return hBuffer[ i ];
}


// return the header buffer
Header::operator unsigned char*()
{
     Bufferize();

     return hBuffer;
}


// place the data into the buffer
void Header::Bufferize()
{
     // this method places the required data into the buffer and avoids structure
     // alignment and padding issues

     // an index into the buffer
     unsigned char* pBuffer = hBuffer;

     // fill in the type byte
     *(pBuffer++) = TypeFlag.GetFlag();

     // copy each element into the buffer
     // this technique, while tedious, avoids size problems and alignment padding issues
     pBuffer = reinterpret_cast<unsigned char*>( memcpy( pBuffer, reinterpret_cast<unsigned char*>( &HeaderLength ),              sizeof( HeaderLength ) ) )              + sizeof(
HeaderLength );
     pBuffer = reinterpret_cast<unsigned char*>( memcpy( pBuffer, reinterpret_cast<unsigned char*>( &DataLength ),          sizeof( DataLength ) ) )           + sizeof(
DataLength );
     pBuffer = reinterpret_cast<unsigned char*>( memcpy( pBuffer, reinterpret_cast<unsigned char*>( &SequenceNumber ),       sizeof( SequenceNumber ) ) )       + sizeof(
SequenceNumber );
     pBuffer = reinterpret_cast<unsigned char*>( memcpy( pBuffer, reinterpret_cast<unsigned char*>( &DestinationAddress ),   sizeof( DestinationAddress ) ) )   + sizeof(
DestinationAddress );
     pBuffer = reinterpret_cast<unsigned char*>( memcpy( pBuffer, reinterpret_cast<unsigned char*>( &SourceAddress ),        sizeof( SourceAddress ) ) )             + sizeof(
SourceAddress );
}


// set methods
void Header::SetTypeFlag(               unsigned char  theTypeFlag )
{
     TypeFlag.SetFlag( theTypeFlag );
}


void Header::SetType(                   unsigned char  theType )
{
     TypeFlag.SetType( theType );
}


void Header::SetHeaderLength(      unsigned char  theHeaderLength )
{
     HeaderLength = theHeaderLength;
}
```

```
void Header::SetDataLength(             unsigned short theDataLength )
{
     DataLength = theDataLength;
}


void Header::SetSequenceNumber(         unsigned short theSequenceNumber )
{
     SequenceNumber = theSequenceNumber;
}


void Header::SetDestinationAddress(     unsigned short theDestinationAddress )
{
     DestinationAddress = theDestinationAddress;
}


void Header::SetSourceAddress(          unsigned short theSourceAddress )
{
     SourceAddress = theSourceAddress;
}


// get methods
unsigned char  Header::GetTypeFlag() const
{
     return TypeFlag.GetFlag();
}


unsigned char  Header::GetHeaderLength() const
{
     return HeaderLength;
}


unsigned short Header::GetDataLength() const
{
     return DataLength;
}


unsigned short Header::GetSequenceNumber() const
{
     return SequenceNumber;
}


unsigned short Header::GetDestinationAddress() const
{
     return DestinationAddress;
}


unsigned short Header::GetSourceAddress() const
{
     return SourceAddress;
}
```

```
// CRC.h
//
// CRC Support
//
// Interface for the CRC class.
// This class provides standard CRC-16 generation and checking.

// only include this file once per compilation
#ifndef        _CRC_H_
#define        _CRC_H_


class CRC
{
     protected:
          // a table of lookup values used to generate the CRC
          static const unsigned short Table[ 256];

          // the current value of the CRC
          unsigned short Val;

     public:
          // constructor - initializes the internal member, Val
          // to the provided value, defaults to zero
          CRC( unsigned short Value = 0 );

          // destructor does nothing, this is a place-holder
          ~CRC();

          // resets the CRC to zero
          void Reset();

          //operators
          // the basic CRC operation is non-trivial and involves a lookup and some
          // mathematical (XOR & shifts (integer multiply & divide) operations:
          //  NewCRC = ( CurrentCRC >> 8 ) XOR ( TableLookup[ ( CurrentCRC << 8 ) XOR NewCharacter )

          // this operator is provided for the type char (signed) and passes its
          // argument to the operator()( const unsigned char c ) below
          unsigned short operator() ( const char c );

          // this operator takes a single character and adds it to the CRC in the complex
          // manner described above
          unsigned short operator() ( const unsigned char c );

          // this operator is provided for a buffer of type char (signed) and passes its
          // arguments to the operator()( const unsigned char* cp, const unsigned char* End ) below
          unsigned short operator() ( const char *cp, const char *End );

          // this operator takes a buffer and adds each character to the CRC sequentially
          // using the same operations described above
          unsigned short operator() ( const unsigned char*cp, const unsigned char *End );

          // this operator is the unsigned short conversion operator for this type.
          // it returns the CurrentCRC
          operator unsigned short &();
};

#endif    _CRC_H_
```

```
// CRC.cpp
//
// CRC Support
//
// Implementation for the CRC class.
// This class provides standard CRC-16 generation and checking.


#include "CRC.h"

#pragma export on

// CRC-16 constant array...
// from Usenet contribution by Mark G. Mendel, Network Systems Corp.
// (ihnp4!umn-cs!hyper!mark)
const unsigned short CRC::Table[] =
{
     //00
    0x0000,  0x1021,  0x2042,  0x3063,  0x4084,  0x50a5,  0x60c6,  0x70e7,
    0x8108,  0x9129,  0xa14a,  0xb16b,  0xc18c,  0xd1ad,  0xe1ce,  0xf1ef,

    //10
    0x1231,  0x0210,  0x3273,  0x2252,  0x52b5,  0x4294,  0x72f7,  0x62d6,
     0x9339,  0x8318,  0xb37b,  0xa35a,  0xd3bd,  0xc39c,  0xf3ff,  0xe3de,

     //20
    0x2462,  0x3443,  0x0420,  0x1401,  0x64e6,  0x74c7,  0x44a4,  0x5485,
    0xa56a,  0xb54b,  0x8528,  0x9509,  0xe5ee,  0xf5cf,  0xc5ac,  0xd58d,

     //30
    0x3653,  0x2672,  0x1611,  0x0630,  0x76d7,  0x66f6,  0x5695,  0x46b4,
    0xb75b,  0xa77a,  0x9719,  0x8738,  0xf7df,  0xe7fe,  0xd79d,  0xc7bc,

     //40
    0x48c4,  0x58e5,  0x6886,  0x78a7,  0x0840,  0x1861,  0x2802,  0x3823,
    0xc9cc,  0xd9ed,  0xe98e,  0xf9af,  0x8948,  0x9969,  0xa90a,  0xb92b,

     //50
    0x5af5,  0x4ad4,  0x7ab7,  0x6a96,  0x1a71,  0x0a50,  0x3a33,  0x2a12,
    0xdbfd,  0xcbdc,  0xfbbf,  0xeb9e,  0x9b79,  0x8b58,  0xbb3b,  0xab1a,

