# File Transfer by XModem Protocol Using UART Module

***V1.3 - Dec 22, 2006***

***English Version***

19, Innovation First Road • Science Park • Hsin-Chu • Taiwan 300 • R.O.C.

Tel: 886-3-578-6005 Fax: 886-3-578-4418 E-mail: mcu@sunplus.com

http://www.sunplusmcu.com http://mcu.sunplus.com

**Important Notice**

SUNPLUS TECHNOLOGY CO. reserves the right to change this documentation without prior notice. Information provided by SUNPLUS TECHNOLOGY CO. is believed to be accurate and reliable. However, SUNPLUS TECHNOLOGY CO. makes no warranty for any errors which may appear in this document. Contact SUNPLUS TECHNOLOGY CO. to obtain the latest version of device specifications before placing your order. No responsibility is assumed by SUNPLUS TECHNOLOGY CO. for any infringement of patent or other rights of third parties which may result from its use. In addition, SUNPLUS products are not authorized for use as critical components in life support systems or aviation systems, where a malfunction or failure of the product may reasonably be expected to result in significant injury to the user, without the express written approval of Sunplus.

# Revision History

| Revision | Date | Translated By | Remark | Page Number(s) |
|---|---|---|---|---|
| V1.3 | 2006/12/22 | Qin Haiyan | Proofreading | |
| V1.2 | 2006/03/28 | Qin Haiyan | Translate 'File Transfer by XModem Protocol Using UART Module V1.2, Chinese version' | |

# Table of Content

# 1 System Design Summary

## 1.1 System Design

This design adopts UART on SPMC75F2413A to upload files (PC->SPMC75F2413A) and ensures accurate transmission in the support of XModem protocol, thus providing a feasible implementation for mass data transmission and driver reloading.

## 1.2 XModem Introduction

FTPs (File Transfer Protocols) of serial communication mainly include: XModem, YModem, ZModem and KERMIT.

XModem generically supports 128-byte packet, checksum and CRC, multiple retransmissions (generically 10 times) at error. It works through receiving program and sending program. First, the sending program sends negotiation character requesting for check method, followed by data packet transmission if succeeds. Second, the receiving program checks the data packet according to check method after receiving an integrated packet. Third, the receiving program sends acknowledgement character if the check succeeds and the sending program will proceed to transmit the next packet. If the check fails, negative acknowledgement character will be sent and the sending program would retransfer this packet.

## 1.3 XModem Protocol

### 1.3.1 Related Description

1. Definition: <SOH> 01H, <EOT> 04H, <ACK> 06H, <NAK> 15H, <CAN> 18H.

2. UART format: Asynchronous, 8-bit data, no parity, 1-bit stop.

### 1.3.2 XModem Introduction

XModem was originally developed in 1970s by Ward Christensen. Its transmission unit is data packet, which consists of a header start character <SOH>, one byte packet number and its complementary code, 128-byte data and one byte checksum. Data is divided into 128-byte packets to send. Each packet would be checked: If correct, the receiver will respond an <ACK> byte; if not, it will respond an <NAK> byte to request for retransmitting. Therefore, XModem is a “send-and-wait” protocol with flow control. It features a general-purpose advantage and is compatible with almost all communication software. But on the other hand, it slows down the communication speed. The data specification is shown in Figure 1-1.

| Byte1 | Byte2 | Byte3 | Byte4 -- 131 | Byte132 |
|---|---|---|---|---|
| Start Of Header | Packet Number | ~(Packet Number) | Packet Data | CheckSum |

Figure 1-1 Checksum Packet Specification

XModem protocol has been revised in the 1990s. The transmission unit is still data packet, which consists of a header start character SOH, a packet number byte and its complementary code, 128 data bytes and a CRC16 word. This new data specification is shown in Figure 1-2.

| Byte1 | Byte2 | Byte3 | Byte4 -- 131 | Byte132 -- 133 |
|---|---|---|---|---|
| Start Of Hearder | Packet Number | ~(Packet Number) | Packet Data | 16-Bit CRC |

Figure 1-2 CRC Packet Specification

### 1.3.3 Checksum Packet

- Checksum packet

<SOH><blk #><255-blk #><--128 data bytes--><cksum>

Where:

<SOH> = 01 hex

<blk #> = Packet number starts at 01, and increases by one until it reaches FF hex, then repeats.

<255-blk #> = the complementary code of packet number.

<cksum> = Retain byte, discard checksum.

- Data transmission with checksum

The receiver sends an NAK to request the sender for data transmission with checksum. The sender will respond accordingly. Table 1-1 shows five block transmission.

Table 1-1 Checksum Data Transmission

| Sender | | | | | Flow | Receiver |
|---|---|---|---|---|---|---|
| | | | | | <--- | NAK |
| | | | | | | Time out after 3 seconds |
| | | | | | <--- | NAK |
| SOH | 0x01 | 0xFE | Data[0-127] | Chksum | ---> | Packet OK |
| | | | | | <--- | ACK |
| SOH | 0x02 | 0xFD | Data[0-127] | Chksum | ---> | Line hit during transmission |
| | | | | | <--- | NAK |
| SOH | 0x02 | 0xFD | Data[0-127] | Chksum | ---> | Packet OK |

| Sender | Flow | Receiver |
|---|---|---|
| | <--- | ACK |
| SOH 0x03 0xFC Data[0-127] Chksum | ---> | Packet OK |
| ACK get garbaged | <--- | ACK |
| SOH 0x03 0xFC Data[0-127] Chksum | ---> | Duplicate packet |
| | <--- | NAK |
| SOH 0x04 0xFB Data[0-127] Chksum | ---> | UART Framing err on any byte |
| | <--- | NAK |
| SOH 0x04 0xFB Data[0-127] Chksum | ---> | Packet OK |
| | <--- | ACK |
| SOH 0x05 0xFA Data[0-127] Chksum | ---> | UART Overrun err on any byte |
| | <--- | NAK |
| SOH 0x05 0xFA Data[0-127] Chksum | ---> | Packet OK |
| | <--- | ACK |
| EOT | ---> | Packet OK |
| ACK get garbaged | <--- | ACK |
| EOT | ---> | Packet OK |
| Finished | <--- | ACK |

### 1.3.4 CRC Data Packet

- CRC data packet

<SOH><blk #><255-blk #><--128 data bytes--><CRC hi><CRC lo>

Where:

<SOH> = 01 hex

<blk #> = Packet number starts at 01 and increases by one until it reaches FF hex, then repeats.

