# 用 UART 做文件传输（采用 Xmodem 协议）

***V1.1 - Dec 8, 2005***

***中文版***

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.cn

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

**版权声明**

凌阳科技股份有限公司保留对此文件修改之权利且不另行通知。凌阳科技股份有限公司所提供之信息相信为正确且可靠之信息，但并不保证本文件中绝无错误。请于向凌阳科技股份有限公司提出订单前，自行确定所使用之相关技术文件及规格为最新之版本。若因贵公司使用本公司之文件或产品，而涉及第三人之专利或著作权等智能财产权之应用及配合时，则应由贵公司负责取得同意及授权，本公司仅单纯贩售产品，上述关于同意及授权，非属本公司应为保证之责任。又未经凌阳科技股份有限公司之正式书面许可，本公司之所有产品不得使用于医疗器材，维持生命系统及飞航等相关设备。

# 目录

# 修订记录

| 版本 | 日期 | 编写及修订者 | 编写及修订说明 |
|---|---|---|---|
| 1.0 | 2004/01/13 | | 初版 |
| 1.1 | 2005/12/08 | | 错误校正 |

# 1 系统概要

## 1.1 系统说明

本应用例使用 SPMC75F2413A 的 UART 完成文件的 Upload（PC->SPMC75F2413A）,通过 Xmodem 文件传输协议的支持达到很好的传输效果。从而为大量数据的传输及驱动程序的重载提供了一个可行的实施方法。

## 1.2 Xmodem 简介

FTP 即 File Transfer Protocol 的缩写，串行通信的文件传输协议主要有：Xmodem、Ymodem、Zmodem 和 KERMIT 等。

Xmodem 协议一般支持 128 字节的数据包，并且支持一般校验和、CRC 两种校验方式，在出现数据包错误的情况下支持多次重传（一般为 10 次）。Xmodem 协议传输由接收程序和发送程序完成。先由接收程序发送协商字符，协商校验方式，协商通过之后发送程序就开始发送数据包，接收程序接收到完整的一个数据包之后按照协商的方式对数据包进行校验。校验通过之后发送确认字符，然后发送程序继续发送下一包；如果校验失败，则发送否认字符，发送程序重传此数据包。

## 1.3 Xmodem 协议

### 1.3.1 相关说明

1、定义：<SOH> 01H、<EOT> 04H、<ACK> 06H、<NAK> 15H、<CAN> 18H。
2、UART 格式：Asynchronous、8 data—bits、no parity、one stop—bit。

### 1.3.2 协议简介

Xmodem 协议是由 Ward Chritensen 于 70 年代提出并实现的，传输数据单位为信息包，包含一个标题开始字符<SOH>，一个单字节包序号，一个包序号的补码，128 个字节数据和一个单字节的校验和。它把数据划分成 128 个字符的小包进行发送，每发送一个小包都要检查是否正确，如果信息包正确接收方发送一个字节<ACK>的应答；有错重发则发送一个字节<NAK>应答，要求重发。因此 Xmodem 是一种发送等待协议，具有流量控制功能。优点：简单通用，几乎所有通信软件都支持该协议。 缺点：慢。检验和信息包格式如图 1—1 所示。

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

图 1—1 检验和信息包格式

Xmodem 协议的数据包格式在 90 年代经过一次修改，传输数据单位仍为信息包，包含一个标题开始字符 SOH，一个单字节包序号，一个包序号的补码，128 个字节数据和一个双字节的 CRC16 校验。所以新的协议格式信息包如图 1－2 所示。

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

图 1－2 CRC 校验信息包格式

### 1.3.3 校验和信息包

1、校验和信息包

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

其中：

<SOH>　　= 01 hex

<blk #>　　= 信息包序号，从 01 开始以发送一包将加 1，加到 FF hex 将循环。

<255-blk #>　= 信息包序号的补码。

<cksum>　　= 保留字节，丢掉进位的和校验。

2、校验和方式数据传输流程

接收方要求发送方以校验和方式发送时以 NAK 来请求，发送方将对此做出应答。如表 1－1 所示传输 5 包数据的示意过程。

表 1－1 校验和数据传输过程

| Sender | | | | | Flow | Receiver |
|---|---|---|---|---|---|---|
| | | | | | <--- | NAK |
| | | | | | | Time out after 3 Second |
| | | | | | <--- | 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 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 |

| Sender | | | | | Flow | Receiver |
|---|---|---|---|---|---|---|
| | | | | | <--- | 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 校验信息包