     //60
    0x6ca6,  0x7c87,  0x4ce4,  0x5cc5,  0x2c22,  0x3c03,  0x0c60,  0x1c41,
    0xedae,  0xfd8f,  0xcdec,  0xddcd,  0xad2a,  0xbd0b,  0x8d68,  0x9d49,

     //70
    0x7e97,  0x6eb6,  0x5ed5,  0x4ef4,  0x3e13,  0x2e32,  0x1e51,  0x0e70,
    0xff9f,  0xefbe,  0xdfdd,  0xcffc,  0xbf1b,  0xaf3a,  0x9f59,  0x8f78,

     //80
    0x9188,  0x81a9,  0xb1ca,  0xa1eb,  0xd10c,  0xc12d,  0xf14e,  0xe16f,
    0x1080,  0x00a1,  0x30c2,  0x20e3,  0x5004,  0x4025,  0x7046,  0x6067,

     //90
    0x83b9,  0x9398,  0xa3fb,  0xb3da,  0xc33d,  0xd31c,  0xe37f,  0xf35e,
    0x02b1,  0x1290,  0x22f3,  0x32d2,  0x4235,  0x5214,  0x6277,  0x7256,

     //A0
    0xb5ea,  0xa5cb,  0x95a8,  0x8589,  0xf56e,  0xe54f,  0xd52c,  0xc50d,
    0x34e2,  0x24c3,  0x14a0,  0x0481,  0x7466,  0x6447,  0x5424,  0x4405,

     //B0
    0xa7db,  0xb7fa,  0x8799,  0x97b8,  0xe75f,  0xf77e,  0xc71d,  0xd73c,
```

```
    0x26d3,  0x36f2,  0x0691,  0x16b0,  0x6657,  0x7676,  0x4615,  0x5634,

     //C0
    0xd94c,  0xc96d,  0xf90e,  0xe92f,  0x99c8,  0x89e9,  0xb98a,  0xa9ab,
    0x5844,  0x4865,  0x7806,  0x6827,  0x18c0,  0x08e1,  0x3882,  0x28a3,

     //D0
    0xcb7d,  0xdb5c,  0xeb3f,  0xfb1e,  0x8bf9,  0x9bd8,  0xabbb,  0xbb9a,
    0x4a75,  0x5a54,  0x6a37,  0x7a16,  0x0af1,  0x1ad0,  0x2ab3,  0x3a92,

     //E0
    0xfd2e,  0xed0f,  0xdd6c,  0xcd4d,  0xbdaa,  0xad8b,  0x9de8,  0x8dc9,
    0x7c26,  0x6c07,  0x5c64,  0x4c45,  0x3ca2,  0x2c83,  0x1ce0,  0x0cc1,

     //F0
    0xef1f,  0xff3e,  0xcf5d,  0xdf7c,  0xaf9b,  0xbfba,  0x8fd9,  0x9ff8,
    0x6e17,  0x7e36,  0x4e55,  0x5e74,  0x2e93,  0x3eb2,  0x0ed1,  0x1ef0
};


CRC::CRC( unsigned short Value ) :
     Val( Value )
{}


CRC::~CRC()
{}


void CRC::Reset()
{
     Val = 0;
}


unsigned short CRC::operator() ( const char c )
{
     return operator() ( static_cast<const unsigned char>( c ) );
}


unsigned short CRC::operator() ( const unsigned char c )
{
     unsigned short interimChar = c;

     return Val = ( Val << 8 ) ^ Table[ ( Val >> 8 ) ^ interimChar ];;
}


unsigned short CRC::operator() ( const char *cp, const char *End )
{
     return operator() ( (const unsigned char *)( cp ), (const unsigned char *)( End ) );
};


unsigned short CRC::operator() ( const unsigned char *cp, const unsigned char *End )
{
     while( cp < End )
     {
          Val = ( Val << 8 ) ^ Table[ ( Val >> 8 ) ^ static_cast<unsigned short>( *cp++ ) ];
     }

     return Val;
```

```
}

CRC::operator unsigned short &() { return Val; };

#pragma export reset
```

```
// RcvBuffer.h
//
// Interface for a receive buffer class
// when streams are decoded, completely decoded packets are stored in objects of this type
// these objects may contain multiple TB or BJ packets and may need further decoding to
// separate into parts
// this class does not attempt to decode the contents of the buffer
// this class is designed to be thread-safe and supports queue-based containers


// only include this file once per compilation
#ifndef        _RCVBUFFER_H_
#define        _RCVBUFFER_H_


#include  "FlagByte.h"


class     RcvBuffer
{
     public:
          // the buffer carries a notion of the status of the buffer
          // in terms of decoding a packet
          // really bad packets (i.e. cannot be decoded) are dumped
          enum
          {
               // set during decoding while data is added to this buffer
               NoPacketYet              = 0x0000,

               // oddities, not necessarily errors
               BigHeader           = 0x0001,           // header's too big
               LargeData           = 0x0002,           // this is a large packet
               SmallData           = 0x0004,           // this is a small packet
               CRCDidNotCheck      = 0x0008,           // the CRC was bad
               OddityBits               = 0x000F,           // mask for these values

               // errors from the packet processor
               ShortPacket              = 0x0010,           // packet was short
               UnknownEscape       = 0x0020,           // unknown escape sequence
               BadCRCBits               = 0x0040,           // the CRC type is invalid
               ConfusedMachine          = 0x0080,           // the state machine is confused
               ErrorBits           = 0x00F0,           // mask for these values

               // aggregates of the above values
               GoodPacketRcvd      = 0x1000,           // Good CRC no decoding errors
               PacketError              = 0x8000            // An error occured
          };

     private:
          // this is the actual buffer of the class
          // the size of the buffer is NOT dynamic after construction
          // this is designed so that there is a one-to-one relationship between the  packets
          // and objects of this class
          unsigned char* Buffer;             // pointer to the buffer
          unsigned char* BufferIndex;   // a moving pointer to the buffer insides, used internally
          unsigned short DataLen;       // the length of the buffer from the  DataLen header field
          unsigned short CurrentLen;         // the "filled" portion of the buffer, used for error checking

          // status Byte
          unsigned short Status;             // carries the packet status as described above

          // Information about the received buffer
          // Flag Byte
```

```
    FlagByte       InfoByte;      // carries the packet flag byte values

    // Sequence Number - as transmitted in the header
    // use for re-assembly
    unsigned short SequenceNumber;

    // address bytes - as transmitted in the header
    unsigned short DestAddress;
    unsigned short SourceAddress;

public:
    // a default constructor, not really used for anything except
    // making buffers that will later be filled by an assignment
    // this makes a one-byte buffer
    // status is set to PacketError to indicate it is not initialized
    RcvBuffer();

