

XMODEM, XMODEM-1K, And XMODEM/CRC

Jonathan Ward

Jonathan Ward is a software engineer for PANDE, Inc. You may contact him at 14275 Midway Rd., Suite 220, Dallas, TX 75244 (214) 242-8628.

XMODEM, created in 1978 by Ward Christensen, is probably the most available file transfer protocol. XMODEM's simplicity and easy implementation have made it an immediate success. This article will discuss the original XMODEM protocol, as well as several of its many enhancements.

Original XMODEM

XMODEM is a half duplex, 8-bit protocol which transfers files by sending a portion of the file in a data structure called a packet and then waiting for a response to that packet. If the response is positive (indicating successful transmission) the next packet is sent. If the response is negative, the packet is resent.

The original XMODEM protocol transferred 128-byte data blocks with a 1-byte checksum for error detection. The layout of the XMODEM packet is:

SOH(0x01)
Packet Number
1's Complement of Packet Number
Data (128 bytes)
Arithmetic checksum

All fields except the data field are one byte. The data field contains 128 bytes of either ASCII or binary data that is usually obtained from the file being transferred.

[Figure 1](#) shows a simplified progression of a typical transmission. The receiver initiates the transfer by asking the sender to transmit the first data packet. For the original XMODEM, this start character is a *NAK (0x15)*. Once a packet has been sent, the sender waits for the receiver to respond to the packet. If the receiver responds with an *ACK (0x06)*, indicating that the packet was received successfully, the sender transmits the next packet. A *NAK* indicates the receiver did not receive the packet successfully and sender resends the packet.

After transferring all the packets, the sender signals end-of-transmission with an *EOT (0x04)* and waits for an *ACK*. The transmission is complete once the sender receives an *ACK*.

Receiver Time-Outs

The sender and receiver do not wait indefinitely for responses. The protocol sets maximum time and retry figures for the different transmission states.

The receiver initiates the transfer by sending a start character. If no data is received within 10 seconds, the start character is sent again. This process is repeated 10 times before the receiver gives up.

While receiving the packet data, the receiver allows a one-second time-out for each character. There is no retry count. If more than one second elapses between received characters, the packet is *NAKed*.

After receiving a packet, the receiver sends the packet response. The receiver then waits for the beginning of a packet or for an *EOT*. If it doesn't receive characters within 10 seconds, the packet response is sent again. If the receiver receives nothing after transmitting the response code 10 times, it aborts the download.

There is no time-out or retry on the receiver for the final *ACK*.

Sender Time-Outs

Once started, the sender simply waits for a start character from the receiver. If the sender doesn't receive a character or if it counts more than 10 invalid start characters within 60 seconds, it aborts the transfer.

After transferring a packet, the sender waits for the response from the receiver. The time-out for the response is 10 seconds. If no response is received within that time, the packet is resent. After 10 retransmissions of the same packet, the transfer will be aborted.

The final *EOT* has a time-out of 60 seconds. If no *ACK* is received within that time, the sender aborts the transfer and returns an appropriate error.

Relaxed Time-Outs

Several implementations of XMODEM offer a feature called relaxed time-outs or relaxed timing. Relaxed timing allows extra time between characters and packets for large systems like CompuServe. I have excluded relaxed timing from my implementation.

Graceful Abort

Original XMODEM does not provide any method for aborting a download in progress. However, the XMODEM protocol now supports a graceful abort enhancement. When the sender is waiting for a start character, an *ACK* or *NAK* after packet transmission, or the final *ACK*, a *CAN* (*0x18*) character indicates that the receiver has aborted the download.

Likewise, when the receiver is waiting for the first character of the packet, reception of a *CAN* character indicates that the sender has aborted the download.

Some implementations of XMODEM abort the download after receiving a single *CAN*, while others abort after three consecutive *CAN* characters. My implementation requires two consecutive *CANs* to abort. Because line noise can sometimes garble characters into a

CAN, requiring multiple consecutive *CAN* characters avoids aborting transfers due to line noise.

If you are trapped in an XMODEM transfer and wish to abort, hit *CTRL-X* several times. *CTRL-X* transmits the *CAN* character and should abort the transfer.

