
Chapter 6

Serial Data Communications - Software Application

A Summary...

Development of software for serial communications. The XON/XOFF protocol. Implementation of simple ACK/NAK protocols. Terminal Emulation and the Kermit protocol.

Read This Chapter If...

- ◆ You want to learn how to program two devices to communicate with one another
- ◆ You want to develop communications protocols for linking devices
- ◆ You would like to learn about commonly used terminal emulation packages.

6.1 Developing Software for Serial Communications

The construction of a physical link between two devices is only one part of the data communications process. In relative terms, the physical hardware link is normally easier to implement than the software that is needed for reliable communications.

When selecting the physical hardware, we choose media and interfacing circuits such that the number of errors on the line is reduced to an absolute minimum. We never assume that a communication link is 100% error free. We do however need to perform a qualitative analysis of the consequences of a data error occurring. If these consequences are minimal (such as in a link between a Personal Computer and printer) then we consciously choose to accept a link that cannot be guaranteed as error-free. In some situations, we have no choice but to accept an unreliable data link and we then need to revert to manual error detection and correction techniques.

Figure 6.1 schematically shows two computers that have been physically connected to one another through a serial communication link. Regardless of which standard we use for the link (RS-232, RS-449, etc.), data will not flow on the link until one (or both) of the devices transfers information from its CPU (or memory) to its UART. In addition, the receiving device requires a program that will read incoming data from its UART in order for the data transfer to have any real significance. Software therefore needs to be developed for each of the devices.

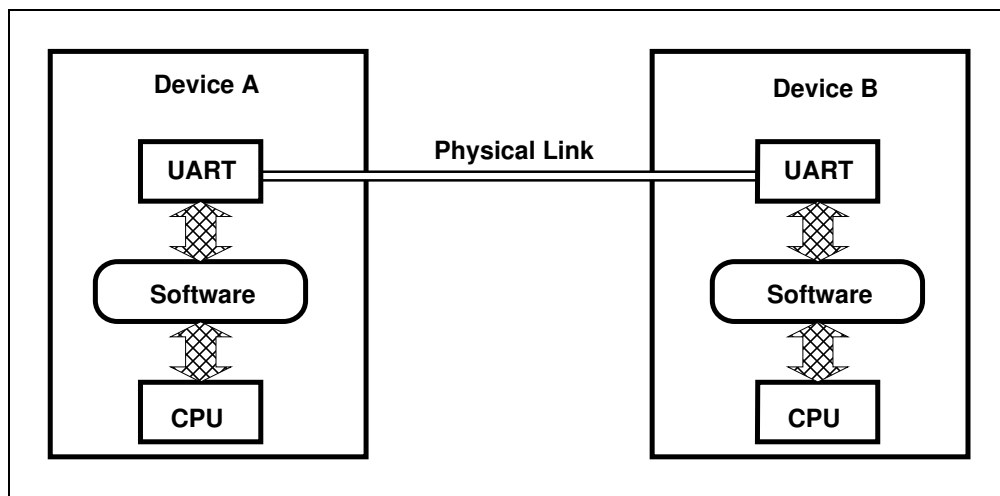


Figure 6.1 - Linking Devices by Hardware and Software

The software required to perform these functions can be as simple or as complex as a system designer makes it. At the lowest level, device A can simply feed out a stream of data and device B can simply read it in and display it. At a more sophisticated level, the transmitting device sends data in packets (or blocks), with error checking performed by the receiver. If we needed a more sophisticated link, then the transmitting device could also send the receiver information (commands) describing how incoming data should be processed once it is received (eg: stored on disk, displayed on screen, etc.).

All these functions are performed by software protocols. There are some commonly used protocols but (as we have come to expect) there are no universal standards. Some of the issues that are addressed by software protocols include:

- (i) The representation of data in character oriented protocols (ie: ASCII, EBCDIC)
- (ii) Establishing the presence and status of a remote device
- (iii) Controlling the flow of information so that the receiving device has time to process the information
- (iv) Error checking and correction techniques
- (v) Recovery from failure of the transmitting or receiving device
- (vi) Resolution of conflict situations where two devices both wish to communicate on the same transmission medium at the same time.

Many of these features add overheads to the transmission of data and subsequently slow down the speed of data transmission. In the final analysis however, the performance of a protocol can be judged by:

- its ability to minimise transmission errors
- the average data transmission rate which can be achieved
- the ability of the software to recover from communications system error conditions.

In subsequent sections we shall look at a number of different protocols, how they can be programmed and used and how they perform.

6.2 The XON/XOFF Protocol

The XON/XOFF protocol is perhaps the simplest of all communications protocols. Its only objective is to prevent a transmitter from sending information to a receiver whilst the receiver is unable to handle that information.

The simplest example is an RS-232 serial link between a computer and a printer. Let us assume that the printer can print at a rate of say 100 characters per second. If we transmit characters to that printer (from a computer) at say 9600 bits per second, then that translates to approximately 960 characters per second (assuming 1 start bit, 1 stop bit, 8 data bits and no parity bits). It should be obvious that we will "overrun" the printer with information in a very short time, since it can only dispose of (print) one character in the time taken for an average of 9.6 characters to arrive. In other words, it is possible that data will be lost.

We could try to resolve this problem by having a memory buffer on the printer. However, no matter how large the memory buffer, if the stream of input data is long enough we will eventually "overrun" the printer. The solution that is commonly employed is simple. When the printer's memory buffer reaches a certain level (upper threshold), it sends the computer the special ASCII character, whose value is decimal 19. This character has the acronym XOFF. When the computer detects the XOFF character, it stops transmitting. Once the level of the printer's memory buffer has reached a lower threshold, it sends the computer the special ASCII character whose value is decimal 17. This character has the acronym XON. When the computer receives an XON it resumes transmission from its previous cut-off point. This simple protocol is shown schematically in Figure 6.2.

Note that with the XON/XOFF scheme, the receiving device must have enough room in its memory buffer to account for the characters coming in whilst it is sending an XOFF to the transmitter. Conversely, to avoid having a receiver idle whilst it waits for the transmitter to resume, the receiver must transmit an XON, at a point that will leave enough characters in the memory buffer to account for the time delay. In other words XOFF and XON are always sent before the printer is 100% full or 100% empty respectively.