    // constructs a buffer object of length "theDataLen" and allocates the
    // necessary memory for that object
    // status is set to PacketError to indicate it is not initialized
    RcvBuffer(     unsigned short theDataLen );

    // the copy constructor for this class that actually allocates
    // new memory and copies the contents of the buffers rather
    // than the buffer pointers
    // length, status, etc. are all copied as well
    RcvBuffer( const RcvBuffer&   theRcvBuffer );

    // deletes any memory that's allocated prior to destruction
    ~RcvBuffer();

    // this method returns the set length of the data buffer
    unsigned short GetDataLen() const;

    // this method returns the actual written length of the buffer
    unsigned short GetActualLen() const;

    // this method sets the status of the packet to one of the three possible
    // values as described above
    void           SetStatus( unsigned short theStatus );

    // this method returns the status of the packet
    unsigned short GetStatus() const;

    // status checkers
    inline    bool HasOddities()  { return ( Status & OddityBits  );      }
    inline    bool HasError()          { return ( Status & PacketError );      }
    inline    bool IsOK()              { return ( Status & GoodPacketRcvd );   }

    // these methods access the information transmitted in the header of the packet
    // this method sets the InfoByte of the packet
    void           SetInfoByte(       FlagByte       theInfoByte             );
    void           SetSequenceNumber(  unsigned short theSequenceNumber   );
    void           SetDestAddress(          unsigned short theDestAddress      );
    void           SetSourceAddress(   unsigned short theSourceAddress    );

    // this method returns the status of the packet
    FlagByte       GetInfoByte() const;
    unsigned short GetSequenceNumber() const;
    unsigned short GetDestAddress() const;
    unsigned short GetSourceAddress() const;

    // InfoByte checkers
```

```
        inline bool         NeedsResponse()     { return ( InfoByte.isReqACK() );  }

        // this method appends a byte to the data buffer
        // and checks to make sure that the buffer doesn't overflow
        // returns the number of bytes written to the buffer
        unsigned short AddByte( unsigned char theByteToAdd );  // adds a byte to the end of the buffer

        // this is the assignment operator that actually copies the buffer rather
        // than just copying the the pointer
        RcvBuffer&          operator=( const RcvBuffer& rhs );

        // this operator extracts a single, indexed byte from the buffer
        // used to support output operators (i.e. <<)
        unsigned char       operator[]( int i ) const;

        // this is the unsigned char* conversion operator that
        // returns a pointer to the actual data buffer
                            operator unsigned char*();

        // this is the less than operator used to support containers
        bool                operator<( const RcvBuffer& rhs ) const;
};

#endif    _RCVBUFFER_H_
```

```
// RcvBuffer.cpp
//
// Implementation of a receive buffer class


#include  <cstring>

#include  "RcvBuffer.h"


using namespace std;


#pragma export on
// constructor/destructor
RcvBuffer::RcvBuffer() :
     DataLen( 1 ),
     CurrentLen( 0 )
{
     Buffer = new unsigned char[ 1 ];
     BufferIndex = Buffer;
     Status = PacketError;
}


RcvBuffer::RcvBuffer( unsigned short theDataLen ) :
     DataLen( theDataLen ),
     CurrentLen( 0 )
{
     if( DataLen < 1 )
     {
          DataLen = 1;
     }

     Buffer = new unsigned char[ DataLen ];
     BufferIndex = Buffer;
     Status = PacketError;
}


RcvBuffer::RcvBuffer( const RcvBuffer& theRcvBuffer ) :
     DataLen(       theRcvBuffer.DataLen          ),
     Status(             theRcvBuffer.Status           ),
     CurrentLen(         theRcvBuffer.CurrentLen       )
{
     Buffer = new unsigned char[ theRcvBuffer.DataLen ];
     memcpy( Buffer, theRcvBuffer.Buffer, theRcvBuffer.DataLen );

     // set the buffer index
     BufferIndex = Buffer + CurrentLen;
}


RcvBuffer::~RcvBuffer()
{
     if( Buffer )
     {
          delete [] Buffer;
     }
}


// buffer information & maintenance
```

```
unsigned short RcvBuffer::GetDataLen() const
{
     return DataLen;
}


unsigned short RcvBuffer::GetActualLen() const
{
     return CurrentLen;
}


void RcvBuffer::SetStatus( unsigned short theStatus )
{
     Status = theStatus;
}


unsigned short RcvBuffer::GetStatus() const
{
     return Status;
}


void           RcvBuffer::SetInfoByte( FlagByte theInfoByte )
{
     InfoByte = theInfoByte;
}


void           RcvBuffer::SetSequenceNumber( unsigned short theSequenceNumber )
{
     SequenceNumber = theSequenceNumber;
}


void           RcvBuffer::SetDestAddress( unsigned short theDestAddress )
{
     DestAddress = theDestAddress;
}


void           RcvBuffer::SetSourceAddress( unsigned short theSourceAddress )
{
     SourceAddress = theSourceAddress;
}


FlagByte       RcvBuffer::GetInfoByte() const
{
     return InfoByte;
}


unsigned short RcvBuffer::GetSequenceNumber() const
{
     return SequenceNumber;
}


unsigned short RcvBuffer::GetDestAddress() const
{
     return DestAddress;
```

```
}


unsigned short RcvBuffer::GetSourceAddress() const
{
     return SourceAddress;
}


// adds a byte to the buffer
unsigned short RcvBuffer::AddByte( unsigned char theByteToAdd )
{
     // this prevents us from overflowing our buffer
     // as long as the current count is less than or
     // equal to the total buffer size (DataLen)
     if( ++CurrentLen <= DataLen )
     {
          *BufferIndex = theByteToAdd;
          ++BufferIndex;
     }

     return CurrentLen;
}


// operators
// assignment operator
RcvBuffer&     RcvBuffer::operator=( const RcvBuffer& rhs )
{
     // simple members
     DataLen        = rhs.DataLen;
     CurrentLen     = rhs.CurrentLen;
     Status         = rhs.Status;

     // delete any previous buffer
     if( Buffer )
     {
          delete [] Buffer;
     }

     // copy the contents of the right-hand side buffer
     Buffer = new unsigned char[ DataLen ];
     memcpy( Buffer, rhs.Buffer, DataLen );

     // set the buffer index
     BufferIndex = Buffer + CurrentLen;

     return *this;
}


// extracts a byte from the buffer
unsigned char  RcvBuffer::operator[]( int i ) const
{
     return Buffer[ i ];
}


// returns a pointer to the buffer
RcvBuffer::operator unsigned char*()
{
     return Buffer;
}
```

```
#pragma export reset
```

```
// RcvPacket.h
//
// Interface for the receive packet class
//
// this class decodes link protocol using a state machine
// decoded packets are placed into RcvBuffer's and onto
// a queue where they can be extracted at a later time
// the RcvBuffer contains only the data and is de-escaped
// and the CRC is checked, if present
// this decoder works on a stream of data and will find
// the packet boundaries itself
// errors (dropped bytes, etc.) will cause the state
// machine to reset and begin looking for another packet
// usually, two packets are lost in this process, the erroneous packet
// and the following packet