If the receiver or sender encounters an unrecoverable error condition, XMODEM should abort the transfer on both ends. My implementation sends eight *CAN* characters followed by eight BACKSPACES (0x08) (so that the *CAN*s are not displayed as received data).

XMODEM-1K

XMODEM-1K, an enhanced XMODEM, transmits 1,024 bytes of data per packet rather than the original 128 bytes. To indicate the larger packet size to the receiver, the first character of the packet sent is an *STX* (0x02). The layout of the XMODEM-1K packet is:

STX (0x02)

Packet Number

1's Complement of Packet Number

Data (1024 bytes)

Arithmetic checksum

XMODEM-1K may send data in either 128-byte packets or in the larger 1,024-byte packets. This flexibility necessitates the *STX* in the 1K packets. Once a packet has been sent as either 128-byte or 1,024-byte, it can't be resent in a different size packet.

Other than the larger packet size and the ability to mix large and small packets, XMODEM-1K does not differ from XMODEM.

XMODEM/CRC

XMODEM/CRC offers an increased error detection rate. XMODEM and XMODEM-1K use an arithmetic checksum to detect transfer errors. This 1-byte checksum detects 95 percent of all errors and is generated by summing all the bytes in the data field (excluding the *SOH/STX*, the packet number, and 1's complement packet number) modulo 256. However, XMODEM/CRC uses a 16-bit CRC (Cyclic Redundancy Check) instead of the checksum. This enhancement causes the error detection rate to exceed 99.997 percent.

The new packet layouts are:

128-byte packet

SOH (0x01)

Packet Number

1's Complement of Packet Number

Data (128 bytes)

16-bit CRC

1024-byte packet

STX (0x02)

Packet Number

1's Complement of Packet Number

Data (1024 bytes)
16-bit CRC

The CRC is calculated by using the binary representation of the data stream as the coefficients of a polynomial. The 128-byte packet would produce a polynomial of degree 1,023 (128 x 8 bits). This polynomial is then divided by the CRC polynomial using modulo-2 arithmetic, the remainder of which is the CRC. For XMODEM, the CRC polynomial is the CCITT polynomial:

$$x^{16} + x^{12} + x^5 + 1$$

The high-byte of the 16-bit CRC is sent first followed by the low-byte.

The receiver determines whether packets are sent using CRC or checksum to detect errors. If the receiver wishes to transfer with CRC error detection, it sends the start character *C* (0x43) with a three-second time-out and a retry count of four. If the sender cannot transmit using CRC, it ignores the *C* and continues waiting for a *NAK*. If sender does not respond to the *C* start character, the receiver will send a *NAK* to attempt transfer using the checksum.

The receiver should always attempt to transfer using CRC and then back down to checksum if the CRC attempt fails.

My Implementation

I have implemented XMODEM, XMODEM/CRC, and XMODEM-1K transfer protocols. Rather than developing separate routines to handle each combination of these, I have written one routine for sending and one for receiving. The receive routine attempts to transmit using CRC block checking but will fall back to checksum if the CRC request fails. An argument passed to the send routine specifies which data block size to use (128 or 1,024). See [Listing 1](#) for the header for the XMODEM routines.

Encapsulating XMODEM

I decided to make the transfer protocol oblivious to the serial I/O by providing medium-level serial functions for the transfer routines. Pointers to these functions are passed as arguments to the XMODEM transfer routines simplifying adaptation to different hardware configurations and avoiding the peculiarities of different UARTs, DUARTs, ACIAs, SIOs, etc.

To display transfer status information and manually abandon the transfer, pointers to functions for these purposes are also passed to the transfer routines.

Serial Receive Interface

I constructed two functions as an interface to the low-level serial routines. The first returns a single character received from the serial port. It is declared as:

```
int uart_receive (long ms_timeout,  
                 unsigned int *error_flag);
```

The *uart_receive* function returns the ASCII code for the received character or a value of

RECEIVER_TIMEOUT if a character is not received within the time specified in *ms_timeout*. *RECEIVER_TIMEOUT* may be defined as anything that is not a valid ASCII character. I chose *-1*.

The *ms_timeout* argument contains the number of milliseconds to wait for a character (if none has already been received). The constant *MS_PER_SEC* is defined as *1000L* to aid in conversions. If the time-out value is *0L* and no characters have been received, the *uart_receive* function should immediately time-out.

