AVR350: XmodemCRC Receive Utility for AVR®

8-bit AVR® Microcontroller

Application Note

Features

- **Programmable Baud Rate**
- **Half Duplex**
- **128-byte Data Packets**
- **CRC Data Verification**
- **Framing Error Detection**
- **OverRun Detection**
- **Less than 1K Bytes of Code Space**
- **C High-level Language Code**

Introduction

The Xmodem protocol was created years ago as a simple means of having two computers talk to each other. With its half-duplex mode of operation, 128-byte packets, ACK/NACK responses and CRC data checking, the Xmodem protocol has found its way into many applications. In fact most communication packages found on the PC today have a Xmodem protocol available to the user.

Theory of Operation

Xmodem is a half-duplex communication protocol. The Receiver, after receiving a packet, will either acknowledge (ACK) or not acknowledge (NACK) the packet. The original Xmodem protocol used a standard checksum method to verify the 128-byte data packet. The CRC extension to the original protocol uses a more robust 16-bit CRC to validate the data block and is used here. Xmodem can be considered to be receiver driven. That is, the Receiver sends an initial character “C” to the sender indicating that it’s ready to receive data in CRC mode. The Sender then sends a 133-byte packet, the Receiver validates it and responds with an ACK or a NACK at which time the sender will either send the next packet or re-send the last packet. This process is continued until an EOT is received at the Receiver side and is properly ACKed to the Sender. After the initial handshake the receiver controls the flow of data through ACKing and NAKing the Sender.

Table 1. XmodemCRC Packet Format

Byte 1	Byte 2	Byte 3	Bytes 4 - 131	Bytes 132 - 133
Start of Header	Packet Number	~(Packet Number)	Packet Data	16-bit CRC

Rev. 1472B–AVR–05/02

Definitions

The following defines are used for protocol flow control.

Table 2. Protocol Flow Control

Symbol	Description	Value
SOH	Start of Header	0x01
EOT	End of Transmission	0x04
ACK	Acknowledge	0x06
NAK	Not Acknowledge	0x15
C	ASCII "C"	0x43

Byte one of the XmodemCRC packet can only have a value of SOH or EOT, anything else is an error. Bytes two and three form a packet number with checksum, add the two bytes together and they should always equal 0xff. Please note that the packet number starts out at "1" and rolls over to "0" if there are more than 255 packets to be received. Bytes 4 - 131 form the data packet and can be anything. Bytes 132 and 133 form the 16-bit CRC. The high byte of the CRC is located in byte 132.

Synchronization

The Receiver starts by sending an ASCII "C" (0x43) character to the sender indicating it wishes to use the CRC method of block validating. After sending the initial "C" the receiver waits for either a three second time out or until a buffer full flag is set. If the receiver is timed out then another "C" is sent to the sender and the three second time out starts again. This process continues until the receiver receives a complete 133-byte packet.

Receiver Considerations

This protocol NACKs the following conditions:

1. Framing error on any byte
2. OverRun error on any byte
3. Duplicate packet
4. CRC error
5. Receiver timed out (didn't receive packet within one second)

On any NAK, the sender will re-transmit the last packet. Items one and two should be considered serious hardware failures. Verify that sender and receiver are using the samebaud rate, start bits and stop bits. Item three is usually the sender getting an ACK garbled and re-transmitting the packet. Item four is found in noisy environments. And the last issue should be self-correcting after the receiver NAKs the sender.

Data Flow Diagram

The data flow diagram below simulates a 5-packet file being sent.

Table 3. XmodemCRC Data Flow with Errors

Sender						Receiver
					<----	"C"
						Times Out after Three Seconds
					<----	"C"
SOH	0x01	0xFE	Data	CRC	---->	Packet OK
					<----	ACK
SOH	0x02	0xFD	Data	CRC	---->	(Line Hit during Data Transmission)
					<----	NACK
SOH	0x02	0xFD	Data	CRC	---->	Packet OK
					<----	ACK
SOH	0x03	0xFC	Data	CRC	---->	Packet OK
(ACK Gets Garbled)					<----	ACK
SOH	0x03	0xFC	Data	CRC	---->	Duplicate Packet
					<----	NACK
SOH	0x04	0xFB	Data	CRC	---->	(UART Framing Error on Any Byte)
					<----	NACK
SOH	0x04	0xFB	Data	CRC	---->	Packet OK
					<----	ACK
SOH	0x05	0xFA	Data	CRC	---->	(UART Overrun Error on Any Byte)
					<----	NACK
SOH	0x05	0xFA	Data	CRC	---->	Packet OK
					<----	ACK
EOT					---->	Packet OK
(ACK Gets Garbled)					<----	ACK
EOT					---->	Packet OK
Finished					<----	ACK

Modifications to Receive Protocol

Users may wish to count how many “C’s” were sent during synchronization and after “n” number of tries abort the receive attempt.