1、CRC 校验信息包

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

其中：

<SOH> = 01 hex

<blk #> = 信息包序号，从 01 开始以发送一包将加 1，加到 FF hex 将循环。

<255-blk #> = 信息包序号的补码。

<CRC hi> = CRC16 高字节。

<CRC lo> = CRC16 低字节。

2、CRC 描述

计算 16－Bit CRC 校验的除数多项式为 X^16 + X^12 + X^5 + 1，信息包中的 128 数据字节将参加 CRC 校验的计算。在发送端 CRC16 的高字节在先，低字节在后。

3、CRC 校验方式数据传输流程

接收方要求发送方以 CRC 校验方式发送时以‘C’来请求，发送方将对此做出应答。如表 1－2 所示传输 3 包数据的示意过程。

表 1－2 CRC 校验数据传输过程

| Sender | | | | | Flow | Receiver |
|---|---|---|---|---|---|---|
| | | | | | <--- | ‘C’ |
| | | | | | | Time out after 3 Second |
| | | | | | <--- | ‘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 |

| Sender | Flow | Receiver |
|---|---|---|
| Finished | <--- | ACK |

4、在发送方仅仅支持校验和的传输方式时，就应对其请求 NAK，要求以 CheckSum 的校验方式来发送数据，如表 1－1 所示过程。如果发送方仅仅支持 CRC 校验的传输方式，应以'C'来请求发送，如表 1－2 所示过程。如果两者都支持的话，将优先以'C'来请求发送。所以接收程序的实现过程将如表 1－3 所示。

表 1－2 CRC 校验数据传输过程

| Sender | | | | | Flow | Receiver |
|---|---|---|---|---|---|---|
| | | | | | <--- | 'C' |
| | | | | | | Time out after 3 Second |
| | | | | | <--- | NAK |
| | | | | | | Time out after 3 Second |
| | | | | | <--- | 'C' |
| | | | | | | Time out after 3 Second |
| | | | | | <--- | 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 |

5、信息包中如果剩余的数据不足 128－Byte 信息包的格式为：【SOH 04 0xFB Data[100] CPMEOF[28] CRC CRC】，不足的 CPMEOF[28]将以 0x1A（^Z）填充。

【注】：关于 Xmodem 协议的细节请参看附录 1 或《Xmodem 协议》。

## 1.4 系统组成

系统包括和 PC 相连的 RS232 电平转换接口，和数据存储的 SRAM 接口，如图 1－3 所示。其中:IOB[0－15]连接到 SRAM HM62864A 的地址线 A[0－15]，IOD[0－7]连接到 SRAM HM62864A 的地址线 I/O[0－7]，IOD8、IOD9 和 IOD10 分别]连接到 SRAM HM62864A 的 CS1、OE 和 WE。Txd2 和 Rxd2 则通过电平转换后通过串口线连接到 PC 串口的 Rxd 和 Txd。