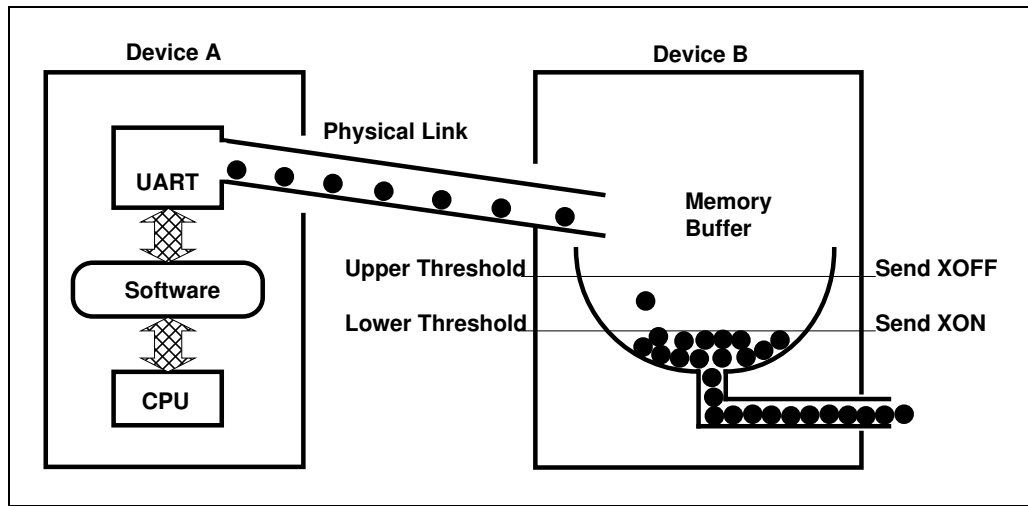


Figure 6.2 - The XON/XOFF Protocol

Another concept that arises in discussing XON/XOFF protocols is the "**circular buffer**". A circular buffer is created in software and is simply a First-In, First-Out (FIFO) memory area that is indexed as each item is removed. This is shown in Figure 6.3. Normally, circular buffers are programmed so that reading is a destructive operation. That is, once a "read" has taken place, the buffer is indexed and the contents of the read location are replaced with new information.

Circular buffers are commonly used at receiver inputs so that information is not lost due to the time constraints of the receiver. Typically, in serial communication, end-users write "Interrupt-Service-Routines" (ISRs) which are automatically activated each time new data enters a UART register. The role of the ISR is to remove incoming data from the UART registers (which are volatile and overwritten each time data enters the UART) and place it into conventional memory. The conventional memory is structured as a circular buffer and normal end-user programs read the data from the buffer. The buffer is indexed each time the user reads from it.

Circular buffers can also be used at transmitter outputs (as output buffers), so that the transmitter does not need to wait until each character is output before sending the next. It is the level of the circular input buffer that is generally responsible for triggering XOFF/XON transmission by a receiver.

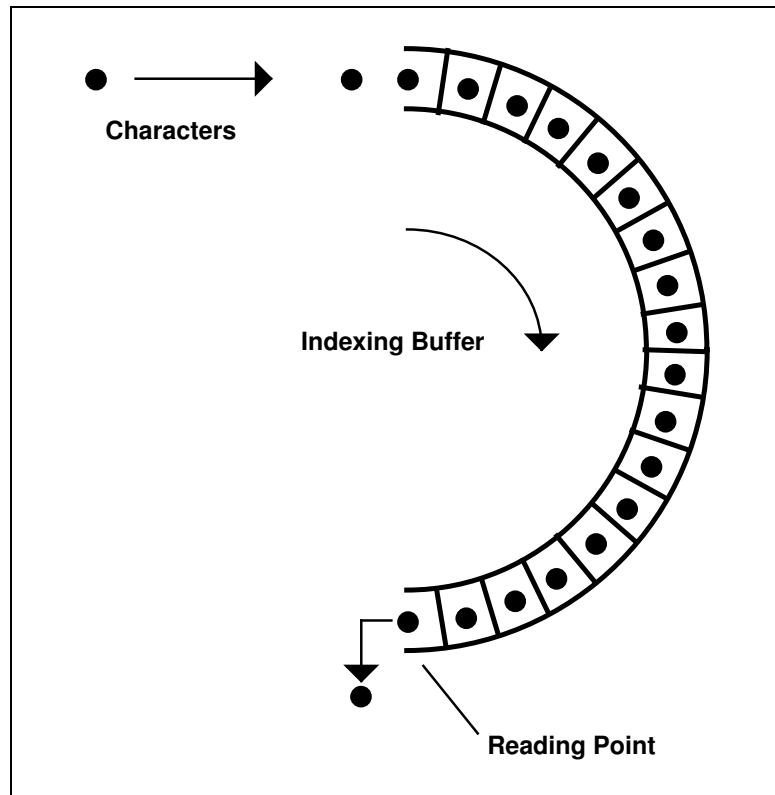


Figure 6.3 - The Circular Buffer Concept

An important point to note about the XON/XOFF protocol is that if it is efficiently implemented, then XON and XOFF should not appear as characters in a circular buffer. Programmers can usually enable or disable the XON/XOFF facility on the UART of each device. When XON/XOFF is enabled, the UARTs respond to these special ASCII bit patterns by switching on or off transmission and the characters are always treated as special control characters - not as data. The protocol is then transparent to user programs. However, when the XON/XOFF facility is disabled, the characters are treated in exactly the same way as normal data and have no effect on flow control. If XON and XOFF are treated as normal data, then it is clearly the programmer's task to generate software which will:

- read the characters
- isolate them from normal data
- carry out the flow control task.

The XON/XOFF protocol and the circular buffer are an effective way of governing data flow in simple links. However, they do not address any of the more complex communications issues related to error detection and correction. The main role of XON/XOFF therefore remains with computer to printer links.

6.3 ACK/NAK Protocols

Once we leave behind relatively simple protocols such as XON/XOFF, we also leave behind the concept of standardisation. There are no complex, point to point serial communications protocols which have been universally adopted in industry, although there are a few which are in common usage.

From this point onwards we will therefore try to examine, in broad terms, the sort of protocols that are in use. They will all be classified as ACK/NAK protocols, for reasons that we shall discover later. There are countless variations on a central theme, so our main objective here is to "define" our own generic protocol and see how this is implemented through software programming. Once you have seen how this hypothetical ACK/NAK protocol is implemented then you should be able to translate the results to other situations.

Most modern, computer-controlled industrial equipment is fitted with at least one RS-232 port. On many devices, this serial port does nothing more than allow a complete program or data file to be transferred into or out of that device. This is sometimes referred to as a "file-dump" facility (rather than a protocol). When a dumped file arrives at its destination, the receiving device has no way of knowing whether the information contained in that file has been corrupted along the communications link. We will look at file-dumping and its consequences a little later. However, we will firstly examine the implementation of ACK/NAK protocols that do perform error checking and correction.

One may well ask why we would ever need to develop a piece of software to implement a communications protocol. The answer to this question is simple. Many computer based devices are designed to communicate through a software protocol. The protocol software on some devices is installed by the device manufacturer and cannot be changed by the user. Therefore if one wishes to transfer a file from a computer to such a device, then that computer must be programmed to conform to the other device's protocol. Sometimes such software is already provided by the device manufacturer. For example, a CNC manufacturer may supply the PC software required for the PC to communicate using the CNC's protocol. As often as not, this software is not readily available and some in-house development must be performed.

As we shall see, the successful development of protocol software requires a sound knowledge of programming concepts. It also requires a considerable amount of time and therefore expense. A lack of standardisation in point to point protocols (particularly those implemented on RS-232 links in manufacturing devices) means that it is often necessary to develop the complementary, computer (host) side software in order for communication to occur. One can then understand why "integration" in manufacturing can become so costly. In subsequent chapters we shall see how attempts have been made to standardise communications protocols in manufacturing and why standardisation is so difficult.