For embedded applications it’s not mandatory to have a 128-byte packet. You could have 64, 32, or even a 16-byte packet. The sender of course would have to comprehend this. For users that may want to migrate to Atmel's MegaAVR series there is a version of Xmodem that uses a 1Kbyte packet. Or you can use an external SRAM with an AT90S4414 or an AT90S8515 to allow the increase in packet size.

If users do not wish to use the CRC method of data verification, simply replace sending a “C” for synchronization with a NAK instead. The sender will then send only the simple checksum of the data packet. Of course, the buffer size decreases by one and data errors may occur. This modification would allow communication with equipment that supports only the checksum method of data verification.

Software

Routines were compiled using IAR’s “C” compiler version 1.40 with max size optimization. The software was tested using ProComm, DynaComm, WinComm, and Hyperterminal at baud rates up to 115.2K bps. The receiver expects 8 start bits, 1 stop bit, and no parity bits.

The STK200 starter kit is used as a test platform with minor, optional, modifications. A baud rate friendly crystal was used for this code. Replace the 4.0 MHz crystal on the STK200 starter kit with a 7.3728 MHz crystal for proper operation. If users wish to use the default crystal then modify the init routine to properly set up the uart baud rate register UBRR. Wait loops in the sendc and the recv_wait routines would also need modification.

To verify proper operation of this code, the PORT D bit 2 should be connected to the switches on the starter kit. Refer to the STK200 user manual for jumper locations and definitions. Connect a 9-pin serial cable from a PC to the starter kit, turn on power and use pushbutton two as a start of reception signal. Use an ICEPRO Emulator, an AT90S4414-8PC, or an AT90S8515-8PC to execute the code.

Table 4. Routines

Name	Size in Bytes	Function
calcrc	60	Calculates 16-bit CRC
init	30	Low-level Hardware Initialization
main	280	Main
purge	36	Reads UART Data Register for One Second
receive	64	Main Receive Routine
recv_wait	40	Waits until Buffer Full Flag is Set or One Second Timeout
respond	44	Sends an ACK or a NACK to the Sender
sendc	88	Sends an ASCII “C” Character to the Sender until the Buffer Full Flag is Set
timer1	28	Timer1 Interrupt
uart	80	Uart Receive Interrupt
validate_packet	155	Validates Senders Packet

Pseudo-Code

purge.c

```
initialize timer1 counter for a 1 second delay read uart for 1 second
```

receive.c

```
send a 'C' character to sender until receive buffer is full validate received packet send an ack or a nak to
sender
    if packet was bad then wait for new  good packet
while not end of transmission
    wait for buffer to fill
    validate the packet
    send an ack or a nak to sender
```

recv_wait.c

```
initialize timer1 counter for a 1 second delay wait till buffer is full or timeout
```

respond.c

```
clear error flags
    if packet was good a duplicate packet or end of transmission then
send an ack
    else
purge senders uart transmit buffer
    send a nack
```

sendc.c

```
initialize timer1 counter for a 3 second delay
    clear error flags
while buffer is not full
    send 'C' character to sender, signaling CRC mode
enable timer counter
    wait for buffer full or timeout
if timed out clear error flags
    restart timer
```

uart.c

```
check uart for framing or overrun errors
    read byte from uart
verify first byte in receive buffer is valid
    if buffer is full set buffer full flag
```

validate_packet.c

```
if not timed out then
    if no uart framing or overrun errors then
    if first character in buffer is SOH then
        if second character in buffer is the next packet number
then
        if second character in buffer plus the third character in buffer = 0xff
then
        compute CRC on packet data
        if CRC ok
then
    increment packet number
    packet = good
else
    packet = bad
else
    bad packet number checksum
else
    duplicate packet number
else
    if first character in buffer is EOT then
    end of transmission
else
    at least 1 byte had a framing or overrun error, packet is bad
else
    timed-out without receiving all characters, packet is bad
```

Code Listing

Calcrc.c

```
#include "xmodem.h"

int calcrc(char *ptr, int count)
{
    int crc;
    char i;

    crc = 0;
    while (--count >= 0)
    {
        crc = crc ^ (int) *ptr++ << 8;
        i = 8;
        do
        {
            if (crc & 0x8000)
                crc = crc << 1 ^ 0x1021;
            else
                crc = crc << 1;
        } while(--i);
    }
    return (crc);
}
```

