

TopSpeed Modula-2



TopSpeed™ Modula-2

For IBM® Personal Computers and Compatibles

Communications Toolkit

Jensen & Partners International

LICENSE STATEMENT

Jensen & Partners International hereby grants you a non-exclusive license to use the TopSpeed Communications Toolkit. You may use the software on any computer, **provided that the software cannot possibly be used on more than one computer at the same time.** You may make copies of the software for the sole purpose of loading it into the computer on which you wish to use it, or for keeping a maximum of two backup copies.

You are furthermore granted the right to sell programs developed by you using the program modules provided in this package. You are not allowed to re-sell or give away any part of the source code provided with this product.

WARRANTY

Jensen & Partners International warrants the enclosed diskette to be an accurate copy of the master disk, and will replace any defective copy free of charge for a period of 60 days after purchase.

Jensen & Partners International hereby explicitly disclaims all other warranties whether express or implied, including without limitation, the implied warranties of merchantability and fitness for any particular purpose. Jensen & Partners International shall have no liability for consequential, incidental, exemplary or special damages, including lost profits.

Jensen & Partners International reserves the right to change the product at any time, without prior notice.

TopSpeed™ is a trademark of Jensen & Partners International.

IBM® is a registered trademark of the International Business Machines Corporation.

Microsoft® is a registered trademark of Microsoft Corporation.

Copyright © 1989 by Jensen & Partners International. All rights reserved.

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

Contents

1	Introduction	1
2	RS232 - Low Level Communications Driver	5
	An Example	8
3	File Transfer Modules	11
	Introduction	11
	Module FTSupport	14
	Xmodem File Transfer	17
	Ymodem and its Variants	22
	Xmodem-1k and Ymodem/Batch	22
	Ymodem-g	25
	Windowed Xmodem (WXModem)	28
	Kermit	32
4	Scripts	35
	Script Syntax and Commands	35
	Script Processing Modules	40
	An Example Script	42
5	Support Modules	45
	Module CommDemo	45
	Commands to the CommDemo program	46
	Module ComSetup	47
	Module Filenames	48
	Module Keyboard	49
	Module MenuLib	50
	Modules PRINTF/SPRINTF	53
	Module Timer	59
	Module CommUtil	61
	VT100/VT52/ANSI Terminal Emulation	66
	Appendix A: The 8250 UART	71
	Receive Buffer Register (RBR) - I/O Address: Base+0	71

Transmit Holding Register (THR) - I/O Address: Base+0	71
Divisor Latch Least (DLL) - I/O Address: Base+0	72
Divisor Latch Most (DLM) - I/O Address: Base+1	72
Interrupt Enable Register (IER) - I/O Address: Base+1	72
Interrupt Identification Register (IIR) - I/O Address: Base+2	73
Line Control Register (LCR) - I/O Address: Base+3	73
Modem Control Register (MCR) - I/O Address: Base+4	74
Line Status Register (LSR) - I/O Address: Base+5	74
Modem Status Register (MSR) - I/O Address: Base+6	74
Appendix B: Cabling Information	77
Index	81

Chapter 1

Introduction

Welcome to the TopSpeed Modula-2 Communications Toolkit — a collection of modules designed to help you with all applications that need to use the IBM PC serial port hardware. The toolkit includes low level communications drivers as well as support for high level file transfer protocols; it also contains a complete communications package whose features include:

- VT100,VT52 and BBS ANSI terminal emulation
- file transfer
- text logging and scripts for automatic logon to remote services.

Full source is provided for all modules.

The IBM PC provides support for serial communications in its BIOS, and PC-DOS makes use of this to provide its own support. However, you will probably find that the IBM serial communications support is practically useless except for the simplest of applications — because it is not interrupt driven, and has no support for software flow control.

The TopSpeed Communications Toolkit does not use the BIOS routines. Instead, the low-level, general-purpose serial driver supports interrupt driven I/O both for transmitting and receiving. The communication can be controlled with either hardware or software flow control.

Hardware flow control is accomplished by sending a signal on particular pins or wires. For example to acknowledge a transmission, a machine will activate a pin to signal Received Data. Sometimes the modems attached to the two machines will control the flow of information between the computers.

In software flow control, the machines indicate various things by sending particular characters. For example, a commonly used software flow control is XON/XOFF. In this system, the receiving machine will send XON (^Q) to indicate that it is ready to receive data, and will send XOFF (^S) to tell the transmitting machine to stop sending until an XON signal is sent.