// only include this file once per compilation
#ifndef        _RCVPACKET_H_
#define        _RCVPACKET_H_


#include  <string>
#include  <deque>
#include  <queue>

#include  "FlagByte.h"
#include  "Header.h"
#include  "CRC.h"
#include  "RcvBuffer.h"


using namespace std;


class RcvPacket
{
     private:
          // these are the internal states of the decoder state machine
          typedef enum
          {
               WaitForOpen,        // waiting for an open flag, 0x02
               ReadType,           // read the TypeFlag (see FlagByte.h)
               ReadHeaderLen,      // read the header length (see Header.h)
               ReadDataLen,        // read the two data length bytes
               ReadSeqNum,              // read the two sequence number bytes (if present)
               ReadDest,           // read the two destination bytes (if present)
               ReadSrc,            // read the two source bytes (if present)
               CollectingData,          // read the data of the packet and add to a RcvBuffer (see RcvBuffer.h)
               ReadCRC,            // read the CRC bytes (if present)
               GotDLE                   // got an escape byte (0x10), translate the next byte 0x45->0x10,, 0x4F->0x02
          }
          PacketState;

          // enums for error/oddity detection
          enum
          {
               LargeHeaderLimit    = 10,          // headers are not normally more than 10 bytes
               LargeDataLimit      = 2048,        // fairly arbitrary, data shouldn't larger than this
               SmallDataLimit      = 0            // there shouldn't be any empty packets
          };

     private:
```

```
        // state machine members
        PacketState         CurrentState;       // the current state of the state machine
        PacketState         LastState;              // the state to return to after GotDLE state
        unsigned char  HeaderCount;        // the number of header bytes to read, supports any number to 255
        unsigned short DataCount;               // the number of data bytes to read (to 65535)
        unsigned long  ByteCount;               // a reused temporary byte count used while reading
        unsigned long  theRunningCRC;      // the calculated CRC during decoding
        CRC                 theCRC;                 // the CRC object used to calculate the CRC

        // the running status used to tag each completed packet
        // each packet is tagged with a status that indicates how well it was decoded
        // i.e. the CRC didn't check or the data section was large, etc...
        unsigned short RunningResult;

        // packet data - basically the header (see Header.h for a description of these bytes)
        FlagByte       PacketType;                  // required
        unsigned char  HeaderLen;                   // required
        unsigned short DataLen;                // required
        unsigned short SeqNum;                      // optional
        unsigned short Dest;                        // optional
        unsigned short Src;                    // optional
        unsigned short theRcvdCRC;                  // optional, depending on PacketType

        // as packets are decoded, the resulting bytes are stored in an object of type RcvBuffer
        // this pointer points to the buffer supporting the currently decoded packet
        // as packets are completed, they are placed on the PacketQueue (see next member)
        // new RcvBuffers are allocated after the DataLen is read out of the next packet's
        // header (see Header.h)
        RcvBuffer*              currentBuffer;

        // this is a queue of RcvBuffers where completed buffers
        // are stored until they can be retrieved
        priority_queue< RcvBuffer >   PacketQueue;

    public:
        // the default constructor sets up the state to WaitForOpen
        RcvPacket();

        // this operator decodes the portion of the stream delineated by
        // the start and end pointers and returns the number of complete
        // packets on the queue, including old ones not yet retrieved
        int  operator() ( unsigned char* start, unsigned char* end );

        // this RcvBuffer conversion operator returns the oldest completed
        // packet on the queue
                  operator RcvBuffer();

    private:
        // this method is the actual state machine that decodes a byte at a time
        unsigned short ProcessByte( unsigned char theByte );
};

#endif    _RCVPACKET_H_
```

```
// RcvPacket.cpp
//
// Implementation of the receive packet base class


#include  "CRC.h"
#include  "RcvPacket.h"
#include  "RcvBuffer.h"


#pragma export on
// default constructor
RcvPacket::RcvPacket() :
     CurrentState( WaitForOpen ),                    // set the initial state machine to look for OpenFlags
     currentBuffer( 0L ),                                 // zero the initial buffer pointer
     RunningResult( RcvBuffer::NoPacketYet )      // initial result is no result
{}


// poll for complete packets, bytes are buffered by the port via interrupts
// this routine will put your thread to sleep until data is received
int       RcvPacket::operator() ( unsigned char* start, unsigned char* end )
{
     // keep track of the current byte to process
     unsigned char* currByte = start;

     // process all the bytes in the buffer
     while( currByte <= end )
     {
          // process a byte and see if a complete packet is formed
          RunningResult |= ProcessByte( *( currByte++ ) );

          // if the running result contains errors or indicates the end of a packet
          if( RunningResult & ( RcvBuffer::ErrorBits | RcvBuffer::GoodPacketRcvd ) )
          {
               // check for error bits
               if( RunningResult & RcvBuffer::ErrorBits )
               {
                    // if some error bit was set after processing this byte
                    // then set the overall status indicator to PacketError
                    RunningResult |= RcvBuffer::PacketError;
               }

               // create a buffer if necessary since we may have
               // a bad packet before buffer creation
               if( currentBuffer == 0L )
               {
                    // allocate a new empty buffer
                    currentBuffer = new RcvBuffer;
               }

               // set the status of the packet buffer
               currentBuffer -> SetStatus( RunningResult );

               // set the infobyte of the packet buffer
               currentBuffer -> SetInfoByte( PacketType );

               // put the buffer on the queue
               PacketQueue.push( *currentBuffer );

               // reset the packet status running result
               RunningResult = RcvBuffer::NoPacketYet;
```

```
                // remove the buffer and set the pointer to NULL
                delete currentBuffer;
                currentBuffer = 0L;

                // a new buffer object will be allocated by ProcessByte
                // after it knows how big the incoming next packet is
            }
        }

        // return the number of good packets
        return PacketQueue.size();
}


// method to extract buffers from the object's
// buffer queue, returns an actual buffer
RcvPacket::operator RcvBuffer()
{
        // if there are buffers in the queue
        if( !PacketQueue.empty() )
        {
            // construct a new buffer on the stack
            // with the contents of the next Queue element
            // and remove the element from th Queue
            RcvBuffer retVal( PacketQueue.top() );

            // remove the element from the Queue
            PacketQueue.pop();

            // pass it back to the caller
            return retVal;
        }
        else
        {
            // the Queue is empty, so make a NULL buffer
            // to return to the caller
            RcvBuffer retVal( 0L );

            // pass it back to the caller
            return retVal;
        }
}


// process buffer bytes one at a time
// this method operates as a state machine - lots of code but fast execution
unsigned short RcvPacket::ProcessByte( unsigned char theByte )
{
        // each time through the state machine, return some status
        // NoPacketYet is the default, no-error status
        unsigned short retVal = RcvBuffer::NoPacketYet;