Init.c

```
#include "xmodem.h"

void init(void)
{

    // portd bit 2 used to start data reception
    // Pb7, Pb6, Pb5, Pb4, Pb3, Pb2, Pb1, Pb0
    //  O    O    O    O    O    O    O    O
    //  1    1    1    1    1    1    1    1
    DDRD = 0xfb;
    PORTD = 0xff;
    TCCR1A = 0x00;                          // timer/counter 1 PWM disable
    TCCR1B = 0x00;                          // timer/counter 1 clock disable

    TIMSK |= 0x80;                          // enable timer counter 1 interrupt on overflow

    UCR = 0x98;                             // enable receiver, transmitter, and receiver interrupt
//  UBRR = 23;                              // 19.2k with 7.3728Mhz crystal
//  UBRR = 11;                              // 38.4k with 7.3728Mhz crystal
//  UBRR = 7;                               // 57.6k with 7.3728Mhz crystal
    UBRR = 3;                               // 115.2k with 7.3728Mhz crystal

}
```

Main.c

```
#include "xmodem.h"

volatile unsigned char buf[133];

struct global
{
  volatile unsigned char *recv_ptr;
  volatile unsigned char buffer_status;
  volatile unsigned char recv_error;
  volatile unsigned char t1_timed_out;
} gl;

//unsigned char packet_number;

// function prototypes
void receive(volatile unsigned char *bufptr1);
void purge(void);
void init(void);

void C_task main(void)
{
    init();                              // low level hardware initialization
    _SEI();                              // enable interrupts


    purge();     // clear uart data register ... allow transmitter opportunity to unload its buffer

    do
    {
      while (PIND &= 0x04);              // wait until pd2 pulled low
      receive(&buf[0]);
    }while (1);

} // main
```

Purge.c

```
#include "xmodem.h"

extern struct global
{
  volatile unsigned char *recv_ptr;
  volatile unsigned char buffer_status;
  volatile unsigned char recv_error;
  volatile unsigned char t1_timed_out;
} gl;

// wait 1 second for sender to empty its transmit buffer
void purge(void)
{
  unsigned char flush;

    gl.t1_timed_out = false;

    // 1 second timeout
    // 7.3728MHz / 1024 = 7200 Hz
    // 7200 Hz = 138.8 us
    // 1 seconds / 138.8 us = 7200
    // 65536 - 7200 = 58336 = e3e0
    // interrupt on ffff to 0000 transition
    TCNT1H = 0xe3;
    TCNT1L = 0xe0;                            // load counter
    TCCR1B = 0x05;                            // timer/counter 1 clock / 1024

    while (!gl.t1_timed_out)                  // read uart until done
    {
        flush = UDR;
    }
    TCCR1B = 0x00;                            // disable timer/counter 1 clock
}
```

Receive.c

```
#include "xmodem.h"

extern struct global
{
  volatile unsigned char *recv_ptr;
  volatile unsigned char buffer_status;
  volatile unsigned char recv_error;
  volatile unsigned char t1_timed_out;
} gl;

// function prototypes
unsigned char validate_packet(unsigned char *bufptr, unsigned char packet_number);
void respond(unsigned char packet);
void recv_wait(void);
void sendc(void);

void receive(volatile unsigned char *bufptr1)
{
  unsigned char packet;                            // status flag
  unsigned char packet_number;

  packet_number = 0x00;                            // xmodem packets start at 1
  gl.recv_ptr = bufptr1;                           // point to recv buffer

  sendc();                                         // send a 'c' until the buffer gets full
  packet = validate_packet(bufptr1,packet_number); // validate packet 1
  gl.recv_ptr = bufptr1;                           // re-initialize buffer pointer before acknowledging
  respond(packet);                                // ack or nak

  while (packet != good)                          // if we nak'ed above wait for packet 1 again
  {
    recv_wait();
    packet = validate_packet(bufptr1,packet_number); // validate packet 1
    gl.recv_ptr = bufptr1;                        // re-initialize buffer pointer before acknowledging
    respond(packet);                              // ack or nak
  }
  while (packet != end)                           // get remainder of file
  {
    recv_wait();                                  // wait for error or buffer full
    packet = validate_packet(bufptr1,packet_number);  // validate the packet
    gl.recv_ptr = bufptr1;                        // re-initialize buffer pointer before acknowledging
    respond(packet);                              // ack or nak
  }                                               // end of file transmission
}
```