<255-blk #> = The complementary code of packet number.

<CRC hi> = CRC16 high byte.

<CRC lo> = CRC16 low byte.

- CRC description

The divisor multinomial to calculate 16-bit CRC is X^16 + X^12 + X^5 + 1, by which the 128 bytes will be checked. The high bytes of CRC16 are ahead of low bytes at the sender.

- CRC data transmission flow

The receiver sends 'C' to request the sender for CRC transmission, the sender will respond correspondingly. Table 1-2 shows a three block transmission.

Table 1-2 CRC Data Transmission

| Sender | | | | | Flow | Receiver |
|---|---|---|---|---|---|---|
| | | | | | <--- | 'C' |
| | | | | | | Time out after 3 seconds |
| | | | | | <--- | 'C' |
| SOH | 0x01 | 0xFE | Data[0-127] | CRC16 | ---> | Packet OK |
| | | | | | <--- | ACK |
| SOH | 0x02 | 0xFD | Data[0-127] | CRC16 | ---> | Line hit during transmission |
| | | | | | <--- | NAK |
| SOH | 0x02 | 0xFD | Data[0-127] | CRC16 | ---> | Packet OK |
| | | | | | <--- | ACK |
| SOH | 0x03 | 0xFC | Data[0-127] | CRC16 | ---> | Packet OK |
| | | | | | <--- | ACK |
| EOT | | | | | ---> | Packet OK |
| ACK get garbaged | | | | | <--- | ACK |
| EOT | | | | | ---> | Packet OK |
| Finished | | | | | <--- | ACK |

- If the sender supports only checksum transmission, the receiver should send an NAK to request for Checksum data transmission, see Table 1-1; if only CRC data transmission, the receiver should send 'C' to request for CRC transmission, see Table 1-2; if both, 'C' will have the priority to be sent for request. The receiving process is shown in Table 1-3.

Table 1-3 CRC Data Transmission

| Sender | | | | | Flow | Receiver |
|---|---|---|---|---|---|---|
| | | | | | <--- | 'C' |
| | | | | | | Time out after 3 seconds |
| | | | | | <--- | NAK |
| | | | | | | Time out after 3 seconds |
| | | | | | <--- | 'C' |
| | | | | | | Time out after 3 seconds |
| | | | | | <--- | NAK |
| SOH | 0x01 | 0xFE | Data[0-127] | Chksum | ---> | Packet OK |
| | | | | | <--- | ACK |
| SOH | 0x02 | 0xFD | Data[0-127] | Chksum | ---> | Line hit during transmission |
| | | | | | <--- | NAK |
| SOH | 0x02 | 0xFD | Data[0-127] | Chksum | ---> | Packet OK |
| | | | | | <--- | ACK |
| SOH | 0x03 | 0xFC | Data[0-127] | Chksum | ---> | Packet OK |
| | | | | | <--- | ACK |
| EOT | | | | | ---> | Packet OK |
| ACK get garbaged | | | | | <--- | ACK |
| EOT | | | | | ---> | Packet OK |
| Finished | | | | | <--- | ACK |

- If the last packet contains less than 128-byte data with the specification of [SOH 04 0xFB Data[100] CPMEOF[28] CRC CRC], write 0x1A (^Z) to fill the rest space of CPMEOF[28].

Note: Please refer to Appendix 1 or "XModem Protocol" for the details.

## 1.4 System Component

As for this system, on the one hand, its main controller SPMC752313A should communicate with PC via an RS232 voltage converter (performed via TXD2 and RXD2); on the other hand, it needs to extend its memory via an SRAM. Thus, interfaced with HM62864A SRAM, IOB [15:0] and IOD[7:0] are occupied as the address bus to A[15:0] and I/O[7:0] and IOD[10:8] are occupied as the control bus to CS1, OE and WE. The hardware connection is shown as Figure 1-3.

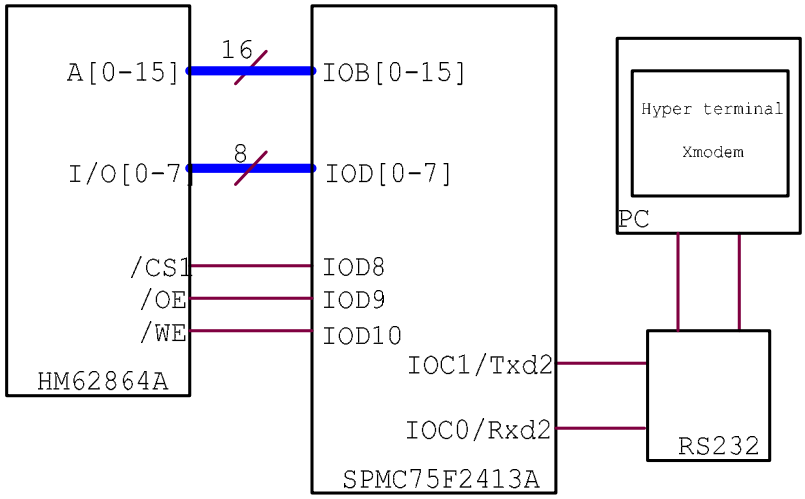


Figure 1-3 System Diagram

SRAM HM62864A 64K is used to store the mass data received. These data can be read to check whether they are consistent with the data sent.

# 2 Software Design

## 2.1 Software Description

The application program is mainly designed to upload files from PC. The software performs the initialization of the related hardware including UART and CMT0, and loads the received data into SRAM and the operations to the SRAM. In this design, these data is simply used for checking and you can process them freely.