        // before we process the byte, check for an out of place open flag
        // in other words, we got an OpenFlag before we expected it
        // so the packet is short and that error is set
        if( ( theByte == Header::OpenFlag ) && ( CurrentState != WaitForOpen ) )
        {
            // we've got a short packet let's return that error
            retVal = RcvBuffer::ShortPacket;

            // set the state to WaitForOpen Flag to start the new packet
            // this will cause the byte to be processed below in this
            // pass of the state machine
```

```
     CurrentState = WaitForOpen;
}

// ignore escapes if we're waiting for the OpenFlag
// if the character is an escape, DLE = 0x10
if( ( CurrentState != WaitForOpen ) && ( theByte == Header::DLE ) )
{
     // save off the current state of the machine
     LastState      = CurrentState;

     // take a detour to process the escaped byte
     // which will be the next byte processed
     CurrentState   = GotDLE;
}
else
{
     // we're de-escaping a character if the state is GotDLE
     // this is done prior to normal processing of the byte
     // in other words, we convert the escaped value and then
     // process it in the same pass
     if( CurrentState == GotDLE )
     {
          // after this byte, return to the last state
          // so that the state machine's switch will
          // process the converted byte
          CurrentState = LastState;

          // let's de-escape the data
          // right now we only know two values,
          // all others are errors
          switch( theByte )
          {
               case Header::OpenESC:
                    theByte = Header::OpenFlag;
                    break;

               case Header::ESCESC:
                    theByte = Header::DLE;
                    break;

               default:
                    // we've got a bad escape sequence, let's return that error
                    retVal = RcvBuffer::UnknownEscape;

                    // set the state to WaitForOpen Flag to start a new packet
                    // this will effectively dump the rest of the corrupted
                    // packet to the bit bucket
                    CurrentState = WaitForOpen;
                    break;
          }
     }

     // process the characters based on the current state
     switch( CurrentState )
     {
          // look for the OpenFlag
          // this dumps bytes until the OpenFlag is received
          case WaitForOpen:
               if( theByte == Header::OpenFlag )
               {
                    // reset for a new packet
                    // note that the OpenFlag does not count in the header size,
                    // data size, nor is it stored anywhere
```

```
          HeaderCount = 0;                                  // initialize the header byte count
          DataCount = 0;                                 // initialize the data byte count
          CurrentState = ReadType;                       // set the state for the next byte
          theCRC.Reset();                                        // initialize the CRC calculator
     }
     break;

// get the required type flag
case ReadType:
     ++HeaderCount;                                         // this is a header byte
     PacketType.SetFlag( theByte );                         // store the type flag

     if( PacketType.ValidCRCBits() )                        // verify that the CRC bits in the
     {                                                           // the flag are valid
          theRunningCRC = theCRC( theByte );           // add this byte to the CRC
          CurrentState = ReadHeaderLen;                // now look for the header length
     }
     else
     {                                                           // the CRC bits are not valid
          retVal = RcvBuffer::BadCRCBits;                 // set the return type for an error
          CurrentState = WaitForOpen;                     // state to WaitForOpen to start the new packet
     }
     break;

// get the length of the header (without escapes) - it is variable up to 256
case ReadHeaderLen:
     ++HeaderCount;                                         // this is a header byte
     HeaderLen = theByte;                                   // store the header length
     CurrentState = ReadDataLen;                            // now look for the data length
     ByteCount = 0;                                         // reset the byte count
     theRunningCRC = theCRC( theByte );                // add this byte to the CRC

     if( HeaderLen > LargeHeaderLimit )                // nominal maximum header length
     {
          retVal = RcvBuffer::BigHeader;               // return this as an oddity
     }                                                           // but keep processing the packet
     break;                                                      // i.e. don't change the state to WaitForOpen

// get the length of the data (without escapes) - it is variable up to 65535
case ReadDataLen:
     *( (char*)&DataLen + ByteCount++ ) = theByte;     // store the byte
     ++HeaderCount;                                         // this is a header byte
     theRunningCRC = theCRC( theByte );                // add this byte to the CRC

     if( ByteCount >= sizeof( DataLen ) )              // check to see if we got it all
     {
          if( currentBuffer )                               // allocate a new buffer for the data
          {
               delete currentBuffer;                        // toss any old buffer, just in case
          }

          currentBuffer = new RcvBuffer( DataLen );    // the new buffer has the received length

          if( HeaderCount >= HeaderLen )                    // if at the end of the header
          {
               CurrentState = CollectingData;               // go collect data
          }
          else
          {
               ByteCount = 0;                               // reset the byte count
               CurrentState = ReadSeqNum;                   // now look for sequence number
          }
```

```
          if( DataLen >= LargeDataLimit )                   // check for large packets
          {
               retVal = RcvBuffer::LargeData;               // return this as an oddity
          }                                                      // but keep processing the packet

          if( DataLen <= SmallDataLimit )                   // check for empty packets
          {
               retVal = RcvBuffer::SmallData;               // return this as an oddity
          }                                                      // but keep processing the packet
     }
     break;

// get the optional sequence number
case ReadSeqNum:
     *( (char*)&SeqNum + ByteCount++ ) = theByte; // store the byte
     ++HeaderCount;                                         // this is a header byte
     theRunningCRC = theCRC( theByte );                // add this byte to the CRC

     if( ByteCount >= sizeof( SeqNum ) )                    // check to see if we got it all
     {
          currentBuffer -> SetSequenceNumber( SeqNum );     // set the return packet sequence number

          if( HeaderCount >= HeaderLen )                    // if at the end of the header
          {
               CurrentState = CollectingData;               // go collect data
          }
          else
          {
               ByteCount = 0;                               // reset the byte count
               CurrentState = ReadDest;                // now look for destination number
          }
     }
     break;

// get the optional destination
case ReadDest:
     *( (char*)&Dest + ByteCount++ ) = theByte;        // store the byte
     ++HeaderCount;                                         // this is a header byte
     theRunningCRC = theCRC( theByte );                // add this byte to the CRC

     if( ByteCount >= sizeof( Dest ) )                 // check to see if we got it all
     {
          currentBuffer -> SetDestAddress( Dest );     // set the return packet destination address

          if( HeaderCount >= HeaderLen )                    // if at the end of the header
          {
               CurrentState = CollectingData;               // go collect data
          }
          else
          {
               ByteCount = 0;                               // reset the byte count
               CurrentState = ReadSrc;                      // now look for destination number
          }
     }
     break;

// get the optional source
case ReadSrc:
     *( (char*)&Src + ByteCount++ ) = theByte;         // store the byte
     ++HeaderCount;                                         // this is a header byte
     theRunningCRC = theCRC( theByte );                // add this byte to the CRC

     if( ByteCount >= sizeof( Src ) )                  // check to see if we got it all
```

```
     {
          currentBuffer -> SetSourceAddress( Src );    // set the return packet source address