Recv_wait.c

```
#include "xmodem.h"

extern struct global
{
  volatile unsigned char *recv_ptr;
  volatile unsigned char buffer_status;
  volatile unsigned char recv_error;
  volatile unsigned char t1_timed_out;
} gl;

void recv_wait(void)
{
    gl.t1_timed_out = false;                     // set in timer counter 0 overflow interrupt routine

    // 1 second timeout
    // 7.3728MHz / 1024 = 7200 Hz
    // 7200 Hz = 138.8 us
    // 1 seconds / 138.8 us = 7200
    // 65536 - 7200 = 58336 = e3e0
    // interrupt on ffff to 0000 transition
    TCNT1H = 0xe3;
    TCNT1L = 0xe0;                             // load counter
    TCCR1B = 0x05;                             // timer/counter 1 clock / 1024
                                               // wait for packet, error, or timeout
    while (!gl.buffer_status && !gl.t1_timed_out);
                                               // turn off timer - no more time outs needed
    TCCR1B = 0x00;                             // disable timer/counter 1 clock
}
```

Respond.c

```
#include "xmodem.h"

extern struct global
{
  volatile unsigned char *recv_ptr;
  volatile unsigned char buffer_status;
  volatile unsigned char recv_error;
  volatile unsigned char t1_timed_out;
} gl;

// function prototypes
void purge(void);

void respond(unsigned char packet)
{
    // clear buffer flag here ... when acking or nacking sender may respond
    // very quickly.
    gl.buffer_status = empty;
    gl.recv_error = false;                       // framing and over run detection

    if ((packet == good) || (packet == dup) || (packet == end))
    {
        while (!(USR & 0x20));                   // wait till transmit register is empty
        UDR = ACK;                               // now for the next packet
    }
    else
    {
        while (!(USR & 0x20));                   // wait till transmit register is empty
        purge();                                 // let transmitter empty its buffer
        UDR = NAK;                               // tell sender error
    }
}
```

Sendc.c

```
#include "xmodem.h"

extern struct global
{
  volatile unsigned char *recv_ptr;
  volatile unsigned char buffer_status;
  volatile unsigned char recv_error;
  volatile unsigned char t1_timed_out;
} gl;

void sendc(void) {

  // 3 second timeout
```

```
    // 7.3728MHz / 1024 = 7200 Hz
    // 7200 Hz = 138.8 us
    // 3 seconds / 138.8 us = 21600
    // 65536 - 21600 = 43936 = aba0
    // interrupt on ffff to 0000 transition
    TCNT1H = 0xab;
    TCNT1L = 0xa0;                              // load counter
    TCCR1B = 0x00;                              // disable timer/counter 1 clock

    // enable entry into while loops
    gl.buffer_status = empty;
    gl.t1_timed_out = false;
    gl.recv_error = false;                      // checked in validate_packet for framing or overruns

    // send character 'C' until we get a packet from the sender
    while (!gl.buffer_status)
    {
        // tell sender CRC mode
        while (!(USR & 0x20));                  // wait till Data register is empty
        UDR = CRCCHR;                  // signal transmitter that I'm ready in CRC mode ... 128 byte packets
        TCCR1B = 0x05;                          // timer/counter 1 clock / 1024

        // wait for timeout or recv buffer to fill
        while (!gl.t1_timed_out && !gl.buffer_status);
       // turn off timer
        TCCR1B = 0x00;                        // disable timer/counter 1 clock
       if (gl.t1_timed_out)                     // start wait loop again
        {
            gl.t1_timed_out = false;
            TCNT1H = 0xab;
            TCNT1L = 0xa0;                    // load counter ... start over
        }
     }
}
```

Timer1.c

```
#include "xmodem.h"

extern struct global
{
  volatile unsigned char *recv_ptr;
  volatile unsigned char buffer_status;
  volatile unsigned char recv_error;
  volatile unsigned char t1_timed_out;
} gl;

interrupt [TIMER1_OVF1_vect] void TIMER1_OVF1_interrupt(void)
{
    gl.t1_timed_out = true;
}
```

Uart.c

```
#include "xmodem.h"

extern volatile unsigned char buf[133];

extern struct global
{
  volatile unsigned char *recv_ptr;
  volatile unsigned char buffer_status;
  volatile unsigned char recv_error;
  volatile unsigned char t1_timed_out;
} gl;

interrupt [UART_RX_vect] void UART_RX_interrupt(void)
{
// use local pointer until IAR optimizes pointer variables better in the next release
    volatile unsigned char *local_ptr;

    local_ptr = gl.recv_ptr;
                              // check for errors before reading data register ... reading UDR clears status
    if (USR & 0x18)           // Framing or over run error
    {
      gl.recv_error = true;   // will NAK sender in respond.c
    }                         // always read a character otherwise another interrupt could get generated
                              // read status register before reading data register
    *local_ptr++ = UDR;       // get char
    switch (buf[0])           // determine if buffer full
    {
        case (SOH) :
            if (local_ptr == (&buf[132] + 1))
            {
                gl.buffer_status = full;
                local_ptr = &buf[0];
            }
        break;
/*        case (EOT) :
            gl.buffer_status = full;
            local_ptr = &buf[0];
        break;*/
        default :
            gl.buffer_status = full;    // first char unknown
            local_ptr = &buf[0];
        break;
    }
    gl.recv_ptr = local_ptr;            // restore global pointer
}
```