The *error_flag* argument points to an unsigned integer that will be initialized to indicate hardware errors detected, i.e., parity error, overrun. Several constants are defined as different bits in the error flag. They are:

```
#define RE_OVERRUN_    0x01
#define RE_PARITY_     0x02
#define RE_FRAMING_    0x04
```

You test these error conditions by *ANDing* the constant with the variable containing the error flags. A *TRUE* test indicates that flag is set and the error condition occurred. If you don't have access to these error conditions, then the XMODEM routines need never know them. Simply set the error flag to *0*.

Serial Transmit Interface

The second serial interface function transmits characters:

```
int uart_transmit (char xmit_char);
```

This function transmits the character value in the *xmit_char* argument. Two return values are possible: *TRANSMITTER_OK* indicates there are no errors with the transmitter, *TRANSMITTER_OFFLINE* indicates the receiving computer is no longer connected. The offline condition can be detected by loss of the carrier detect signal or some other hardware line. It may also be ignored entirely by always returning *TRANSMITTER_OK*.

You can use these functions with polled or interrupt-driven serial routines. Either way, the functions simplify the transfer protocol by keeping serial I/O out of the way.

Status Display Interface

A function to display the transfer status is declared as

```
void display_xfer_stats (
    long blocks_transferred,
    long bytes_transferred,
    const char *status_message);
```

The *bytes_transferred* argument contains the number of bytes that have been successfully transferred. If it is *-1L*, the number of bytes displayed should not be disturbed.

The *blocks_transferred* argument contains the total number of data blocks that have been transferred. If *blocks_transferred* is *-1L*, the total data blocks displayed should not be disturbed.

The *status_message* argument points to the status message to be displayed. It may also be *NULL* if no status message is to be displayed.

Abort Transfer Interface

A function for testing an abort condition may be provided to the send and receive routines. Its prototype is:

```
int check_abort (void);
```

The function returns a non-zero value if the user aborted the transfer manually by pressing the *ESC* or *CTRL-C* keys.

Receiving

[Listing 2](#) shows the XMODEM receive function. Its prototype is:

```
int xmodem_recv (
    FILE *f,
    int    (*transmit) (char),
    int    (*receive) (long,
        unsigned int *),
    void   (*dispstat) (long, long,
        const char *),
    int    (*check_abort) (void));
```

The *f* argument points to a file that must already be opened for writing.

To transfer data, the receiver begins by requesting a packet from the sender. The receiver sends a *C* in an attempt to transfer using CRC instead of checksum for error detection. If it doesn't receive a response within three seconds, it resends the *C*. This process is repeated four times. If the attempt to use CRCs fails, the receiver reverts to checksum error detection and sends a *NAK* to request the first data packet. If there is no response within 10 seconds, it sends another *NAK*. This is repeated 10 times. If there is no response from the sender after 10 *NAK*s the transmission is automatically aborted.

The only valid characters for the beginning of a packet are: *SOH*, for a 128-byte packet, *STX*, for a 1K packet, or two consecutive *CAN*s for an aborted transfer. All other characters are ignored.

SOH and *STX* signal the beginning of a packet. The next two bytes received are the block number (modulo 256 and starting with one) and the 1's complement of the block number respectively. These two bytes are compared, and if they don't represent the same block number, the packet will be *NAK*ed.

Following the block numbers are the data block and the CRC or checksum. The data portion of the packet is collected and a checksum or CRC is generated. After receiving the data block and the block check, the receiver compares the generated checksum or CRC to the received checksum or CRC. If they are not the same, the packet is *NAK*ed.

Once the receiver successfully receives a packet, it compares the received block number to the expected block and to the previously received block number. If the received block number doesn't match either of these other numbers, the packets have gotten out of

sequence and the transmission must be aborted since XMODEM does not resynchronize block numbers that are out of sequence.

If the received block number matches the next expected block number, the block data is written to the file and an *ACK* is sent.