          ByteCount = 0;                                    // reset the byte count
          CurrentState = CollectingData;                    // now look for destination number
     }
     break;

case CollectingData:
     theRunningCRC = theCRC( theByte );                // add this byte to the CRC

     // if we're at the end of the header do something with the characters
     if( HeaderCount >= HeaderLen )
     {
          // this is where the data inside the link protocol is stored
          // once we've stored the appropriate number of bytes
          // check for the CRC bytes
          if( ( currentBuffer -> AddByte( theByte ) ) >= DataLen )
          {
               // determine if we need to check the CRC
               if( PacketType.hasCRC() )
               {
                    ByteCount =  0;                              // yeah, we'll read the CRC
                    CurrentState = ReadCRC;                 // set that state
               }
               else
               {
                    retVal = RcvBuffer::GoodPacketRcvd;          // we've completed this packet
                    CurrentState = WaitForOpen;                       // start looking for the next OpenFlag
               }
          }
     }
     else
     {
          // ignore any data after the header if the header is longer than we think
          ++HeaderCount;
     }
     break;

// get the optional CRC
case ReadCRC:
     *( (char*)&theRcvdCRC + ByteCount++ ) = theByte;       // store the byte

     // check to see if we got it all
     if( ByteCount >= sizeof( theRcvdCRC ) )
     {
          // Check the CRC
          if( theRcvdCRC == theRunningCRC )
          {
               retVal = RcvBuffer::GoodPacketRcvd;               // the CRC matches set it good
          }
          else
          {
               retVal = RcvBuffer::CRCDidNotCheck;               // the CRC did not match
          }

          // regardless of the outcome of the CRC
          // start looking for the next packet
          CurrentState = WaitForOpen;
     }
     break;

default:
```

```
                    // we're confused abort and
                    // start looking for the next packet
                    retVal = RcvBuffer::ConfusedMachine;
                    CurrentState = WaitForOpen;
                    break;
               }
          }

          return retVal;
}


bool RcvBuffer::operator<( const RcvBuffer& rhs ) const
{
     return InfoByte.Prio() < rhs.InfoByte.Prio();
}


#pragma export reset
```

```
// SendPacket.h
//
// Interface for the send packet class
// this class formats buffers, adds the link protocol elements (header & CRC)
// and provides support methods to output the protocol packets to classes
// that take buffers as inputs
// the input interface is the constructor, the primary output interface is
// an unsigned char* conversion operator
// this class owns its data buffer and performs a copy on creation that allows
// the incoming data to go invalid after construction
// many features are to support queuing of objects of this type


// only include this file once per compilation
#ifndef        _SENDPACKET_H_
#define        _SENDPACKET_H_


#include  "Header.h"
#include  "CRC.h"

// if compiled under the NewReceiver (Blackjack or BJ)
// then include the receiver serial port definitions
#ifdef  _NR_
  #include "SerialPort.h"
#endif  _NR_


class SendPacket
{
     private:
          // this is the largest size of a  protocol packet
          // this size is due to the two byte DataLength field of the header (16-bits)
          enum
          {
               MaxOutPacketSize = 65535
          };

     private:
          // the  link protocol header to transmit, see Header.h
          Header              packetHeader;

          // this buffer holds the incoming data to be converted to  link protocol
          // it is allocated and deallocated as necessary
          // the buffer consists of two parts, a pointer to the allocated data
          // and the size of that allocation as determined by the constructor arguments
          unsigned char* dataBuffer;
          short int      dataBufferSize;

          // this buffer holds the formatted output data after the link protocol has
          // been applied to the dataBuffer
          // this buffer is often larger than the data buffer due to header overhead
          // and escape sequences to remove 0x02, and 0x10 from the data stream
          // the output buffer is reallocated and reformatted each time the packet
          // is used for output rather than when it is used for input
          // this allows for the construction of this class to happen quickly while
          // outputs can take more time in a different execution context if necessary
          // (supports queuing)
          unsigned char* outBuffer;
          short int      outBufferSize;

          // this is the CRC of the packet, it is calculated at output time as the
          // output buffer is formatted, it is only appended to the packet if the
```

```
    // typeflag bits in the header indicated that the CRC is present (see FlagByte.h)
    CRC                 packetCRC;

    // a rolling sequence number used to provide sequencing if necessary (see Header.h)
    // this static remains in the library for all instances of this class to share so that the
    // sequence numbers are monotonic and so that they have no holes
    // the sequence numbers will roll over from 0xFFFF to 0x0000 automatically
    static short SequenceNum;

public:
    // constructors / destructor
    // the default constructor does nothing except zero
    // the data buffer and output buffer pointers
    SendPacket();

    // this copy constructor copies the packet's buffers byte for byte so
    // that these objects can be copied and then destroyed (supports queuing)
    SendPacket( const SendPacket& theSendPacket );

    // this is the primary constructor that is used to create a  link protocol packet
    // the contents pointed to by bufferToSend are copied into the data buffer (bufferSize bytes worth)
    // the priority of the packet is supported so that priority queues can be used in the future
    // to send packets out of band, the default setting is zero (0-3 possible settings) which is lowest
    SendPacket( const unsigned char* bufferToSend, const int bufferSize, const int Priority = 0 );

    // yeah! a destructor that actually does something, this destructor deletes the buffers
    // and de-allocates the memory before destruction
    ~SendPacket();

    // this method allows you to add more data to the packet (up to 65535 total bytes)
    // and returns the new size of the packet
    // if the new data would overflow the packet, the data is ignored and the returned
    // size does not change
    long AppendPacket( const unsigned char* theBuffer, const int bufferSize );

    // this method zeros the data buffers and de-allocates any memory associated with them
    void ZeroPacket();

    // this method returns the size of the OUTPUT buffer, which is different than the
    // data buffer and includes everything in the buffer including the open flag, header,
    // and CRC (if present) as well as all the escape bytes
    // this method is useful for calculating the transmit time of the packet
    int       GetOutBufferSize();

    // this operator formats the data buffer by escaping necessary bytes, adds the
    // open flag, header and CRC, if necessary, and places those bytes in the output buffer
    // it returns a pointer to the output buffer
    operator unsigned char*();

    // this is the assignment operator that actually copies the data buffers rather
    // than just copying the the pointers
    SendPacket&         operator=( const SendPacket& rhs );

    // this operator provides support for queue-based containers that need to
    // sort packets using less than
    // packets are sorted based on their priority
    bool                operator<( const SendPacket& rhs ) const;

public:
    // access commands to manipulate the packet's header
    void SetTypeFlag( unsigned char theTypeFlag );                      // set the raw Flag Value
    void SetTypeToReqACK();                                                 // set to request an acknowledge
    void SetTypeToTransport();                                              // set to normal transport packet
```