## 2.2 Source File

| File Name | Function | Type |
|---|---|---|
| Main.C | Initialization, files receiving and data checking | C |
| XModem .C | XModem procedure | C |
| UART.C | UART and CMT0 initialization | C |
| CRC16.Asm | CRC16 and Checksum calculating | ASM |
| HM62864.C | SRAM operation | C |
| ISR.C | Interrupt service routine | C |
| AN_SPMC75_0109.H | API calling and related arguments definition | H |
| HM62864.H | GPIO driver for HM62864 operation | H |

## 2.3 Subroutines

**Spmc75_XModem Initial()**

| | |
|---|---|
| **Prototype** | void Spmc75_XModem Initial(void) |
| **Description** | UART and CMT0 initialization and flag reset |
| **Input Arguments** | None |
| **Output Arguments** | None |
| **Head Files** | AN_SPMC75_0109.H |
| **Library Files** | None |
| **Note** | Initialize UART Baud rate=15200bps, use UART2, set CMT0 as 8Hz interrupt |
| **Example** | Spmc75_XModem Initial();//Initial XModem hardware. |

**Spmc75_XModem Receive()**

| | |
|---|---|
| **Prototype** | void Spmc75_XModem Receive(void) |
| **Description** | Send a receiving request through XModem protocol, receive files. |

| | |
|---|---|
| **Input Arguments** | None |
| **Output Arguments** | None |
| **Head Files** | AN_SPMC75_0109.H |
| **Library Files** | None |
| **Note** | Wait until file transmission finishes.<br>During data receiving, apply for a break interrupt after receiving a correct packet to process data.<br>Set a flag (xmodemstatus.B._recvrdy) after the whole file is received. |
| **Example** | Spmc75_XModem Receive(); //Receive file transfer |

## XModem _Rxd_ISR( )

| | |
|---|---|
| **Prototype** | void XModem _Rxd_ISR(void) |
| **Description** | UART Rxd interrupt service function |
| **Input Arguments** | None |
| **Output Arguments** | None |
| **Head Files** | AN_SPMC75_0109.H |
| **Library Files** | None |
| **Note** | Used for receiving data in UART Rxd ISR |
| **Example** | void IRQ6(void) __attribute__ ((ISR));<br>void IRQ6(void)<br>{<br>if(P_UART_Status->B.RXIF)<br>{<br>XModem _Rxd_ISR(); //XModem Rxd ISR.<br>}<br>} |

## Spmc75_TimeOut_ISR( )

| | |
|---|---|
| **Prototype** | void Spmc75_TimeOut_ISR(void) |
| **Description** | Time out interrupts service function periodically. |
| **Input Arguments** | None |
| **Output Arguments** | None |
| **Head Files** | AN_SPMC75_0109.H |
| **Library Files** | None |
| **Note** | CMT0 interrupt service routine, CMT0 interrupt frequency is 8Hz. |
| **Example** | void IRQ7(void) __attribute__ ((ISR));<br>void IRQ7(void)<br>{ |

```
        if(P_INT_Status->B.CMTIF)
        {
            if(P_CMT_Ctrl->B.CM0IF && P_CMT_Ctrl->B.CM0IE)
            {
                Spmc75_TimeOut_ISR();  //8Hz ISR for timout.
            }
        }
    }
```

# 3 Design Tips

## 3.1 Demo Listing

The example program is to receive the file and load it into SRAM (see Figure 1-3). First, SPMC75F2314A receives the data from PC and loads it into SRAM. Second, we can check the data stored, in other words, we can read the received data to see whether it accords with that from PC.

```
#include  "AN_SPMC75_0109.H"
//=====================================================================
unsigned char Buffer[128];
externUInt16 Intputaddr;
UInt16 Outputaddr = 0;

main()
{
  UInt16  i;

  P_IOA_SPE->W = 0x00;
  P_IOB_SPE->W = 0x00;
  P_IOC_SPE->W = 0x00;
  SRAM_Initial();
  IRQ_ON();
  while(1)
  {
      Spmc75_XModem Initial();               //Initial XModem hardware.
      Spmc75_XModem Receive();               //Receive file Transfer

      if(xmodemstatus.B._recvrdy)         //Recv ready?
      {
          xmodemstatus.B._recvrdy = 0;
          while(Outputaddr != Intputaddr) //Read file from SRAM
          {// Read 128-byte each time and check it yourself
           // Be sure to affirm SRAM operation
              for(i=0;i<128 && Outputaddr != Intputaddr;i++)
                  Buffer[i] = SRAM_ReadAByte(++Outputaddr);
          }
      }
  }
}
//=====================================================================
// Description: BREAK interrupt is used to XXX
// Notes: Get Data.
//=====================================================================
extern unsigned char rxdbuf[133];   // Defined in XModem.c
                                    //Buffer for receiving data ()
UInt16 Intputaddr = 0;
void BREAK(void) __attribute__ ((ISR));
void BREAK(void)
{
```

```
  UInt16  i;
  for(i=3;i<131;i++)
  {
      SRAM_WriteAByte(++Intputaddr, rxdbuf[i]); //Data loaded into SPAM
  }
}
//=====================================================================
// Description: IRQ6 interrupt source is XXX,used to XXX
// Notes:
//=====================================================================
void IRQ6(void) __attribute__ ((ISR));
void IRQ6(void)
{
  if(P_INT_Status->B.UARTIF)
  {
      if(P_UART_Status->B.TXIF && P_UART_Ctrl->B.TXIE){;}
      if(P_UART_Status->B.RXIF)
      {
          XModem _Rxd_ISR();                    //XModem Rxd ISR.
      }
  }
}
//=====================================================================
// Description: IRQ7 interrupt source is XXX,used to XXX
// Notes:
//=====================================================================
void IRQ7(void) __attribute__ ((ISR));
void IRQ7(void)
{
  if(P_INT_Status->B.CMTIF)
  {
      if(P_CMT_Ctrl->B.CM0IF && P_CMT_Ctrl->B.CM0IE)
      {
          Spmc75_TimeOut_ISR();                 //8Hz ISR for timout.
      }
      P_CMT_Ctrl->W = P_CMT_Ctrl->W;
  }
}
```

## 3.2 File Transfer

XModem file is transferred via Console port. The following operations are the demo under Window2000.