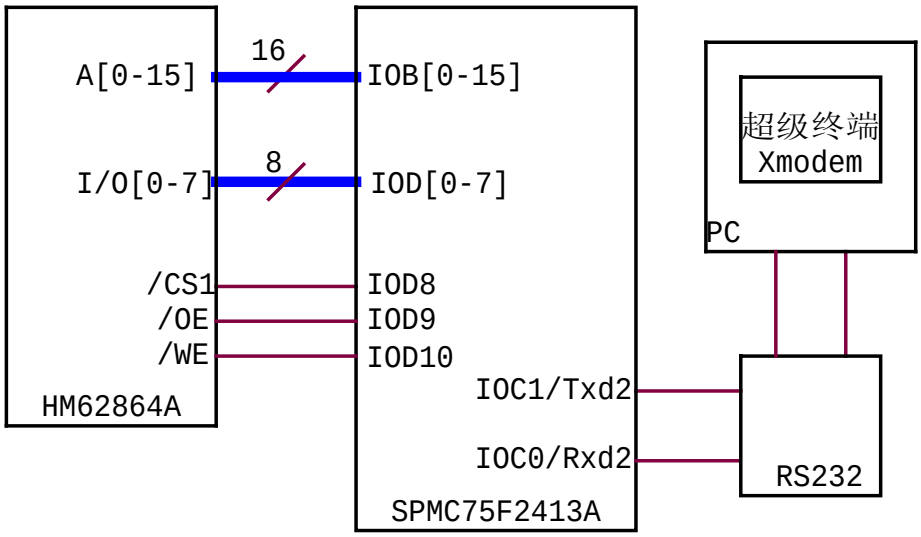



图 1－3 系统框图

SRAM HM62864A 64K 的数据存储空间是用来存储接收来的大量数据，当然如果要验证所接收的数据和发送的数据是否一致，可以读取来查看。

# 2 软件说明

## 2.1 软件说明

应用例程序部分主要完成 Xmodem 协议 Upload 文件，即接收来自 PC 端的文件。应用例软件部分要完成相关硬件资源的初始化，包括 UART 和 CMT0 的初始化。在范例中将把接收来的数据缓存到 SRAM 中，所以还有对 HM62864A SRAM 的操作部分。当然接收来的数据可以很灵活的来处理，这里只是为了方便对数据的校对。

## 2.2 档案构成

| 文件名称 | 功能 | 类型 |
|---|---|---|
| Main.C | 初始化、接收文件及校对数据 | C |
| Xmodem.C | Xmodem 协议的实现 | C |
| UART.C | 对 UART 和 CMT0 的初始化 | C |
| CRC16.Asm | CRC16 和 Checksum 的计算 | ASM |
| HM62864.C | 对 SRAM 的操作 | C |
| ISR.C | 中断服务程序 | C |
| AN_SPMC75_0109.H | API 引用和相关参数定义 | H |
| HM62864.H | HM62864 操作中相关 GPIO 驱动的数据类型定义 | H |

## 2.3 子程序说明

### Spmc75_XmodemInitial()

**原　　形**　　void Spmc75_XmodemInitial(void)

**描　　述**　　UART 和 CMT0 初始化及标志位复位。

**输入参数**　　无

**输出参数**　　无

**头 文 件**　　AN_SPMC75_0109.H

**库 文 件**　　无

**注意事项**　　UART 波特率初始为 115200bps，UART 初始化使用 Channel2，CMT0 固定为 8Hz 中断

**例　　子**

```
Spmc75_XmodemInitial();       //Initial Xmodem hardware.
```

## Spmc75_XmodemReceive()

| | |
|---|---|
| 原　　形 | void Spmc75_XmodemReceive(void) |
| 描　　述 | 以 Xmodem 协议发送接收请求，接收文件。 |
| 输入参数 | 无 |
| 输出参数 | 无 |
| 头 文 件 | AN_SPMC75_0109.H |
| 库 文 件 | 无 |
| 注意事项 | 等待…… 直到文件传输完毕。<br>在接收过程中，接收到一个校验无误的信息包后将申请软中断(break Interrupt)。可以在软中断中对数据进行处理。<br>整个文件接收完后将有一个标志位（xmodemstatus.B._recvrdy）置位。 |
| 例　　子 | Spmc75_XmodemReceive();　　//Receive file Transfer |

## Xmodem_Rxd_ISR( )

| | |
|---|---|
| 原　　形 | void Xmodem_Rxd_ISR(void) |
| 描　　述 | UART Rxd 中断服务函数。 |
| 输入参数 | 无 |
| 输出参数 | 无 |
| 头 文 件 | AN_SPMC75_0109.H |
| 库 文 件 | 无 |
| 注意事项 | 在 UART Rxd ISR 中使用，用作接收数据。 |
| 例　　子 | 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( )

| | |
|---|---|
| 原　　形 | void Spmc75_TimeOut_ISR(void) |
| 描　　述 | Time out 定时中断服务函数。 |
| 输入参数 | 无 |
| 输出参数 | 无 |
| 头 文 件 | AN_SPMC75_0109.H |
| 库 文 件 | 无 |
| 注意事项 | CMT0 中断服务函数，CMT0 固定中断频率为 8Hz。 |
| 例　　子 | |

```
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.
       }
   }
}
```