validate_packet.c

```
#include "xmodem.h"

extern struct global
{
  volatile unsigned char *recv_ptr;
  volatile unsigned char buffer_status;
  volatile unsigned char recv_error;
  volatile unsigned char t1_timed_out;
} gl;

// function prototypes
int calcrc(char *ptr, int count);

unsigned char validate_packet(unsigned char *bufptr,unsigned char packet_number) {

unsigned char packet;
int crc;

    packet = bad;
    if (!gl.t1_timed_out)
    {
        if (!gl.recv_error)
        {
            if (bufptr[0] == SOH)
            {                                                   // valid start
                if (bufptr[1] == ((packet_number+1) & 0xff))
                {                                               // sequential block number ?
                    if ((bufptr[1] + bufptr[2]) == 0xff)
                   {                                        // block number and block number checksum are ok
                        crc = calcrc(&bufptr[3],128);// compute CRC and validate it
                      if ((bufptr[131] == (unsigned char)(crc >> 8)) && (bufptr[132] == (unsigned char)(crc)))
                        {
                            packet_number++;                        // good packet ... ok to increment
                            packet = good;
                        }
                    }                                 // block number checksum
                }                                   // bad block number or same block number
                else if (bufptr[1] == ((packet_number) & 0xff))
                {                                   // same block number ... ack got glitched
                    packet = dup;                   // packet is previous packet don't inc packet number
                }
            }
            // check for the end
            else if (bufptr[0] == EOT)
                packet = end;
        }
        else
            packet = err;
        }
    else
        packet = out;
    return (packet);
}
```

xmodem.h

```
#include "io8515.h"
#include "ina90.h"
#pragma language=extended

#define SOH 01
#define EOT 04
#define ACK 06
#define NAK 25
#define CRCCHR 'C'

#define true 0xff
#define false 0x0

#define full 0xff
#define empty 0x00

#define bad 0x00
#define good 0x01
#define dup 0x02
#define end 0x03
#define err 0x04
#define out 0x05
```

Atmel Headquarters

Corporate Headquarters
2325 Orchard Parkway
San Jose, CA 95131
TEL 1(408) 441-0311
FAX 1(408) 487-2600

Europe
Atmel Sarl
Route des Arsenaux 41
Case Postale 80
CH-1705 Fribourg
Switzerland
TEL (41) 26-426-5555
FAX (41) 26-426-5500

Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimhatsui
East Kowloon
Hong Kong
TEL (852) 2721-9778
FAX (852) 2722-1369

Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
TEL (81) 3-3523-3551
FAX (81) 3-3523-7581

Atmel Operations

Memory
2325 Orchard Parkway
San Jose, CA 95131
TEL 1(408) 441-0311
FAX 1(408) 436-4314

Microcontrollers
2325 Orchard Parkway
San Jose, CA 95131
TEL 1(408) 441-0311
FAX 1(408) 436-4314

La Chantrerie
BP 70602
44306 Nantes Cedex 3, France
TEL (33) 2-40-18-18-18
FAX (33) 2-40-18-19-60

ASIC/ASSP/Smart Cards
Zone Industrielle
13106 Rousset Cedex, France
TEL (33) 4-42-53-60-00
FAX (33) 4-42-53-60-01

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906
TEL 1(719) 576-3300
FAX 1(719) 540-1759

Scottish Enterprise Technology Park
Maxwell Building
East Kilbride G75 0QR, Scotland
TEL (44) 1355-803-000
FAX (44) 1355-242-743

RF/Automotive
Theresienstrasse 2
Postfach 3535
74025 Heilbronn, Germany
TEL (49) 71-31-67-0
FAX (49) 71-31-67-2340

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906
TEL 1(719) 576-3300
FAX 1(719) 540-1759

Biometrics/Imaging/Hi-Rel MPU/
High Speed Converters/RF Datacom
Avenue de Rochepleine
BP 123
38521 Saint-Egreve Cedex, France
TEL (33) 4-76-58-30-00
FAX (33) 4-76-58-34-80

e-mail
literature@atmel.com

Web Site
http://www.atmel.com