- Hyper terminal (Figure 3-1).

[Start]->[Programs]->[Accesories]->[Communications]->[Hyper Terminal]


Figure 3-1 Hyper Terminal

- New connection (Figure 3-2).

Input [Name], select an icon…, click [OK].

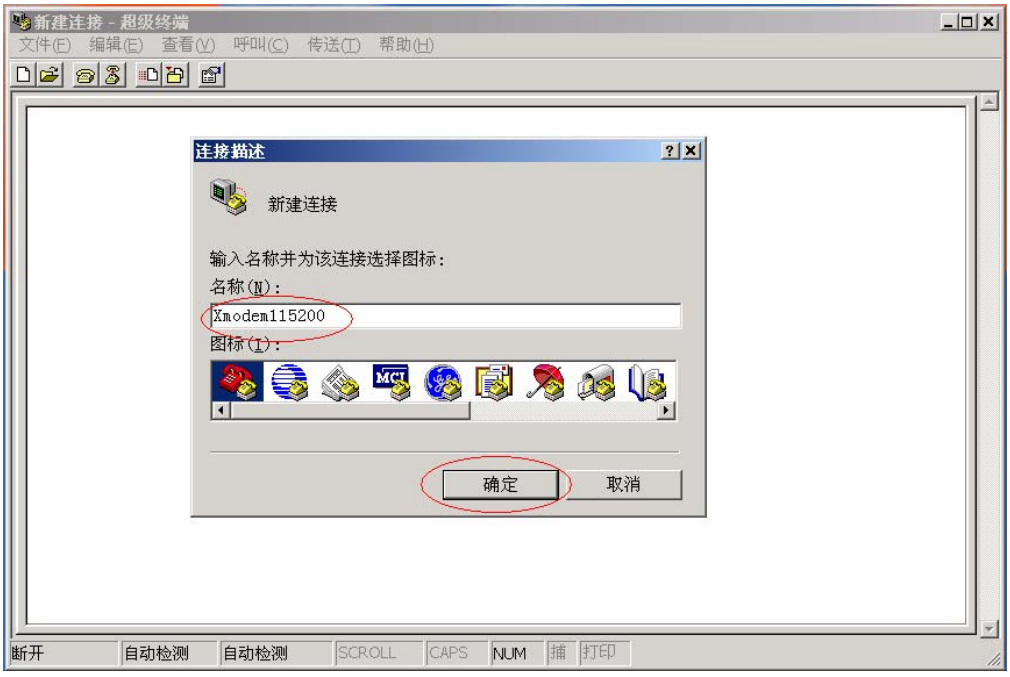


Figure 3-2 New Connection

- Connect to COM1 (Figure 3-3).

Select the COM port you connect to serial line in [Connect Using], click [OK].

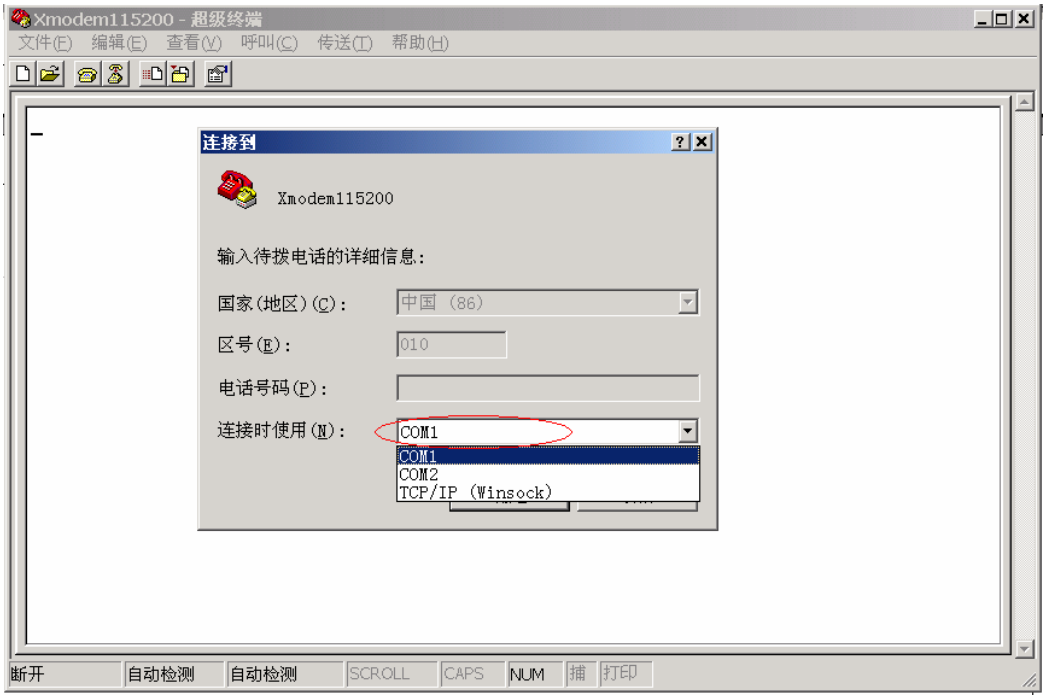


Figure 3-3 Connect To

- COM1 properties (Figure 3-4).

Select baud rate as **115200**bps in [Bits per second], **none** in [Flow control], others defaults, click [OK].