Before proceeding further with software protocol development, we need to re-examine the first 32 characters of the ASCII character set (see Table 1.3). We shall be making use of some of these characters as we progress because most of them are intended for communications purposes. The bit patterns for the first 32 ASCII characters do not generally appear amidst normal text (data) in user files, except where they are used as delimiters (eg: carriage return). There is therefore nothing remarkable about these characters, other than the fact that they are normally reserved for special purposes. It is also important to note that although each of these characters may have had some specific task when originally defined, their application often varies considerably, depending upon the protocol in use. The ones that we shall be using in our hypothetical protocol are:

<i>ASCII</i>			
		<i>Hex Value</i>	<i>Decimal Value</i>
<i>STX</i>	(Start of TeXt)	02	02
<i>ETX</i>	(End of TeXt)	03	03
<i>EOT</i>	(End of Transmission)	04	04
<i>ENQ</i>	(ENQuiry)	05	05
<i>ACK</i>	(ACKnowledge)	06	06
<i>DLE</i>	(Data Link Escape)	10	16
<i>NAK</i>	(Negative AcKnowledge)	15	21

Most of the acronyms given to the characters are self-explanatory, but the role of the DLE character is not quite clear. The basic objective of the Data Link Escape character is to separate normal data from communication link control functions. In other words, when a receiver reads in strings of characters, the DLE becomes a delimiter by which it can determine that either the data has ended or the link control has ended. The actual use for the DLE seems to vary significantly from one protocol to another and so it is best viewed as nothing more than a delimiter. As we go through the development of our protocol you can see one particular application for the character.

Our hypothetical protocol (which we shall name HYPOLINK) will be developed in such a way that two devices (computers) are able to transfer data files to one another as shown in Figure 6.4. The originator of the file will be able to tell the recipient whether to display the file on screen or store it on disk under a particular file name. The protocol will include some error detection and correction facilities through a Block Check Sum (BCS) calculation.

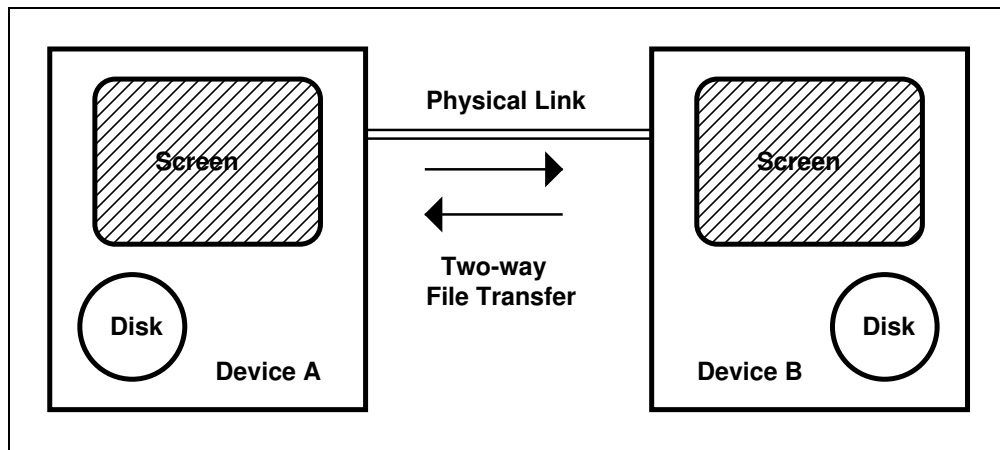


Figure 6.4 - Basic Model for Point to Point Serial Protocol 'HYPOLINK'

The rules of the HYPOLINK protocol are defined with respect to the diagram shown in Figure 6.4 as follows:

- (i) Communication between devices is based upon RS-232 signalling with no hardware hand-shaking.
- (ii) The following transmission parameters are used:
 - 9600 bits per second
 - 1 start bit
 - 8 data bits
 - 1 stop bit
 - Odd parity
 - Half-duplex transmission
 - XON/XOFF protocol disabled

- (iii) Only ASCII characters can be transmitted as information on the link. With the exception of the control characters ENQ, ACK, NAK, and EOT all information on the link must be transmitted in either command, data or message packets of the form shown in Figure 6.5. For the purposes of this protocol, a packet is defined to be composed of an information field, encapsulated between delimiters. The DLE character is used to signify to the recipient that the next character is a communication control character (STX or ETX) and not data. If a DLE character should coincidentally appear within a transmitted data file, then the DLE must be repeated to avoid ambiguity.

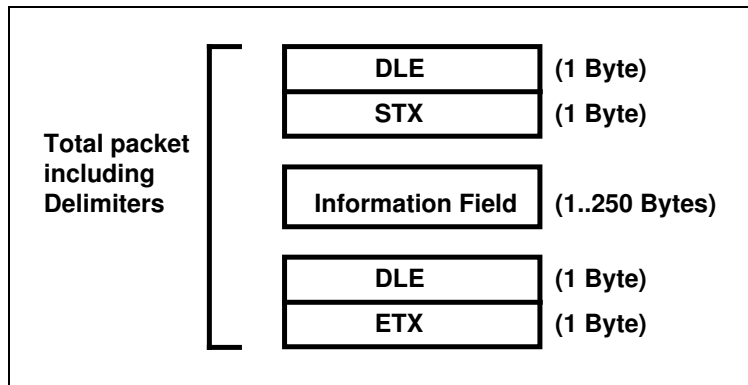


Figure 6.5 - Structure of Command, Data and Message Packets

- (iv) The information field of a command packet has the structure shown in Figure 6.6. The first byte in the information field is the ASCII character 'C', which signifies a command packet. The second byte is either an 'S' or 'D' and is used to instruct the recipient to show the file on *Screen* or write it to *Disk* respectively. The Source File Name is the name of the transmitted file as it appears on the originator. The Target File Name is the disk file name under which the recipient is to store the received file. If the recipient only shows the file on screen then it ignores the Target File Name.

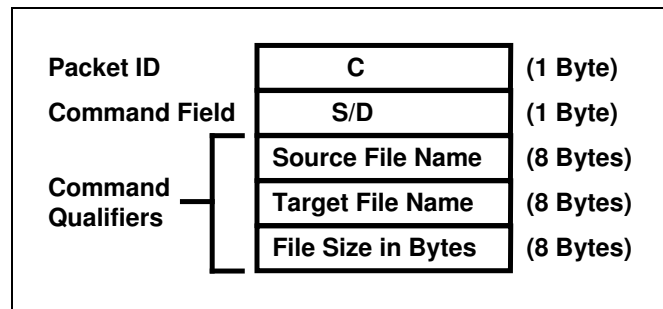


Figure 6.6 - Structure of Information Field for Command Packet

- (v) The information field of a data packet has the structure shown in Figure 6.7. The first byte in the field is the ASCII character 'D' to signify a data packet. The ASCII data section contains a block of the file being transmitted.

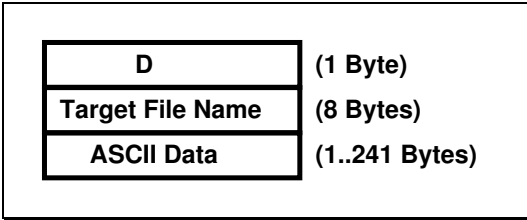


Figure 6.7 - Structure of Information Field for Data Packet

- (vi) The information field of a message packet has the structure shown in Figure 6.8. The first ASCII character in the field, 'M', signifies a message packet. The Message Code is a single ASCII character that tells the originator of a command packet whether or not the recipient is able to carry out a command after that command has been correctly received.