# 3 程序范例

## 3.1 DEMO 程序

范例为接收文件并存入 SRAM。范例程序可以结合图 1－3 所示的原理框图来操作，首先是 SPMC75F2314A 接收来自 PC 的数据，并且存储到 SRAM 中。接收完毕可以对说存储的数据进行验证，即：从 SRAM 中读取接收来的数据来查看，人为的进行评估。

```
#include  "AN_SPMC75_0109.H"
//=====================================================================
unsigned char Buffer[128];
extern UInt16 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_XmodemInitial();        //Initial Xmodem hardware.
       Spmc75_XmodemReceive();        //Receive file Transfer

       if(xmodemstatus.B._recvrdy)    //Recv ready?
       {
           xmodemstatus.B._recvrdy = 0;
           while(Outputaddr != Intputaddr)    //Read file from SRAM
           {//每次读取 128－Byte，请在每次读取 128－Byte 后来人为验证数据的正确性。
            //尤其要保证 SRAM 的操作是无误的。
               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];  //接收数据的缓存器（Xmodem.C 中定义）
UInt16 Intputaddr = 0;
void BREAK(void) __attribute__ ((ISR));
void BREAK(void)
{
   UInt16  i;
   for(i=3;i<131;i++)
   {
       SRAM_WriteAByte(++Intputaddr, rxdbuf[i]);  //数据存入 SRAM
   }
}
```

```
//=====================================================================