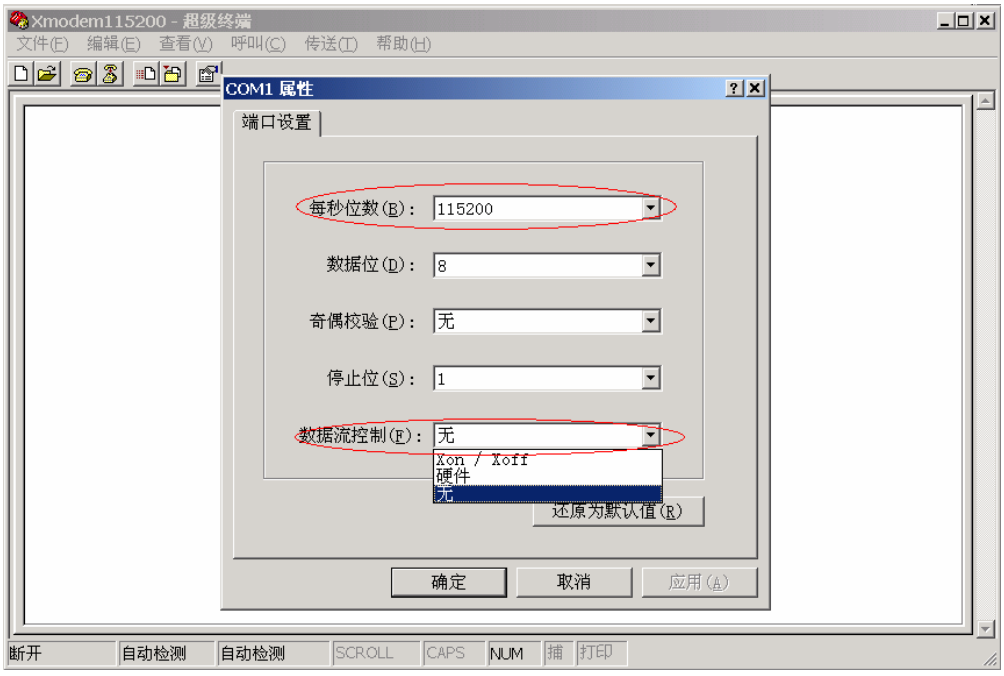


Figure 3-4 COM Properties

- Buttons (Figure 3-5).

Three buttons are useful here, **call**, **disconnect** and **send**.


Figure 3-5 Buttons

- Protocols selection (Figure 3-6).

Click **send**, navigate the file to send in [Filename], select XModem in [Protocol].

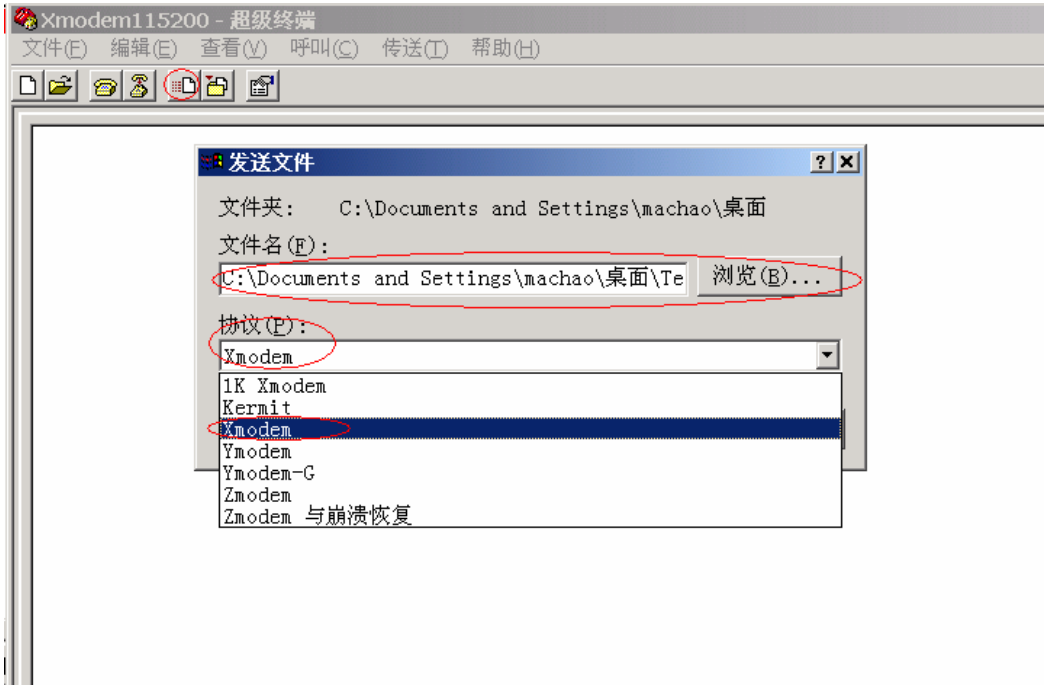


Figure 3-6 Protocol Selection

- Send file (Figure 3-7).

After the setting listed above, click [Send] to start the file transfer and pop up a watch window as Figure 3-7.

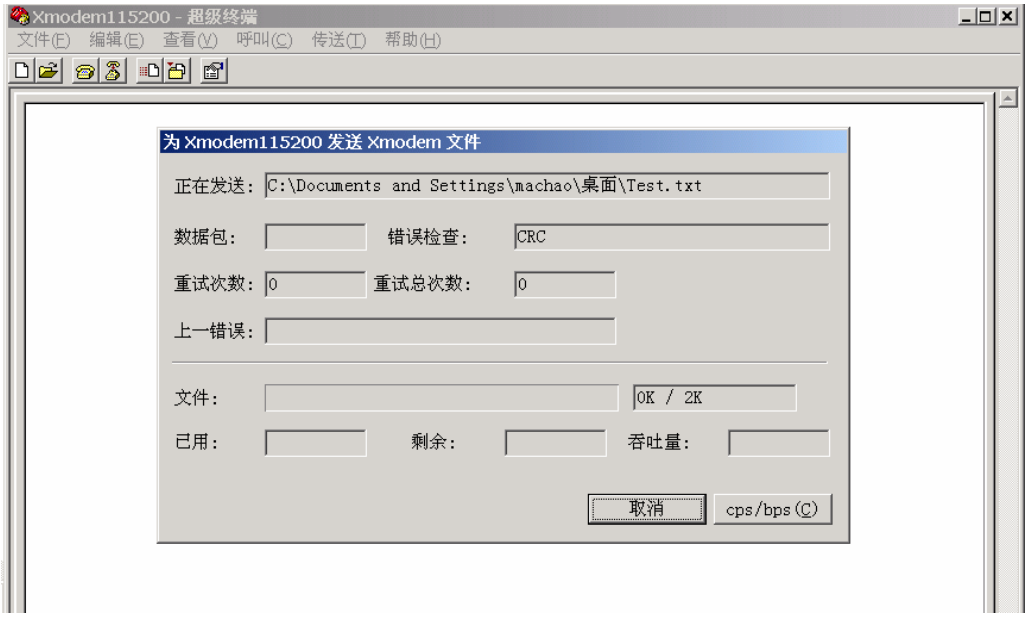


Figure 3-7 Send File

- Transmit information (Figure 3-8).