```
          void SetTypeToNAQ();                                                    // set as a Negative Acknowledgement
          void SetTypeToACK();                                                    // set as a Positive Acknowledgement
          void SetPriority( int thePriority );                                  // set the priority
          void SetToUseSequenceNumber();                                        // set to use the sequence number
          void SetDestinationAddress(   unsigned short theDestination );   // set to use a destination address
          void SetSourceAddress(        unsigned short theSource );            // set to use a source address

     private:
          // this is the method that actually takes the data buffer and reformats it
          // according to the  protocol (see operator unsigned char*() description)
          void PrepareBuffer();

          // this method escapes bytes
          // it takes a buffer pointer and writes the escaped bytes
          // according to this rule:
          //     STX (Open Flag) -> DLE, "O"
          //     DLE (Escape)    -> DLE, "E"
          //     others          -> unmodified
          // it returns the number of bytes written to the buffer ( 1 or 2 )
          int       EscapeAndWrite( unsigned char theByte, unsigned char* theBuffer );

// if compiled under the NewReceiver (Blackjack or BJ)
// then allow the overloaded operator<< to access our data members for output support
// to receiver serial ports
// this operator, implemented as a part of this library, takes a SendPacket and
// outputs it as a buffer to the serial port which locks the port during this packet
// preventing the interleaving of packets and making this class thread-safe
#ifdef  _NR_
     friend    SerialPort& operator<< ( SerialPort& s, SendPacket& thePacket );
#endif  _NR_
};

#endif    _SENDPACKET_H_
```

```
// SendPacket.cpp
//
// Implementation of the  send packet base class


#include  <cstring>

#include  "FlagByte.h"

#include  "SendPacket.h"


using namespace std;


#ifdef  _NR_
#include "SerialPort.h"
#endif  _NR_


#pragma export on
short SendPacket::SequenceNum = 0;


// constructors/destructor
SendPacket::SendPacket() :
     dataBuffer( 0 ),
     outBuffer( 0 )
{
     // bump the sequence number and store it in the header
     packetHeader.SetSequenceNumber( ++SequenceNum );
}


SendPacket::SendPacket( const SendPacket&    theSendPacket ) :
     packetHeader( theSendPacket.packetHeader ),
     dataBuffer( 0 ),
     outBuffer( 0 ),
     packetCRC( theSendPacket.packetCRC )
{
     // allocate the data buffer and copy the data into it
     dataBuffer = new unsigned char[ theSendPacket.dataBufferSize ];
     memcpy( dataBuffer, theSendPacket.dataBuffer, theSendPacket.dataBufferSize );
     dataBufferSize = theSendPacket.dataBufferSize;

     // allocate the output buffer and copy the data into it
     outBuffer = new unsigned char[ theSendPacket.outBufferSize ];
     memcpy( outBuffer, theSendPacket.outBuffer, theSendPacket.outBufferSize );
     outBufferSize = theSendPacket.outBufferSize;
}


SendPacket::SendPacket( const unsigned char* bufferToSend, const int bufferSize, const int Priority ) :
     packetHeader( bufferSize ),
     outBuffer( 0 )
{
     // bump the sequence number and store it in the header
     packetHeader.SetSequenceNumber( ++SequenceNum );

     // allocate the buffer and copy the data into it
     dataBuffer = new unsigned char[ bufferSize ];
     memcpy( dataBuffer, bufferToSend, bufferSize );
     dataBufferSize = bufferSize;
```

```
     FlagByte  theFlag( packetHeader.GetTypeFlag() );
     theFlag.SetPrio( Priority );
     packetHeader.SetTypeFlag( theFlag );
}


SendPacket::~SendPacket()
{
     if( dataBuffer )
     {
          delete [] dataBuffer;
     }

     if( outBuffer )
     {
          delete [] outBuffer;
     }
}


long SendPacket::AppendPacket( const unsigned char* theBuffer, const int bufferSize )
{
     if( ( dataBufferSize + bufferSize ) < SendPacket::MaxOutPacketSize )
     {
          // hold the buffer with a temporary pointer
          unsigned char* oldBuffer = dataBuffer;

          // allocate a new and larger buffer
          dataBuffer = new unsigned char[ bufferSize + dataBufferSize ];

          // copy the previous buffer to the new buffer
          memcpy( dataBuffer, oldBuffer, dataBufferSize );

          // append the new buffer to the old data in the new buffer
          memcpy( dataBuffer + dataBufferSize, theBuffer, bufferSize );

          // adjust the buffer size
          dataBufferSize += bufferSize;

          // remove the old data
          delete [] oldBuffer;

          // adjust the data size in the header
          packetHeader.SetDataLength( dataBufferSize );
     }

     // return the new size
     return dataBufferSize;
}


void SendPacket::ZeroPacket()
{
     if( dataBuffer )
     {
          delete [] dataBuffer;
     }
     dataBuffer = 0;
     dataBufferSize = 0;

     if( outBuffer )
     {
```

```
          delete [] outBuffer;
     }
     outBuffer = 0;
     outBufferSize = 0;
}


int  SendPacket::GetOutBufferSize()
{
     PrepareBuffer();
     return outBufferSize;
}


SendPacket::operator unsigned char*()
{
     PrepareBuffer();
     return outBuffer;
}


SendPacket&    SendPacket::operator=( const SendPacket& rhs )
{
     // take care of the header
     packetHeader = rhs.packetHeader;

     // take care of the buffers (data & out)
     // allocate the data buffer and copy the data into it
     if( dataBuffer )
     {
          delete [] dataBuffer;
     }
     dataBuffer = new unsigned char[ rhs.dataBufferSize ];
     memcpy( dataBuffer, rhs.dataBuffer, rhs.dataBufferSize );
     dataBufferSize = rhs.dataBufferSize;

     // allocate the output buffer and copy the data into it
     if( outBuffer )
     {
          delete [] outBuffer;
     }
     outBuffer = new unsigned char[ rhs.outBufferSize ];
     memcpy( outBuffer, rhs.outBuffer, rhs.outBufferSize );
     outBufferSize = rhs.outBufferSize;

     // take care of the CRC
     packetCRC = rhs.packetCRC;

     return *this;
}


bool SendPacket::operator<( const SendPacket& rhs ) const
{
     FlagByte  thisFlagByte( packetHeader.GetTypeFlag() );
     FlagByte  thatFlagByte( rhs.packetHeader.GetTypeFlag() );

     return thisFlagByte.Prio() < thatFlagByte.Prio();
}


void SendPacket::SetTypeFlag( unsigned char theTypeFlag )
{
```

```
     packetHeader.SetTypeFlag( theTypeFlag );
}


void SendPacket::SetTypeToReqACK()
{
     // make a copy of the flag byte
     FlagByte  theFlagByte( packetHeader.GetTypeFlag() );

     // set to request the acknowledge
     theFlagByte.SetReqACK( true );

     packetHeader.SetTypeFlag( theFlagByte );
}


void SendPacket::SetTypeToTransport()
{
     // make a copy of the flag byte
     FlagByte  theFlagByte( packetHeader.GetTypeFlag() );