// 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 文件传输

通过 Console 口完成文件的 Xmodem 传输。一下所介绍的将是在 Window2000 操作系统下所作的演示。

1、超级终端，如图 3－1 所示。

【开始】－>【程序】－>【附件】－>【通讯】－>【超级终端】


图 3－1 超级终端

2、新建连接，如图 3－2 所示。

在【名称】中输入名称，选择图标…… 单击【确定】。

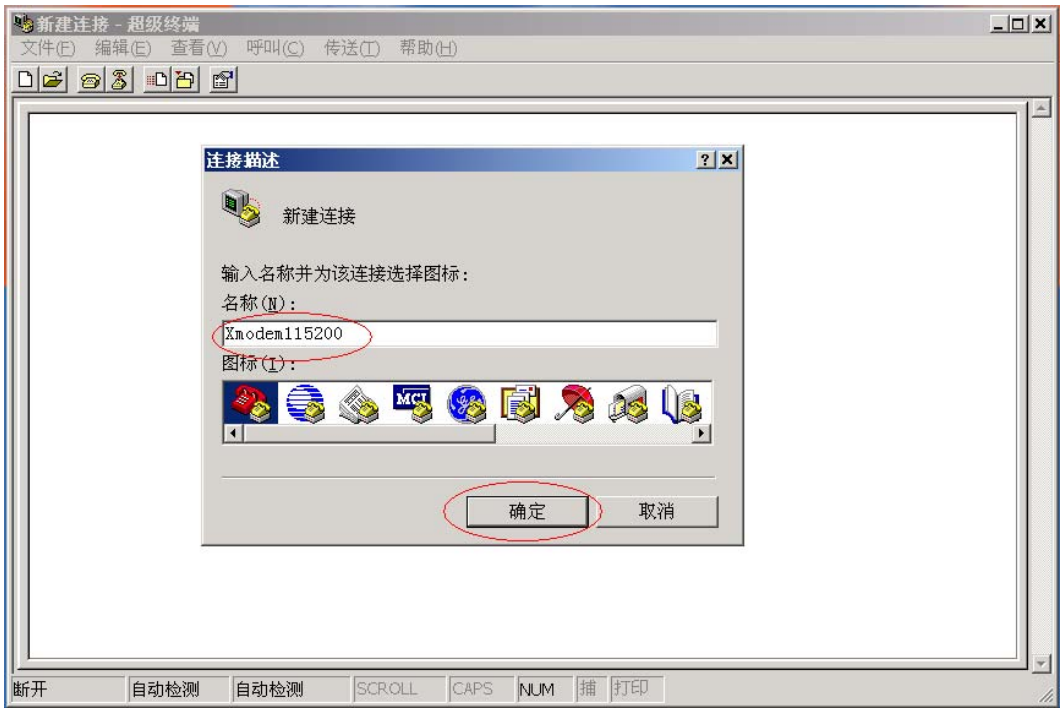



图 3－2 新建连接

3、连接到，如图 3－3 所示。

连接到，在【连接时使用】中选择串口线所连接的 COM 口，单击【确定】。

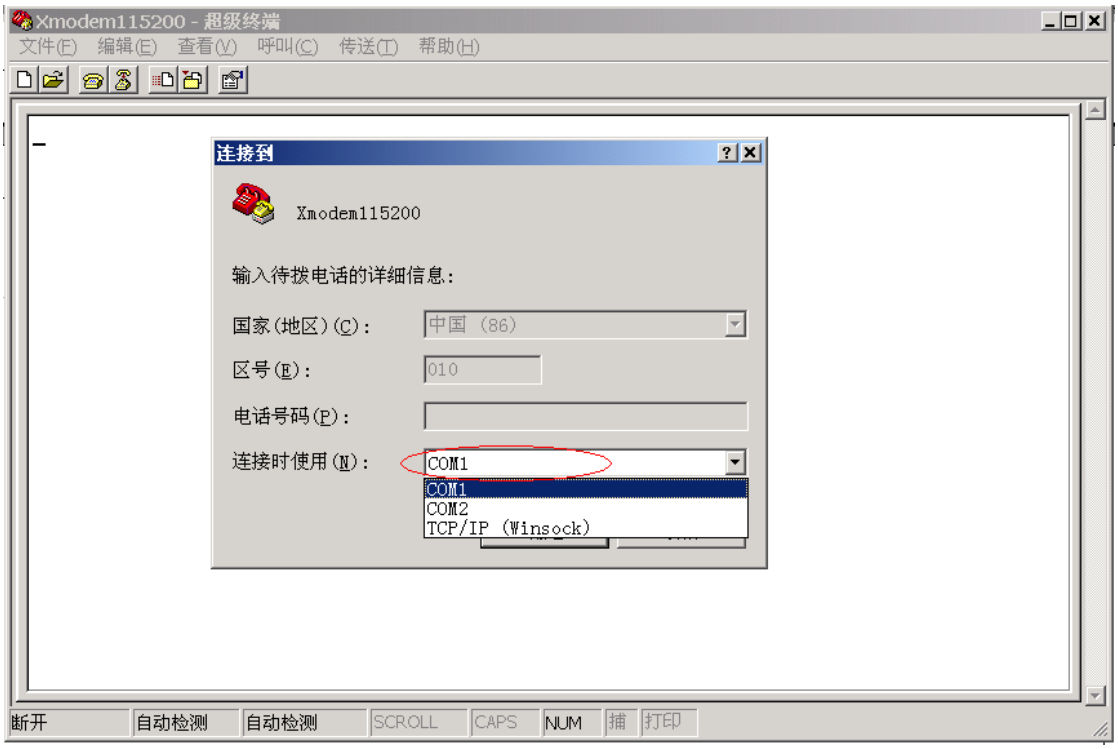



图 3－3 连接到

4、COM 属性，如图 3－4 所示。

在【每秒位数】选择波特率 115200bps，在【数据流控制】中选择无，其它默认，单击【确定】。

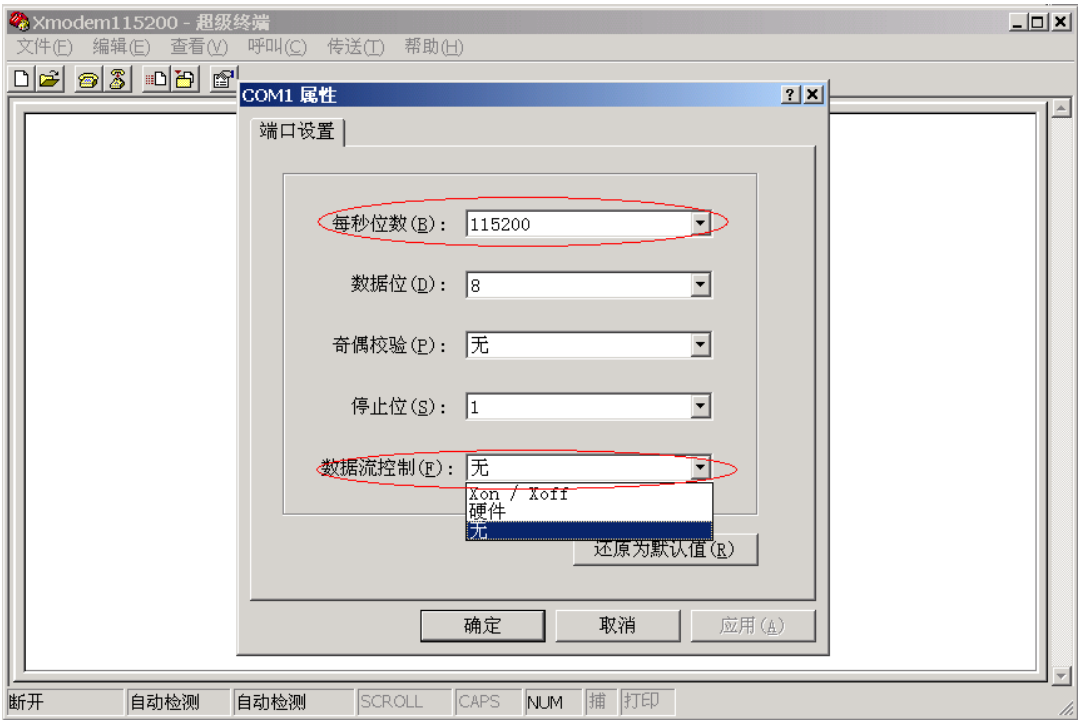



图 3－4 COM 属性