Information on file, packet, check method, error checking, retries, total retries, respond, file size, elapse etc..

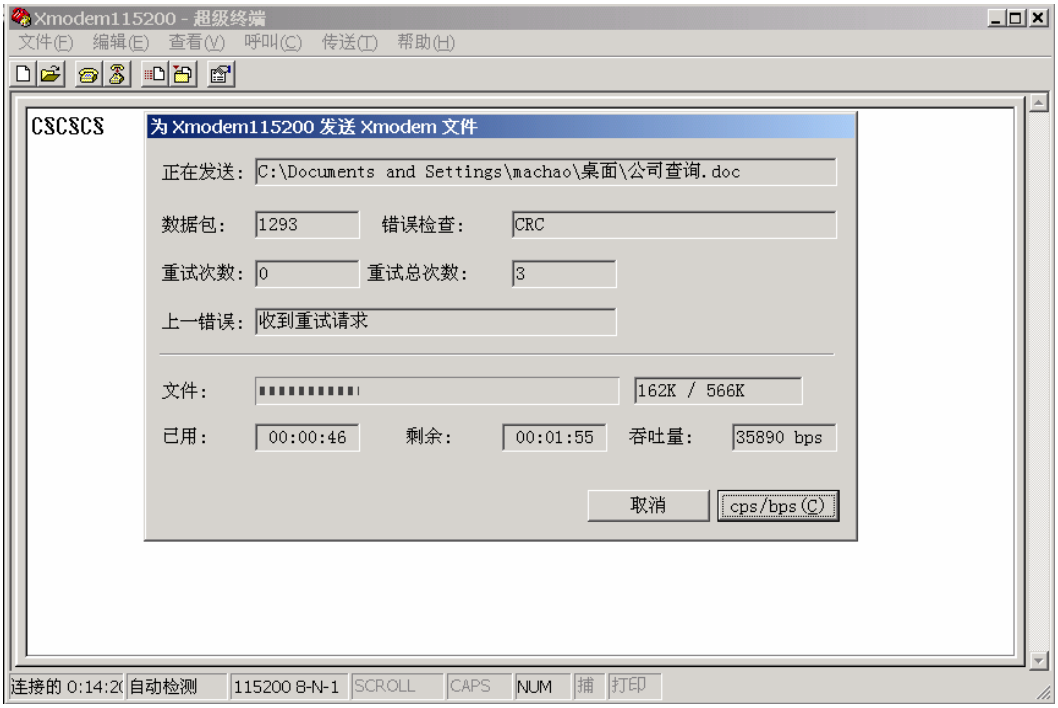


Figure 3-8 Transmit Information

# 4 MCU Resource

| CPU | SPMC75F2413A | Package | QFP80-1.0 |
|---|---|---|---|
| Oscillator | ☒ crystal | Frequency | 6MHz |
| | ☐External | Input Frequency | |
| WATCHDOG | ☐ Enable ☒ Disable | | |
| IO Occupation | IOB[0-15] | SRAM HM62864A address control | |
| | IOC[0-1] | UART serial communication interface | |
| | IOD[0-7] | Data communication interface connected with SRAM HM62864A | |
| | IOC[8-10] | Connect to SRAM HM62864A control pins | |
| | The rest IO pins | Main control/GND | |
| Timer | CMT0 | System clock | |
| Interrupt Occupation | UART(IRQ6) | UART int for receiving data | |
| | CMT0(IRQ7) | System interrupt | |
| ROM Occupation | 1.78K Words | | |

# 5 [Appendix 1] XModem Protocol

**Table of Contents:**



```
1. DEFINITIONS.
  <soh> 01H   <eot> 04H   <ack> 06H   <nak> 15H   <can> 18H

2. TRANSMISSION MEDIUM LEVEL PROTOCOL
Asynchronous, 8 data bits, no parity, one stop bit.
   The protocol imposes no restrictions on the contents of the data being
transmitted. No control characters are looked for in the 128-byte data messages.
  Absolutely any kind of data may be sent - binary, ASCII, etc.  The protocol
has not formally been adopted to a 7-bit environment for the transmission of
ASCII-only (or unpacked-hex) data , although it could be simply by having both
ends agree to AND the protocol-dependent data with 7F hex before validating
it.  I specifically am referring to the checksum, and the block numbers and
their ones-complement.
   Those wishing to maintain compatibility of the CP/M file structure, i.e.
to allow modemming ASCII files to or from CP/M systems should follow this data
format:
  * ASCII tabs used (09H); tabs set every 8.
  * Lines terminated by CR/LF (0DH 0AH)
  * End-of-file indicated by ^Z, 1AH.  (one or more)
  * Data is variable length, i.e. should be considered a
    continuous stream of data bytes, broken into 128-byte
    chunks purely for the purpose of transmission.
  * A CP/M "peculiarity": If the data ends exactly on a
    128-byte boundary, i.e. CR in 127, and LF in 128, a
    subsequent sector containing the ^Z EOF character(s)
    is optional, but is preferred.  Some utilities or
    user programs still do not handle EOF without ^Zs.
  * The last block sent is no different from others, i.e.
there is no "short block".

3. MESSAGE BLOCK LEVEL PROTOCOL
  Each block of the transfer looks like:
<SOH><blk #><255-blk #><--128 data bytes--><cksum>
   in which:
<SOH>    = 01 hex
<blk #>   = binary number, starts at 01 increments by 1, and
       wraps 0FFH to 00H (not to 01)
<255-blk #> = blk # after going thru 8080 "CMA" instr, i.e.
       each bit complemented in the 8-bit block number.
```

```
       Formally, this is the "ones complement".
<cksum>  = the sum of the data bytes only.  Toss any carry.
```