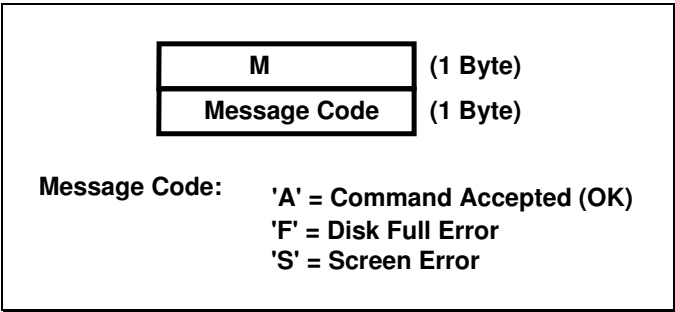


Figure 6.8 - Structure of Information Field for Message Packet

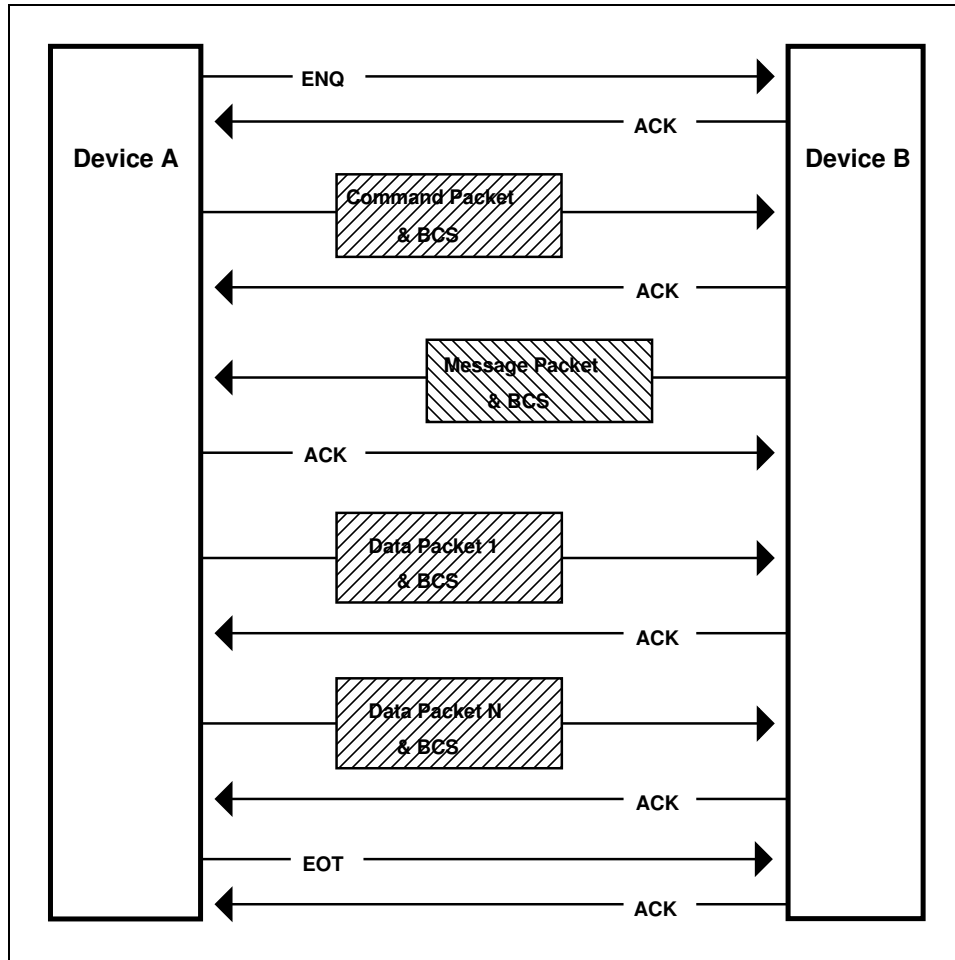
- (vii) After each command or data packet is transmitted on the link, the originator will transmit a Block Check Sum (BCS), which is the exclusive or (xor) of the bit pattern of every transmitted character in the information field of a packet.
- (viii) A receiving device will carry out the same BCS calculation as the transmitter, on each incoming packet. The receiver will then compare the incoming BCS with its own local value. If they are the same then the receiver will send an ACK to the transmitter, otherwise it will send a NAK.

- (ix) If a transmitting device receives a NAK (negative acknowledgment) from the recipient, then it must re-transmit the last data packet. If a transmitter does not receive an ACK (positive acknowledgment) after 3 attempts at transmission then it must assume link failure and cease communication.
- (x) Communication between two devices consists of six phases:
- Establishment of Link
 - Issue of a Command Packet
 - Acknowledgment of a Command Packet
 - Dissection and Transmission of Packetised Data
 - Error Detection and Correction
 - Termination of Transmission
- (xi) Any remote device that does not respond to an ENQ, command or information packet within 4 seconds, with either an ACK or a NAK, will be assumed to be off-line and therefore the originator of a communications sequence will desist from further communications activity.
- (xii) After the final packet in a communications sequence is transmitted by the originator, it must terminate transmission by sending an EOT character that must be acknowledged by the recipient.

A typical communications sequence for the Hypolink protocol is shown in Figure 6.9. In order to implement such a protocol in software, we need to have a number of sub-programs (procedures) which will:

- set the serial port (UART) to the appropriate bit rate, parity system, etc.
- write a character to the serial port and report errors
- read a character from the serial port and report errors.

The software for reading a character from the serial port needs to be intelligent and "interrupt-driven". That is, each time a character arrives into the UART, a system interrupt (and Interrupt Service Routine) should cause the character to be automatically removed from the UART's register and placed into a circular memory buffer. The program that uses the "serial port read" procedure is then actually reading from memory and not the UART itself. The reason for this is straightforward. If several characters arrive into the UART's input register before the program has "read" and acted upon the previous character, then data will be lost. It is therefore necessary to set up a memory buffer, whose length will accommodate the time delay between the "serial port read" statements in the executing program.



*Figure 6.9 - Typical 'HYPOLINK' Communications Sequence
(Assuming No Transmission Errors)*

The development of software to execute the functions noted above is generally carried out in assembly language and is totally dependent upon the computer hardware in use. For many programming languages (compilers) available on PCs, the appropriate procedures (sub-programs) have already been developed and are either included as part of the package, or can be purchased as optional "tool-boxes". The form of these procedures is usually such that a "parameter list" is used to pass information to the serial ports and return error codes (parity, overrun, etc.) generated on the serial port/s. Let us assume that the three procedures are:

- **Set_Serial_Port_Parameters** (Port, Baud, Parity, Stop_bits)
- **Write_Character_to_Port** (Port, Output_Character, Error_code)
- **Read_Character_from_Port** (Port, Input_Character, Error_code)

where:

- Port is the number of the serial port (*)
- Baud is the Baud rate (*)
- Parity is 0 for Even and 1 for Odd (*)
- Stop_bits is either 1 or 2 (*)
- Input_Character is the ASCII character read from the port (#)
- Output_Character is the ASCII character written to the port (*)
- Error_Code is the returned status of the serial port (#)

and:

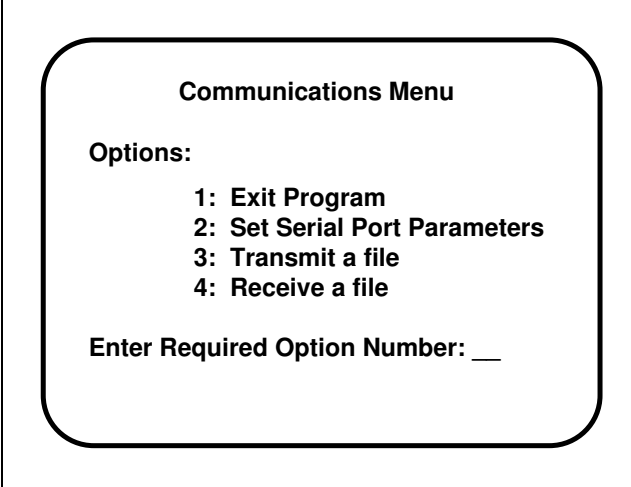
- * indicates that the parameter is passed to the procedure
- # indicates that the parameter is returned from the procedure.

We shall use these procedures to create a PASCAL program that implements the above protocol. The purpose of this exercise is to illustrate how high level language structures can be used to develop protocols, and not to highlight any language-specific constructs. For this reason, we will use only the simplest (generic) programming structures. The only salient points you need to note about the PASCAL syntax are that:

- Individual statements are always separated by a semi-colon (;)
- The symbol ":=" is read as "becomes equal to"
- A group of program statements is lumped together into a block through the use of the key words called "Begin" and "End"
- Program comments are contained between braces { }
- The "\$" symbol is used to indicate a hexadecimal value
- Sub-programs are referred to as "Procedures"
- Procedures are called up from a main program or another procedure simply by issuing the name of the procedure, together with a list of parameters that are passed to and from that procedure. Parameters are enclosed in brackets ().

Most of the program loops and structures you will find in the code are common to all modern, structured programming languages. You should also note that in the included program examples, global variables are used in place of local variables in order to make the programs easier to read for those not familiar with PASCAL syntax. In the program listings provided below, all PASCAL "reserved words" are highlighted in bold text

In order to make the HYPOLINK protocol function sensibly, each of the devices on the link needs a simple user interface, where either the transmit or receive modes can be selected. This is shown in Figure 6.10.



Communications Menu

Options:

- 1: Exit Program**
- 2: Set Serial Port Parameters**
- 3: Transmit a file**
- 4: Receive a file**

Enter Required Option Number: ___

Figure 6.10 - User Screen Interface on Communicating Devices

The main program executing on each device is called the "Applications Program" and is given the name "Serial_Communications". It is little more than a menu generation program which calls a number of high-level procedures that actually carry out the communications tasks. The structure of the main program is as follows:

```
Program Serial_Communications;
```

```
Const
```

```
{Define All Communications Constants}  
STX = Chr ($02);  
ETX = Chr ($03);  
EOT = Chr ($04);  
ENQ = Chr ($05);  
ACK = Chr ($06);  
DLE = Chr ($10);  
NAK = Chr ($15);
```

```
Var
```

```
{Define All Communications Variables}  
Menu_Number      : Integer;      {Main Menu Selection}  
Error_Code       : Integer;      {Serial Port Error}  
Information_Field : String [250]; {Data Packet Contents}  
Remote_BCS       : Char;         {Block Check Sum}  
Local_BCS        : Char;         {Block Check Sum}  
Character        : Char;  
EOT_Received     : Boolean;      {Last Packet Flag}
```

```
Begin {Main Program}
```

```
Clear_Display_Screen;
```

```
Repeat
```

```
Write_Main_Menu;  
Select_User_Option (Menu_Number);  
Case Menu_Number of  
  1: Clear_Display_Screen;  
  2: Serial_Port_Set_Up;  
  3: File_Transmission_Mode;  
  4: File_Receive_Mode;
```

```
End {Case};
```

```
Until Menu_Number = 1;
```

```
End {Main Program}.
```

The main program looks straightforward, but what do high-level procedures such as "File_Transmission_Mode" and "File_Receive_Mode" actually look like? Clearly, such a program cannot execute until these procedures are defined.

Before we develop the high-level procedures used by the applications program, it is necessary to see how some of the lower level procedures are constructed. These procedures will firstly read in the information field of a single data packet (from the serial port) and secondly, execute the software hand-shaking and error checking sequence demanded by our protocol. The first of these procedures, which will take an incoming data packet, shed the link control characters (DLE, STX, ETX) and store the information field into the variable "Information_field" is as follows:

```
Procedure Read_Data_Packet;

Var
    {Local Indexing And Character Storage Variables}
    String_index      : Integer;
    Character         : Char;

Begin {Read_Data_Packet Procedure}

    Information_field := ""; {Initialise the information field to a null string}

    {Wait for a DLE-STX sequence or wait for an EOT}
    Repeat
        Repeat
            Read_Character_from_Port (1,Character,Error_code);
        Until (Character = DLE) or (Character = EOT);
        If Character = DLE then
            Read_Character_from_Port (1,Character,Error_code)
        Else
            EOT_Received := True;
    Until (Character = STX) or (EOT_Received);

    If NOT EOT_Received then
        Begin {Read Packet}
        String_index := 1;
        While (Character <> DLE) and (String_Index < 250) do
            Begin {Read information field}
            Read_Character_from_Port (1,Character,Error_code);
            Information_field [String_index] := Character;
            String_index := String_index + 1;
            End {Read information field};
        Read_Character_from_Port (1,Character,Error_code); {ETX}
        End {Read Packet}

    End {Read_Data_Packet Procedure};
```

The above procedure assumes that we are receiving through serial port number 1. The procedure also assumes that no DLE characters will appear in the ASCII data. If there was a possibility that a DLE character could appear in the data then the procedure would have to be modified. Also note that the above procedure checks for an EOT character as a first step and sets up an EOT_Received flag. At the moment the procedure is still very crude. If everything goes according to the protocol it will work - but what would happen if the DLE-ETX sequence didn't appear after 250 characters were read into the information field?

The next procedure is the one which checks for errors in the information field of the packet that has been read by the Read_Data_Packet Procedure. This procedure compares the incoming Block Check Sum (BCS) with its own local calculation. If they are the same then it issues an ACK, otherwise it issues a NAK in order to request a re-transmission.

```
Procedure Read_and_Acknowledge_Packet;

Var
    {Local Transmission Counter Variable}
    Receive_counter : Integer;

Begin {Read_and_Acknowledge_Packet Procedure}

Receive_counter := 0; {Initialise}

Repeat
    Read_Data_Packet;
    If EOT Received then
        Write_Character_to_Port (1,ACK, Error_code)
    Else
        Begin {Check and Correct Transmission Errors}
        Read_Character_from_Port (1,Remote_BCS,Error_code);
        Calculate_Local_BCS;
        If Remote_BCS = Local_BCS then
            Write_Character_to_Port (1,ACK, Error_code);
        Else
            Begin {error}
            Write_Character_to_Port (1,NAK, Error_code);
            Receive_counter := Receive_counter + 1;
            End {error};
        End {Check and Correct Transmission Errors};
Until (Remote_BCS = Local_BCS) or (Receiver_counter = 3) or (EOT_Received);
End {Read_and_Acknowledge_Packet Procedure};
```

If we were serious about covering all the possibilities, then we should really be acting upon any error message that arises from either the "Read_Character_from_Port" procedure or the "Write_Character_to_Port" procedure. The sort of errors which can occur include parity, overrun, etc. One needs to keep in mind the cumulative effect of all the possible error conditions we have thus far neglected to flag. We have also failed to account for situations where the link is severed in the middle of the transmission sequence. The procedures provided are inadequate because they either fall into endless loops or else cause the program to crash entirely. However, accounting for possible link failure conditions complicates the structure of all the above procedures, and sometimes we may choose to ignore certain error conditions, with the expectation that they may be rare.