5、按钮介绍，如图 3－5 所示。

有用的按钮这里有三个，连接、断开和发送。

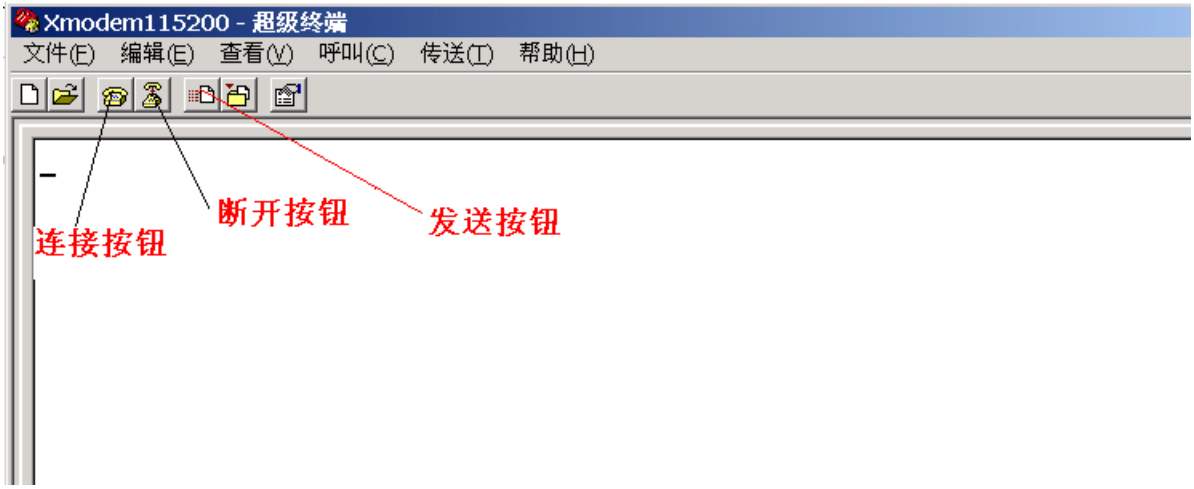
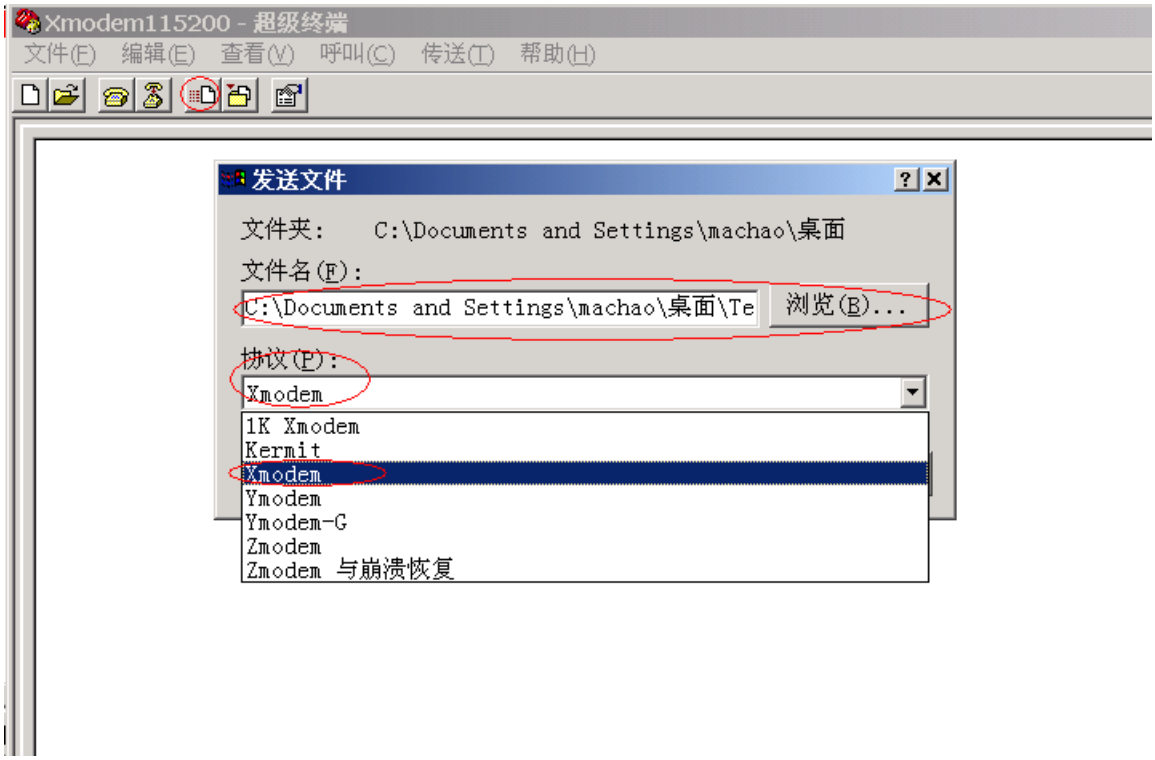



图 3－5 按钮介绍

6、协议选择，如图 3－6 所示。

单击发送按钮，在对话框【文件名】中选择要传输的文件，在【协议】中选择 Xmodem。




7、发送文件，如图 3－7 所示。

在上边都设置完以后，单击【发送】弹出信息框就开始进行文件的传输。

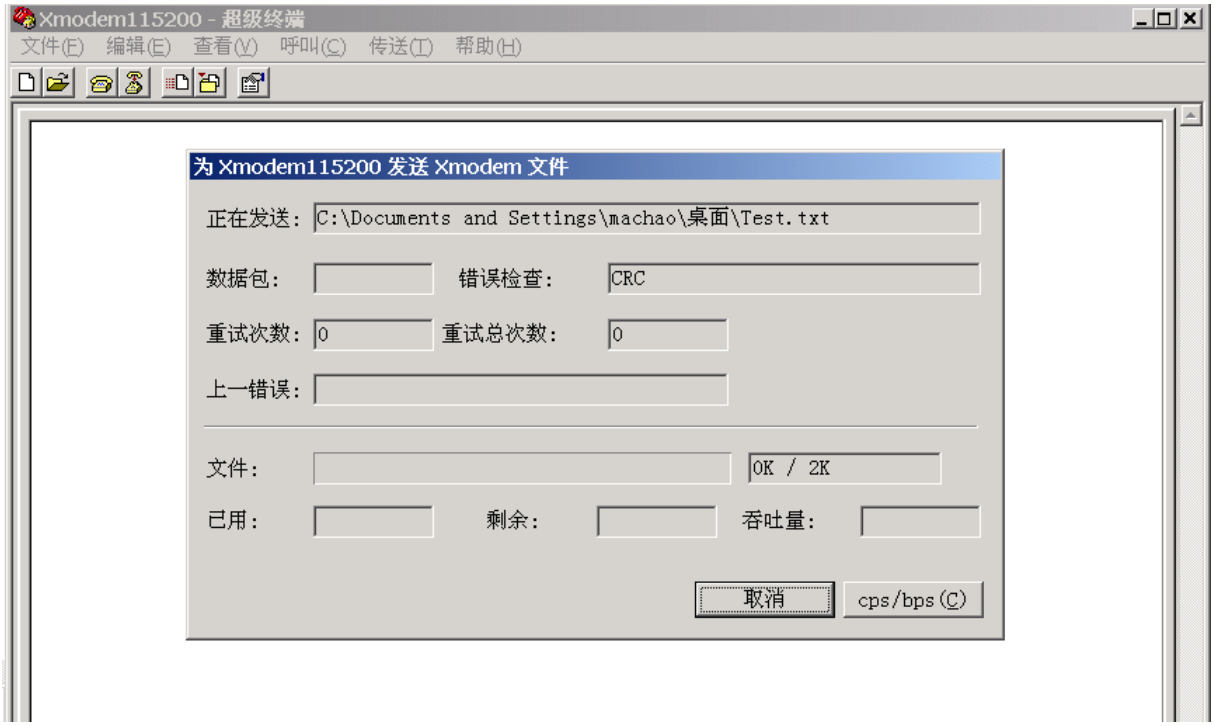



图 3－7 发送文件

8、传输信息，如图 3－8 所示。

传输的文件、数据包、校验方式、重试次数、重试总次数、应答情况、文件大小、用时等信息。

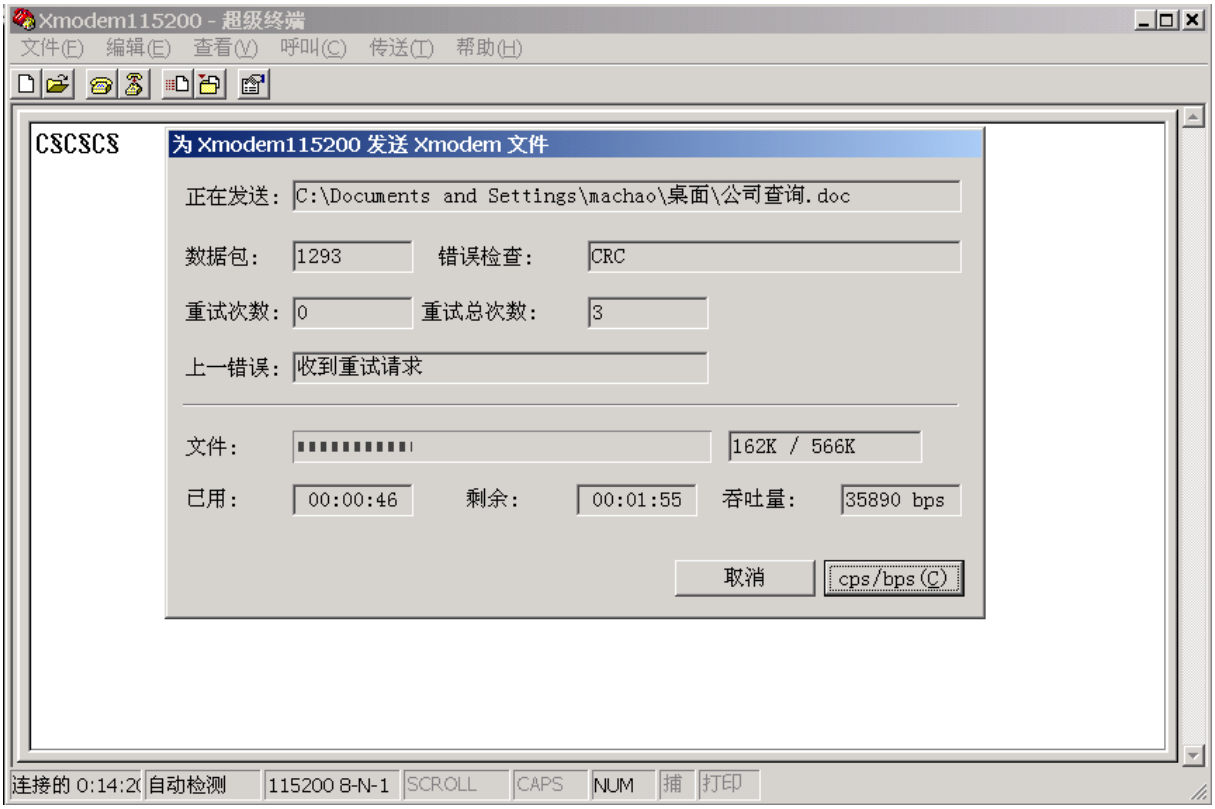



图 3－8 传输信息

# 4 MCU 使用资源

## 4.1 MCU 硬件使用资源说明

| | | | |
|---|---|---|---|
| CPU 型号 | SPMC75F2413A | 封装 | QFP80-1.0 |
| 振荡器 | ☒ crystal | 频率 | 6MHz |
| | ☐外部 | 输入频率 | |
| WATCHDOG | ☒ 有 ☐ 无 | ☐ 启用 ☒ 未启用 | |
| IO 口使用情况 | IOB[0-15] | SRAM HM62864A 地址控制 | |
| | IOC[0-1] | UART 串行通讯接口 | |
| | IOD[0－7] | 与 SRAM HM62864A 的数据通讯接口 | |
| | IOC[8-10] | SRAM HM62864A 控制引脚 | |
| | 剩余 IO 及处理方式 | 主控应用/GND | |
| Timer 使用情况 | CMT0 | 系统定时 | |
| 中断使用情况 | UART(IRQ6) | 串行数据通讯中断，用作接收数据 | |
| | CMT0(IRQ7) | 系统中断 | |
| ROM 使用情况 | 1.78KWord | | |

【附录 1】Xmodem 协议部分。

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>
```

# 5 参考文献

【1】SUNPLUS　SPMC75F2413A 编程指南 V1.0　Oct 12 2004

【2】Xmodem 协议