**4. FILE LEVEL PROTOCOL**

```
--4A. COMMON TO BOTH SENDER AND RECEIVER:
   All errors are retried 10 times.
   Some versions of the protocol use <can>, ASCII ^X, to cancel transmission.
This was never adopted as a standard, as having a single "abort" character makes
the transmission susceptible to false termination due to an <ack> <nak> or <soh>
being corrupted into a <can> and cancelling transmission.
   The protocol may be considered "receiver driven", that is, the sender need
not automatically re-transmit, although it does in the current implementations.

--4B. RECEIVE PROGRAM CONSIDERATIONS:
   The receiver has a 10-second timeout.  It sends a <nak> every time it times
out.  The receiver's first timeout, which sends a <nak>, signals the transmitter
to start.  Optionally, the receiver could send a <nak> immediately, in case
the sender was ready.  This would save the initial 10-second timeout. However,
the receiver MUST continue to timeout every 10 seconds in case the sender wasn't
ready.
   Once into a receiving a block, the receiver goes into a one-second timeout
for each character and the checksum.  If the receiver wishes to <nak> a block
for any reason (invalid header, timeout receiving data), it must wait for the
line to clear.See "programming tips" for ideas Synchronizing:  If a valid block
number is received, it will be: 1) the expected one, in which case everything
is fine; or 2) a repeat of the previously received block.  This should be considered
OK, and only indicates that the receivers <ack> got glitched, and the sender
re-transmitted; 3) any other block number indicates a fatal loss of
synchronization, such as the rare case of the sender getting a line-glitch that
looked like an <ack>.  Abort the transmission, sending a <can>

--4C. SENDING PROGRAM CONSIDERATIONS.
   While waiting for transmission to begin, the sender has only a single very
long timeout, say one minute. In the current protocol, the sender has a 10 second
timeout before retrying.  I suggest NOT doing this, and letting the protocol
be completely receiver-driven.    This will be compatible with existing
programs. When the sender has no more data, it sends an <eot>, and awaits an
<ack>, resending the <eot> if it doesn't get one. Again, the protocol could
be receiver-driven, with the sender only having the high-level 1-minute timeout
to abort.
```

**5. DATA FLOW EXAMPLE INCLUDING ERROR RECOVERY**

```
Here is a sample of the data flow, sending a 3-block message. It includes the
two most common line hits - a garbaged block, and an <ack> reply getting garbaged.
<xx> represents the checksum byte.

SENDER                RECEIVER
              times out after 10 seconds,
                           <---        <nak>
<soh> 01 FE -data- <xx>    --->
                           <---        <ack>
<soh> 02 FD -data- <xx>    --->    (data gets line hit)
                           <---        <nak>
<soh> 02 FD -data- <xx>    --->
                           <---        <ack>
```

```
<soh> 03 FC -data- <xx>   --->
   (ack gets garbaged)    <---        <ack>
<soh> 03 FC -data- <xx>   --->        <ack>
<eot>                     --->
                          <---        <ack>
```

**6. PROGRAMMING TIPS.**

*The character-receive subroutine should be called with a parameter specifying the number of seconds to wait. The receiver should first call it with a time of 10, then <nak> and try again, 10 times.

After receiving the <soh>, the receiver should call the character receive subroutine with a 1-second timeout, for the remainder of the message and the <cksum>. Since they are sent as a continuous stream, timing out of this implies a serious like glitch that caused, say, 127 characters to be seen instead of 128.

*When the receiver wishes to <nak>, it should call a "PURGE" subroutine, to wait for the line to clear. Recall the sender tosses any characters in its UART buffer immediately upon completing sending a block, to ensure no glitches were mis- interpreted.

The most common technique is for "PURGE" to call the character receive subroutine, specifying a 1-second timeout, and looping back to PURGE until a timeout occurs. The <nak> is then sent, ensuring the other end will see it.

*You may wish to add code recommended by John Mahr to your character receive routine - to set an error flag if the UART shows framing error, or overrun. This will help catch a few more glitches - the most common of which is a hit in the high bits of the byte in two consecutive bytes. The <cksum> comes out OK since counting in 1-byte produces the same result of adding 80H + 80H as with adding 00H + 00H.

```
>----------------------------------------------------------------<
MODEM PROTOCOL OVERVIEW,  CRC OPTION ADDENDUM
Last Rev: (preliminary  1/13/85)
This document describes the changes to the Christensen Modem Protocol that
implement the CRC option. This document is an addendum to Ward Christensen's
"Modem Protocol Overview". This document and Ward's document are both required
for a complete description of the Modem Protocol.
```

**--A. Table of Contents**



```
--B. ADDITIONAL DEFINITIONS
<C>   43H
```

**7. OVERVIEW OF CRC OPTION**

The CRC used in the Modem Protocol is an alternate form of block check which provides more robust error detection than the original checksum. Andrew S.

Tanenbaum says in his book, Computer Networks, that the CRC-CCITT used by the Modem Protocol will detect all single and double bit errors, all errors with an odd number of bits, all burst errors of length 16 or less, 99.997% of 17-bit error bursts, and 99.998% of 18-bit and longer bursts.
The changes to the Modem Protocol to replace the checksum with the CRC are straight for ward. If that were all that we did we would not be able to communicate between a program using the old checksum protocol and one using the new CRC protocol. An initial handshake was added to solve this problem. The handshake allows a receiving program with CRC capability to determine whether the sending program supports the CRC option, and to switch it to CRC mode if it does. This handshake is designed so that it will work properly with programs which implement only the original protocol. A description of this handshake is presented in section 10.