Once we have established these two procedures (plus the minor procedures which performing menu selection, screen handling, etc.) we can look at the File_Receive_Mode procedure, which handles the entire hand-shaking sequence and data transfer. This could be developed in the following procedure form:

Procedure File_Receive_Mode:

Var

```
{Local Device Status Flags}
Screen_OK      : Boolean;
Disk_OK       : Boolean;
Disk_Transfer  : Boolean;
```

Begin {File_Receive_Mode Procedure}

```
EOT_Received := False;
Clear_Display_Screen;
```

Writeln ('Waiting to Receive a file...');

Repeat

```
  Read_Character_from_Port (1,Character,Error_code);
```

Until Character = ENQ;

```
Write_Character_to_Port (1,ACK,Error_code);
```

```
Read_and_Acknowledge_Packet; {Incoming Command Packet}
```

```
If Information_field [1] = 'C' then  
    Case Information_field [2] of  
        'S': Begin { Screen listing }  
            Disk_Transfer := False;  
            Check_Screen_State (Screen_OK);  
            If Screen_OK then  
                Transmit_Message ('A') {Command Accepted}  
            Else  
                Transmit_Message ('S'); {Screen Error}  
            End {Screen listing};  
        'D': Begin {Disk store}  
            Disk_Transfer := True;  
            Check_Disk_Space (Input_File_Size, Disk_OK);  
            If Disk_OK then  
                Transmit_Message ('A') {Command Accepted}  
            Else  
                Transmit_Message ('D'): {Disk Error};  
            End {Disk store};  
    End {case};  
Repeat  
    Read_and_Acknowledge (Packet);  
    If Disk_Transfer then  
        Write_Information_field_to_Disk  
    Else  
        Write_Information_field_to_Screen;  
Until EOT_Received;  
Writeln ('End of File Transfer...');  
End {File_Receive_Mode Procedure};
```

At this point we have established the structure for the file receiving procedure. We now need to look at the structure of the File_Transmission_Mode procedure, which will perform the complementary side of the protocol during transmission. The structure of this procedure would be in the following form:

```
Procedure File_Transmission_Mode;
Var
    Local_File_Name      : String [8];
    Remote_File_Name     : String [8];
    Transfer_Mode       : Char;

Begin {File_Transmission_Mode Procedure}

Clear_Display_Screen;
Write ('Enter file to be transmitted: ');
Readln (Local_File_Name);
Write ('Transfer to remote Disk (D) or Screen (S): ');
Readln (Transfer_Mode);

If Transfer_Mode = 'D' then
    Begin {Get remote name}
        Write ('Enter target file name on remote device: ');
        Readln (Remote_File_Name);
    End {Get remote name};

Writeln ('Transmitting file ',Local_File_Name,'...');
Write_Character_to_Port (1, ENQ, Error_code);

Repeat
    Read_Character_from_Port (1,Character,Error_code);
Until (Character = ACK) or (Time_Out);

Transmit_Command (Transfer_Mode);
Read_and_Acknowledge_Packet; {Reply Message to Command}
If Information_field [2] = 'A' then
    Begin {File Transmission}
        Repeat
            Read_Block_From_File_to_Info_Field (End_of_File);
            If NOT End_of_File then
                Transmit_Data_Block;
        Until End_of_File;
        Write_Character_to_Port (1,EOT,Error_code);
        Writeln ('Transmission completed...');
    End {File Transmission}

End {File_Transmission_Mode Procedure};
```

The "File_Transmission_Mode" procedure gives the basic structure for the transmission of a file under the HYPOLINK protocol. There are obviously a number of other procedures which need to be developed in order to realise transmission. Procedures such as "Transmit_Command", "Transmit_Message" and "Transmit_Data_Block" build up and send a packet through the serial link.

The structure of the "Transmit_Data_Block" procedure is shown below and is typical of the packet transmission procedures:

```
Procedure Transmit_Data_Block;  
  
Var  
    Index           : Integer;  
    Transmission_index : Integer;  
  
Begin {Transmit_Data_Block Procedure}  
  
    Transmission_index := 0;  
  
Repeat  
    Write_Character_to_Port (1,DLE, Error_code);  
    Write_Character_to_Port (1,STX, Error_code);  
    For Field_index := 1 to length [Information_field] do  
        Write_Character_to_Port (1,Information_field [index], Error_code);  
        Write_Character_to_Port (1,DLE, Error_code);  
        Write_Character_to_Port (1,ETX, Error_code);  
        Read_Character_from_Port (1,Character, Error_code);  
        Transmission_index := Transmission_index + 1;  
Until (Character = ACK) or (Transmission_index = 3);  
End {Transmit_Data_Block Procedure};
```

These procedures are only intended as a rough guide to how one goes about developing the software for a communications protocol. The procedures shown above are already showing signs of complexity, even though our hypothetical protocol is very simple. The more that you study these procedures the more scope you will find for program failure under abnormal circumstances. For example, what happens if a communications error causes an STX to appear as an ETX? What happens if a DLE and ETX appear as part of the data? The list of "what-if" scenarios is enormous.

We can continue to enhance procedures, such as those shown above, to take into account a much broader range of abnormal conditions. In the final analysis however we must accept that no protocol will account for every possible error that can occur. The final implementation is ultimately based upon a sensible calculation of risks and possible error conditions.

A good way of determining how far one should proceed with error checking in a protocol is to ask the question:

"What can a device do to rectify a given problem through software?"

In many cases the answer is nothing except to flag an error. Take a good look at the Hypolink protocol for examples. What would happen if a receiver found that there were 251 characters in the information field of a packet? How would it resolve the problem? Would it assume that the entire packet is worthless and ask for a re-transmission? Would it assume that the first 250 characters are correct and that the 251st was a corrupted delimiter?

The net result is that it is often too difficult to account for every error. There are many instances when the software must be designed to simply abort a transaction. The most obvious example of this is when an acknowledgment is not received to an enquiry, but there are numerous other combinations that need to be thought out by the protocol developer.

The protocol software developed above is clearly just a crude beginning to what must inevitably be a demanding task for those who attempt to program protocols - even when the protocols themselves are not complex.

6.4 Communications Software Layering

In section 6.3 we examined how the PASCAL source code for a hypothetical communications protocol could be developed. An important point to note about the exercise is the way in which various software modules or procedures interact with one another.