     // set as a Transport type
     theFlagByte.SetToTransportPkt();

     packetHeader.SetTypeFlag( theFlagByte );
}


void SendPacket::SetTypeToNAQ()
{
     // make a copy of the flag byte
     FlagByte  theFlagByte( packetHeader.GetTypeFlag() );

     theFlagByte.SetToNAQ();            // set as a negative acknowledge type
     theFlagByte.SetCRC( true );        // NAQ packets always have CRC
     theFlagByte.SetPrio( 3 );          // NAQ packets always have priority 3 (highest)

     packetHeader.SetTypeFlag( theFlagByte );
}


void SendPacket::SetTypeToACK()
{
     // make a copy of the flag byte
     FlagByte  theFlagByte( packetHeader.GetTypeFlag() );

     // set as a positive acknowledge type
     theFlagByte.SetToACK();            // set as a positive acknowledge type
     theFlagByte.SetCRC( true );        // ACK packets always have CRC
     theFlagByte.SetPrio( 3 );          // ACK packets always have priority 3 (highest)

     packetHeader.SetTypeFlag( theFlagByte );
}


void SendPacket::SetPriority( int thePriority )
{
     // make a copy of the flag byte
     FlagByte  theFlagByte( packetHeader.GetTypeFlag() );

     // set the priority
     theFlagByte.SetPrio( thePriority );
```

```
     packetHeader.SetTypeFlag( theFlagByte );
}


void SendPacket::SetToUseSequenceNumber()
{
     // if the header length is less than the number
     // of bytes in a header with a sequence number
     if( packetHeader.GetHeaderLength() < 6 )
     {
          // set to 6 bytes, this turns on the sequence number
          packetHeader.SetHeaderLength( 6 );
     }
     // else, header already includes sequence number
     // do nothing
}


void SendPacket::SetDestinationAddress( unsigned short theDestination )
{
     // if the header length is less than the number
     // of bytes in a header with a destination address
     if( packetHeader.GetHeaderLength() < 8 )
     {
          // set to 6 bytes, this turns on the destination address
          packetHeader.SetHeaderLength( 8 );

          // set the address
          packetHeader.SetDestinationAddress( theDestination );
     }
     // else, header already includes destination address
     // do nothing
}


void SendPacket::SetSourceAddress( unsigned short theSource )
{
     // if the header length is less than the number
     // of bytes in a header with a source address
     if( packetHeader.GetHeaderLength() < 10 )
     {
          // set to 6 bytes, this turns on the source address
          packetHeader.SetHeaderLength( 10 );

          // set the address
          packetHeader.SetSourceAddress( theSource );
     }
     // else, header already includes source address
     // do nothing
}


void SendPacket::PrepareBuffer()
{
     // if this packet is an ACK or NAQ type then zero the data
     FlagByte  theFlagByte( packetHeader.GetTypeFlag() );
     if( theFlagByte.isACKPkt() || theFlagByte.isNAQPkt() )
     {
          // zero out the data buffers
          ZeroPacket();

          // make a copy of the flag byte
          FlagByte  theFlagByte( packetHeader.GetTypeFlag() );
```

```
     // now set the request for ACK to no acknowledge
     // since this is an acknowledge packet
     theFlagByte.SetReqACK( false );

     packetHeader.SetTypeFlag( theFlagByte );
}

// reset the CRC
packetCRC.Reset();

// count how many extra bytes will be required for escapes
// and calculate the CRC while we're at it
int  escapeBytes = 0;

// for the header
for( int i = 0; i < packetHeader.GetHeaderLength(); ++i )
{
     // add to the CRC
     packetCRC( packetHeader[ i ] );

     if( ( packetHeader[ i ] == Header::OpenFlag ) || ( packetHeader[ i ] == Header::DLE ) )
     {
          ++escapeBytes;
     }
}

// for the data
for( int i = 0; i < dataBufferSize; ++i )
{
     // add to the CRC
     packetCRC( dataBuffer[ i ] );

     if( ( dataBuffer[ i ] == Header::OpenFlag ) || ( dataBuffer[ i ] == Header::DLE ) )
     {
          ++escapeBytes;
     }
}

// for the CRC
unsigned short int  theCRC( packetCRC );
for( int i = 0; i < sizeof( theCRC ); ++i )
{
     char tempByte = *( (unsigned char*)(&theCRC) + i );

     if( ( tempByte == Header::OpenFlag ) || ( tempByte == Header::DLE ) )
     {
          ++escapeBytes;
     }
}

// allocate a new buffer
// remove any previous buffer
if( outBuffer )
{
     delete [] outBuffer;
}

// size it based on the size of the header, the size of the data, the escape bytes, two for the CRC, and one for the open flag
outBufferSize = packetHeader.GetHeaderLength() + dataBufferSize + escapeBytes + 2 + 1;
outBuffer = new unsigned char[ outBufferSize ];

// copy the header information
```

```
     unsigned char* bufferPointer = outBuffer;

     // setup the output buffer
     // open flag
     *(bufferPointer++)  = Header::OpenFlag;

     // place the data into the output buffer
     // escape the data
     // the  Header
     for( int i = 0; i < packetHeader.GetHeaderLength(); ++i )
     {
          bufferPointer += EscapeAndWrite( packetHeader[ i ], bufferPointer );
     }

     // the actual data
     for( int i = 0; i < dataBufferSize; ++i )
     {
          bufferPointer += EscapeAndWrite( dataBuffer[ i ], bufferPointer );
     }

     // attach the CRC to the end
     for( int i = 0; i < sizeof( theCRC ); ++i )
     {
          bufferPointer += EscapeAndWrite( *( (unsigned char*)(&theCRC) + i ), bufferPointer );
     }
}


int  SendPacket::EscapeAndWrite( unsigned char theByte, unsigned char* theBuffer )
{
     // the return value
     int  RetVal = 1;

     if( theByte == Header::OpenFlag )
     {
          // for OpenFlags in the data stream
          *(theBuffer++) = Header::DLE;
          *(theBuffer)   = Header::OpenESC;

          // we wrote two bytes
          RetVal = 2;
     }
     else
     {
          if( theByte == Header::DLE )
          {
               // for ESC's in the data stream
               *(theBuffer++) = Header::DLE;
               *(theBuffer)   = Header::ESCESC;

               // we wrote two bytes
               RetVal = 2;
          }
          else
          {
               // for normal bytes in the data stream
               *(theBuffer) = theByte;
          }
     }

     return RetVal;
}
```

```
#ifdef  _NR_
SerialPort& operator<< ( SerialPort& s, SendPacket& thePacket )
{
     // make sure the packet is formatted
     thePacket.PrepareBuffer();

     s.SendBuffer( thePacket.outBuffer, thePacket.outBufferSize );

     return s;
}
#endif  _NR_

#pragma export reset
```