**8. MESSAGE BLOCK LEVEL PROTOCOL, CRC MODE**

Each block of the transfer in CRC mode looks like:

**<SOH><blk #><255-blk #><--128 data bytes--><CRC hi><CRC lo>**

in which:

```
<SOH>     = 01 hex
<blk #>   = binary number, starts at 01 increments by 1, and
              wraps 0FFH to 00H (not to 01)
<255-blk #> = ones complement of blk #.
<CRC hi>   = byte containing the 8 hi order coefficients of the CRC.
<CRC lo>   = byte containing the 8 lo order coefficients of the CRC.
              See the next section for CRC calculation.
```

**9. CRC CALCULATION**

--9A. FORMAL DEFINITION OF THE CRC CALCULATION

To calculate the 16 bit CRC the message bits are considered to be the coefficients of a polynomial. This message polynomial is first multiplied by X^16 and then divided by the generator polynomial (X^16 + X^12 + X^5 + 1) using modulo two arithemetic. The remainder left after the division is the desired CRC. Since a message block in the Modem Protocol is 128 bytes or 1024 bits, the message polynomial will be of order X^1023. The hi order bit of the first byte of the message block is the coefficient of X^1023 in the message polynomial. The lo order bit of the last byte of the message block is the coefficient of X^0 in the message polynomial.

--9B. EXAMPLE OF CRC CALCULATION WRITTEN IN C

```
/*
This function calculates the CRC used by the "Modem Protocol" The first argument
is a pointer to the message block. The second argument is the number of bytes
in the message block. The message block used by the Modem Protocol contains
128 bytes. The function return value is an integer which contains the CRC. The
lo order 16 bits of this integer are the coefficients of the CRC. The The lo
order bit is the lo order coefficient of the CRC.
*/
int calcrc(ptr, count) char *ptr; int count;
{
   int crc = 0, i;

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

```
      else crc = crc << 1;
  }
   return (crc & 0xFFFF);
   }
```

**10. FILE LEVEL PROTOCOL, CHANGES FOR COMPATIBILITY**

--10A. COMMON TO BOTH SENDER AND RECEIVER:

The only change to the File Level Protocol for the CRC option is the initial handshake which is used to determine if both the sending and the receiving programs support the CRC mode. All Modem Programs should support the checksum mode for compatibility with older versions. A receiving program that wishes to receive in CRC mode implements the mode setting handshake by sending a <C> in place of the initial <nak>. If the sending program supports CRC mode it will recognize the <C> and will set itself into CRC mode, and respond by sending the first block as if a <nak> had been received. If the sending program does not support CRC mode it will not respond to the <C> at all. After the receiver has sent the <C> it will wait up to 3-seconds for the <soh> that starts the first block. If it receives a <soh> within 3-seconds it will assume the sender supports CRC mode and will proceed with the file exchange in CRC mode. If no <soh> is received within 3-seconds the receiver will switch to checksum mode, send a <nak>, and proceed in checksum mode. If the receiver wishes to use checksum mode it should send an initial <nak> and the sending program should respond to the <nak> as defined in the original Modem Protocol. After the mode has been set by the initial <C> or <nak> the protocol follows the original Modem Protocol and is identical whether the checksum or CRC is being used.

--10B. RECEIVE PROGRAM CONSIDERATIONS:

There are at least 4 things that can go wrong with the mode setting handshake.

1. the initial <C> can be garbled or lost.
2. the initial <soh> can be garbled.
3. the initial <C> can be changed to a <nak>.
4. the initial <nak> from a receiver which wants to receive in Checksum can be changed to a <C>. The first problem can be solved if the receiver sends a second <C> after it times out the first time. This process can be repeated several times. It must not be repeated a too many times before sending a <nak> and switching to checksum mode or a sending program without CRC support may time out and abort. Repeating the <C> will also fix the second problem if the sending program cooperates by responding as if a <nak> were received instead of ignoring the extra <C>.

It is possible to fix problems 3 and 4 but probably not worth the trouble since they will occur very infrequently. They could be fixed by switching modes in either the sending or the receiving program after a large number of successive <nak>s. This solution would risk other problems however.

--10C. SENDING PROGRAM CONSIDERATIONS.

The sending program should start in the checksum mode. This will insure compatibility with checksum only receiving programs. Anytime a <C> is received before the first <nak> or <ack> the sending program should set itself into CRC mode and respond as if a <nak> were received. The sender should respond to additional <C>s as if they were <nak>s until the first <ack> is received. This will assist the receiving program in determining the correct mode when the <soh> is lost or garbled. After the first <ack> is received the sending program should ignore <C>s.

**11. DATA FLOW EXAMPLES WITH CRC OPTION**

--11A. RECEIVER HAS CRC OPTION, SENDER DOESN'T

Here is a data flow example for the case where the receiver requests transmission in the CRC mode but the sender does not support the CRC option. This example also includes various transmission errors. <xx> represents the checksum byte.

```
SENDER                RECEIVER
                          <---        <C>
             times out after 3 seconds,
                          <---        <nak>
<soh> 01 FE -data- <xx> --->
                          <---        <ack>
<soh> 02 FD -data- <xx> --->  (data gets line hit)
                          <---        <nak>
<soh> 02 FD -data- <xx> --->
                          <---        <ack>
<soh> 03 FC -data- <xx> --->
   (ack gets garbaged)  <---        <ack>
             times out after 10 seconds,
                          <---        <nak>
<soh> 03 FC -data- <xx> --->
                          <---        <ack>
<eot>                     --->
                          <---        <ack>
```

--11B. RECEIVER AND SENDER BOTH HAVE CRC OPTION

Here is a data flow example for the case where the receiver requests transmission in the CRC mode and the sender supports the CRC option. This example also includes various transmission errors.

<xxxx> represents the 2 CRC bytes.

```
SENDER                  RECEIVER
                          <---          <C>
<soh> 01 FE -data- <xxxx> --->
                          <---          <ack>
<soh> 02 FD -data- <xxxx> --->      (data gets line hit)
                          <---          <nak>
<soh> 02 FD -data- <xxxx> --->
                          <---          <ack>
<soh> 03 FC -data- <xxxx> --->
   (ack gets garbaged)    <---        <ack>
                times out after 10 seconds,
                          <---          <nak>
<soh> 03 FC -data- <xxxx> --->
                          <---          <ack>
<eot>                     --->
                          <---          <ack>
```

# 6 Reference

[1] SUNPLUS, *SPMC75F2413A Programming Guide V1.0*, Oct 12, 2004.

[2] *XModem Protocol*