We started the process of establishing a serial communication link by discussing the type of hardware (UARTs, cabling and hand-shaking) that could be used to facilitate the link - for example, RS-232 or RS-449. We then moved on to look at some basic pieces of software that would interact with that hardware in order to perform some meaningful communications task. In our case, these procedures were called:

- Set_Serial_Port_Parameters
- Write_Character_to_Port
- Read_Character_from_Port.

Once the basic hardware and software procedures were put into place, it was then possible to develop other software procedures that could perform some data packet assembly and disassembly, error-detection, correction and hand-shaking between devices. Ultimately we developed two high-level procedures:

- File_Transmission_Mode
- File_Receive_Mode.

These procedures could be incorporated into a user program (application) to perform a file transfer operation. We can schematically represent the structure that was achieved in our protocol development as shown in Figure 6.11.

The diagram of Figure 6.11 is a "layered model" of the software and the hardware that we have put together in order to create a functioning communication link. Each layer in the system is a well-defined entity. It takes information from the layer above or below, processes that information and feeds the output to the layer below or above. A layer that is involved with the actual link hardware (including modulation, cables, connectors, etc.) is referred to as the "physical layer". At the other end of the spectrum, the layer that most closely interacts with the applications program is referred to as the "applications layer". However, there is no question of one layer being any more or less important than any other layer. For without all the layers present, the protocol is not satisfied.

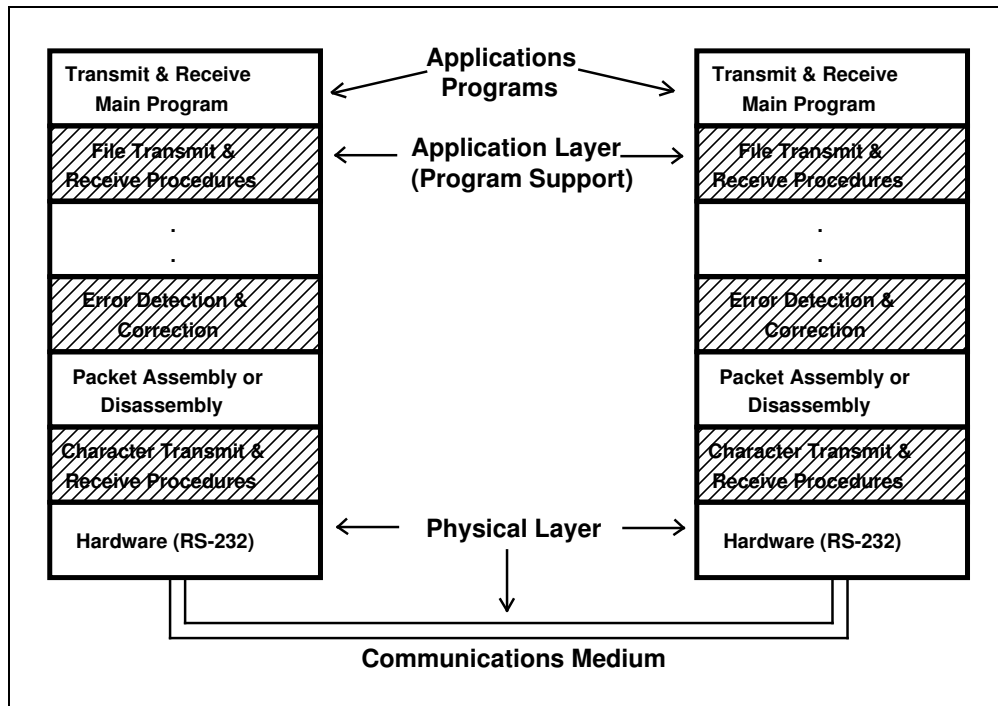


Figure 6.11 - Schematic of Communications "Layering" Concept

The advantage of looking at communications system development from a "layered" point of view is flexibility. Provided that the information flowing into or out of a layer (ie: the upper and lower interfaces) remains the same, then the internal operation of any one layer can be completely independent of the surrounding layers.

Let us look at some examples of this point. Supposing that we replaced the communications medium and RS-232 hardware layer with a different standard - say RS-449. Provided that the RS-449 hardware on each device could supply the read and write procedures (of the layer above) with the same information as the RS-232 system, then these procedures could not detect the change.

At the other end of the spectrum, we can look at the applications programs. It is irrelevant how we structure these programs or their user interfaces. It is irrelevant how we construct menus or screen displays. Provided that we call the file transfer procedures of the layer below in the correct way, then they can still function independently.

We shall look more closely at layering of communications software and hardware in a subsequent chapter.

6.5 Terminal Emulation and the Kermit System

One of the major applications for serial links (such as those defined by RS-232) is in communications between a mainframe (or workstation) computer system and a terminal.

Originally, a computer terminal was little more than a Visual Display Unit (VDU) and keyboard. The device essentially had no processing or bulk storage facilities and so was referred to as a "dumb terminal". In today's computer environment however these so-called "dumb" terminals have often been replaced with Personal Computers (PCs). PCs can either act as intelligent devices in their own right or else mimic particular brands of (dumb) computer terminals. This provides the best of both worlds. The mainframe is relieved of tasks such as word-processing and spreadsheets, but is available for centralised data (file) storage and complex tasks (such as advanced CAD, MRP, etc.)

A schematic for a typical computer to terminal arrangement is shown in Figure 6.12.

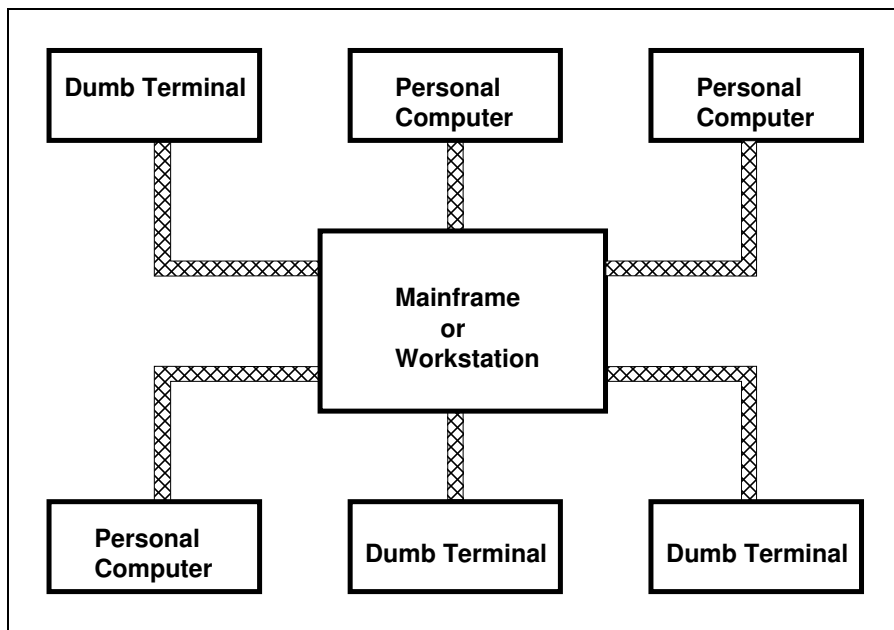


Figure 6.12 - Typical Computer to Terminal Arrangement

Over the years, many of the major computer manufacturers have developed dumb computer terminals that can interact with their mainframe systems in a clearly defined way. A number of these terminals are in very common usage in industry and many multi-user operating systems, on computers, are specifically designed to facilitate communications with such devices.