If the received block number matches the previously received block number, the packet is *ACKed*, but the data is *not* written since it was written when the block was successfully received the first time. The received block number might match the previously received block number for two reasons. First, if the *ACK* for the previous block was converted into a *NAK* because of line noise, the sender would resend the last packet. Second, some senders may assume that anything that is not an *ACK* must be a *NAK*, and would resend the last packet if the *ACK* was garbled.

After successfully receiving a packet, the receiver must wait for the start of the next packet or an *EOT*. For the sender to know that the file was successfully received, the *EOT must be ACKed*.

Sending

[Listing 3](#) shows the XMODEM send function. The prototype for the XMODEM send function is:

```
int xmodem_send (
    FILE *f,
    int (*transmit) (char),
    int (*receive) (long,
        unsigned int *),
    void (*dispstat) (long, long,
        const char *),
    int (*check_abort) (void));
```

The *f* argument points to the file to be transferred. This file must already be open for reading.

The sender is much simpler than the receiver. It begins by waiting for a start character. Valid characters are: *C*, to indicate CRC block checking; *NAK*, to indicate checksum block checking; or two consecutive *CANs*, to indicate that the transfer has been aborted. If the sender receives no character within 60 seconds, or if it receives 10 invalid characters, the transfer is aborted.

After receiving a valid start character, the sender prepares and transmits the first packet, including the block start character (*SOH* or *STX* depending on the data size), the block number, the one's complement block number, the data block, and the checksum or CRC.

The sender waits for a response from the receiver. Valid responses are: *ACK*, indicating that the packet was successfully received; *NAK*, indicating the packet was not successfully received; or two consecutive *CANs*, to cancel the transfer. All other characters are invalid and are ignored. If the sender doesn't receive a character within 10 seconds (or receives a *NAK*), it retransmits the packet. If it receives an *ACK*, the sender prepares and sends the next packet, and the cycle is repeated. After the last packet is sent and *ACKed*, the sender sends an *EOT* to indicate end of transfer. The receiver must *ACK* to *EOT* for the sender to

know that the transfer completed successfully.

XMODEM Problems

XMODEM presents several problems:

- The original file size is not preserved in the transmission.
- The modification time and date stamp, as well as the original filename, are lost.
- The last packet is padded with CP/M *EOF* (0x1A) characters, which can cause problems under operating systems where file size and check information are embedded in the file header. I believe this is the case with the Commodore Amiga.
- Only one file can be sent at a time.

The YMODEM protocol makes several enhancements to XMODEM to solve these problems.

Does It Work?

I have successfully used my XMODEM routines to transfer files to and from local BBSs with great success.

Different implementations of XMODEM handle different events. The length of time-outs varies widely. Some implementations will not fall back to checksum error detection if the sender doesn't recognize the CRC start character. Error situations seem to be handled differently as well. I handled these events following whatever documentation I could find.

After implementation, I discovered a problem with interrupt-driven, buffered serial I/O, which involved the sender's packet response time-out. After transmitting a packet, the sender immediately begins waiting for the response. While this situation works fine with polled I/O, interrupt-driven serial routines buffer the data to send. When the sender starts waiting for the response, very little of the packet may have been sent. At 300 baud, it takes 35 seconds to transfer a 1K packet, and I'd bet that even a 4.77 MHz PC doesn't need 35 seconds to dump 1,000 bytes into a buffer. Even at 1,200 baud, the time-out is cut pretty close. It is left as an exercise for the reader to determine the best method of solving this problem. Here are some suggestions:

- Ignore the problem, buy a faster modem (9,600 baud or so), and be done with it.
- Pad the time-out values to account for the baud rate.
- Modify the transmit routine to wait until the transmit buffer is empty before returning.
- Devise a method of checking the number of characters in the transmit buffer.

Documentation

The best documentation for XMODEM and its derivatives is online. I located a half dozen document files describing the XMODEM protocol. Some were just an overview while others were quite in-depth. Interestingly enough, the texts I own which describe XMODEM don't do nearly as well as the public domain document files.

Conclusion

XMODEM and its extensions are easy to implement. If you need to implement file transfer capability in an application, XMODEM is a good choice if the volume of data to transmit is small or if you are using a slow modem. If you are using a high-speed modem or are transferring large volumes of data, other transfer protocols like YMODEM-G and ZMODEM are more efficient, but a little more difficult to implement.