Normally, alpha-numeric characters on dumb terminals are represented by bit patterns corresponding to either 7 bit ASCII or 8 bit EBCDIC codes. However there are many other keys on a typical dumb terminal besides alpha- numerics. For example, the function keys, scroll lock keys, cursor movement keys, etc. The bit patterns for these are not defined by either the ASCII or EBCDIC system. Therefore, if we wish to replace a dumb terminal with a PC, it is necessary to have a piece of software that will make the PC read and write through its serial port in the same way as a dumb terminal. It is additionally necessary to convert the various bit patterns generated by special purpose function keys on a PC to the bit patterns that would be generated by a specific type of terminal.

The software used to perform these functions is referred to as "Terminal Emulation" software. At its most basic level, a Terminal Emulator echoes data (coming from the mainframe) to the PC screen and transmits data (from the PC keyboard) through its serial port. However, commonly available Terminal Emulation packages tend to do far more than just emulate specific types of terminals. The reason for this is because the role of a PC acting as a terminal to another computer is totally different to the role of a dumb terminal.

A PC acting as a terminal to a mainframe can do a vast amount of its own processing work - for example spreadsheets, word-processing, desktop publishing, micro-CAD, etc. In these situations, the PC generates data files that can then be fed into the mainframe for centralised storage and access. Conversely, files can be fed from the mainframe back to a PC as part of the retrieval cycle. Dumb terminals only read in data from a mainframe for display purposes. Data written on a dumb terminal is transmitted back to the mainframe as soon as it is keyed in. There are normally no local file generation or file transfer capabilities associated with dumb terminals.

If PCs are to act as terminals and transfer files to and from a mainframe, then we need to have a protocol which checks and corrects for transmission errors. There are many such protocols available for communications between PCs and mainframes and they are not unlike the hypothetical ACK/NAK protocol that we examined in section 6.3. The more common protocols are often provided as a standard feature of a Terminal Emulation software package that one buys for a PC.

One of the most prolific of the protocols used for transferring data between a mainframe and a PC is known as the "Kermit" protocol. The Kermit protocol was developed by Columbia University in the United States specifically as a means for transferring files between mainframes and PCs. The Kermit protocol is essentially an ACK/NAK protocol, where files are transmitted in blocks, and error checking is through Block Check Sum calculations. The software for Kermit is available for a range of different computer platforms (in addition to PCs) and is therefore widely used. A number of Terminal Emulation packages, available for PCs, provide Kermit as a standard feature.

Another protocol that has been developed for transferring data files between computer systems is the Ward Christensen system, referred to as "XMODEM". The XMODEM system was originally designed in 1977 as a protocol for transferring data between PCs, but has since been used for mainframe to terminal communication. Initially, the XMODEM system was a simple ACK/NAK protocol that used a single character Block Check Sum calculation for error detection, but more sophisticated versions utilising Cyclic Redundancy Check (CRC) polynomials have also become available.

Unfortunately, it is not always possible to use common protocols such as Kermit or XMODEM to transfer data to and from PCs. Many programmable devices, such as serial printers, CNCs and PLCs cannot be equipped with the necessary software that will allow this to occur. Many industrial devices use their own peculiar protocols and therefore one has to develop the corresponding PC software as described in section 6.3.

The worst-case scenario is where a device does not have its own protocol (with error checking) and cannot have such a protocol installed. Such devices often allow files or data to be simply "dumped" to or from remote devices through a serial port. In simple terms, file dumping means transmitting an entire file (character by character), from one device to another, without rules of protocol, error detection or correction.

File dumping is not a secure means of transferring files between devices and the possibility of error must always be considered when using such a technique. It is often used as a last resort and if the transmitted data is important, then some manual checking of received data must always be performed.

The most common example of file dumping, in the manufacturing environment, is where a CNC program, generated on a CAD system, needs to be down-loaded to an older style CNC machine. If the distance between the CAD system and the target controller is large, and noise-immune communications links are not in place, then a PC, running "Terminal Emulation" software can be used as a relay stage in the transmission process to minimise errors. This is shown in Figure 6.13.

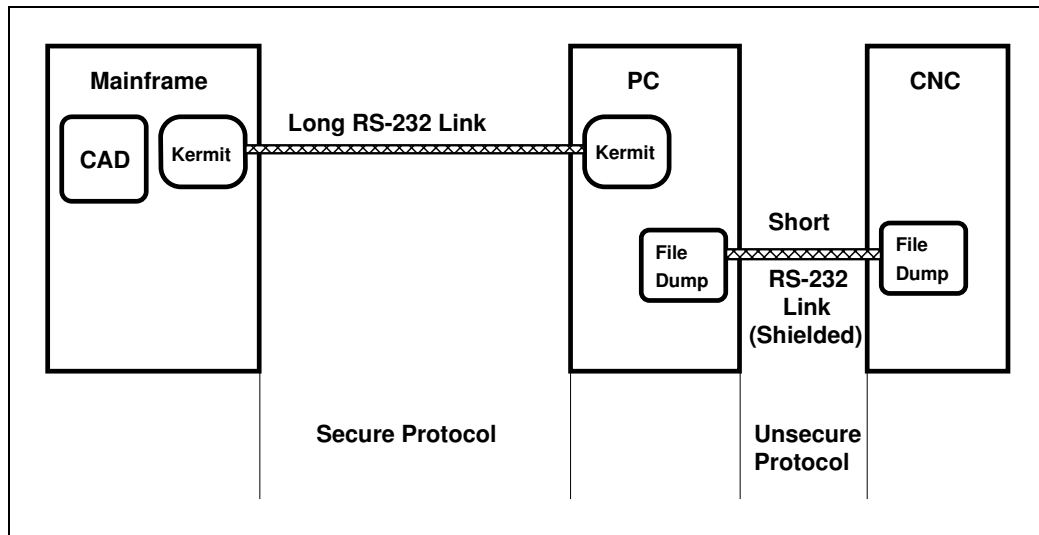


Figure 6.13 - Using Terminal Emulation to Minimise Errors in Data Transmission to Devices without Secure Protocols

The CNC machine program file is transferred from the CAD computer to the PC via a secure protocol such as Kermit, which can detect and correct communications errors. The PC is then located in close proximity to the target machine controller and the file "dumped" to the controller over a very short, shielded RS-232 link (with shielded connectors) in order to minimise chances of data corruption. Although this is clearly a "makeshift" solution, it may prove worthwhile for transferring long CNC program files that are generally related to the production of high cost products.

Terminal Emulation software has become an extremely valuable tool in the manufacturing environment. A Comprehensive Terminal Emulation package for a PC must therefore be capable of performing the functions described above and needs to contain the following modules:

- (i) Serial Port Parameter Set up
- (ii) Emulation of common dumb terminals
- (iii) Kermit Protocol
- (iv) XMODEM Protocol
- (v) File Dump procedures

Terminal emulation does not however, resolve the problem of transferring data according to specialised protocols, which must still be specially tailored for each application.