The toolkit also provides modules that support the standard file transfer protocols:

- Xmodem
- Ymodem
- Ymodem Batch
- Ymodem-g Batch
- Windowed Xmodem, and
- Kermit

As a programmer you are assumed to be familiar with the TopSpeed Modula-2 language and environment, and also with the basic terms used in PC serial communications, although many of these terms are explained where appropriate in this manual. This book is divided as follows:

- | | |
|------------|--|
| Chapter 1 | The chapter you're reading, which provides a brief introduction to the package. |
| Chapter 2 | This chapter discusses a module for implementing a low-level communications driver. Such a driver enables you to select and use your PC's communications ports, detect whether you have made a connection, and move information to and from buffers used to store characters before transmitting them or after receiving them. |
| Chapter 3 | This chapter discusses the high level protocols that are used to specify the format of each transmission and to detect transmission errors, so the information can be sent again. The chapter includes modules to support several protocols. |
| Chapter 4 | This chapter discusses the toolkit's script language — for automating certain aspects of transmission. In addition to a summary of the script commands, the chapter includes an example script. |
| Chapter 5 | This chapter discusses several support modules that are used by some or all of the other modules. |
| Appendix A | This Appendix summarizes the registers on the chip used for serial transmissions. |

Appendix B

This Appendix shows the pin configurations and combinations needed to hook up various types of hardware.

RS232 - Low Level Communications Driver

Module RS232 provides all the procedures and functions you will need to control and use any of four serial ports on the PC. The definition module describes a collection of data types and constants which may be used in your own program. Appendix A contains a detailed description of the IBM PC hardware support for serial communications. In this chapter we'll describe the procedures provided by the module for controlling this hardware.

SelectPort

```
USES (crtio.h);
PROCEDURE SelectPort (Port:PortNo);
```

This procedure allows you to select which of four ports will be used as the default serial port in further calls to the RS232 module. If another port was already installed when SelectPort was called then a line drop is forced, and the newly selected port will need to be installed again after it has been initialized with the required data format parameters. The initialization code of module RS232 chooses COM1 as the default port. If you wish to use this port, then you won't need to call this procedure.

Support for COM1 and COM2 is standard across machines. The I/O addresses of COM3 and COM4 may vary on different machines, however. You may have to modify the source code in this case.

Chapter 2

RS232 - Low Level Communications Driver

Module **RS232** provides all the procedures and functions you will need to control and use any of four serial ports on the PC. The definition module describes a collection of data types and routines which may be used in your own program. Appendix A contains a detailed description of the IBM PC hardware support for serial communications. In this chapter we'll describe the procedures provided by the toolkit for controlling this hardware.

SelectPort

```
TYPE PortNos = [1..4];  
PROCEDURE SelectPort(Port:PortNos);
```

This procedure allows you to select which of four ports will be used as the default serial port in further calls to the **RS232** module. If another port was already installed when **SelectPort** was called then a line drop is forced, and the newly selected port will need to be installed again after it has been initialized with the required data format parameters. The initialization code of module **RS232** chooses COM1 as the default port. If you wish to use this port, then you won't need to call this procedure.

Support for COM1 and COM2 is standard across machines. The I/O addresses of COM3 and COM4 may vary on different machines, however. You may have to modify the source code in this case.

InitPort

```

TYPE Bits = [5..8];
   Stops = [1..2];
   ParityCodes = (None, Odd, dontuse1, Even, dontuse2, Mark, dontuse3, Space);

PROCEDURE InitPort (Baud:CARDINAL; DataBits:Bits; StopBits:Stops;
                   Parity:ParityCodes);

```

This procedure is used to initialize the chosen port with the required data format parameters. **Baud** is the required data rate in bits per second (with apologies to purists for the imprecise use of the word "baud"). **DataBits** is the number of data bits per word (normally 8). **StopBits** is the number of stop bits per word (normally 1). **Parity** is the required parity (usually **None**, but many networks require even parity). If **Even** parity is selected then it is usually necessary to reduce the number of data bits to 7.

There is no default setting for these parameters. Every program using the **RS232** module must call **InitPort** at least once.

SetFlowControl

```

TYPE FlowControls = (NoFlowControl, RtsCts, XonXoff);

PROCEDURE SetFlowControl (Which:FlowControls);

```

Flow control is automatically handled by the **RS232** module provided that one of the two flow control methods is chosen using this procedure. Your programs use either hardware flow control with **RTS/CTS**, or software flow control with **XON/XOFF**.

In **RTS/CTS** flow control, transmission is controlled by the states of two specific pins in the hardware. The **RTS** ("request to send") the **CTS** ("clear to send") pins effectively control who is sending and who is receiving. Actually, in most configurations, the **RTS** modem signal is simply toggled to control the transmission.

In software flow control, the receiver sends the **XOFF** character (normally **^S**) to stop the **remote system** (that is, the other computer), and **XON** (normally **^Q**) to start it again.

The Default is **RTS/CTS** flow control. If you do not wish to use any flow control then you must call this procedure with the parameter set to **NoFlowControl**.

The receiver interrupt routine inside module **RS232** handles flow control by checking how the receiver ring buffer fills up. When the buffer pointer reaches the "high water mark" (normally three quarters of the total buffer size), then a signal is sent to the remote system causing it to stop sending characters (for this to work the remote system must of course have been configured to recognize the signal). When the

receiver buffer empties out sufficiently to reach the “low water mark” — normally one quarter of the total buffer size — then another signal is sent to the remote system allowing it to start sending characters again. The type of signal sent in either case is dependent on the flow control method chosen.

Install

PROCEDURE Install;

This procedure installs the serial interrupt handler. This procedure must be called at least once before any attempt is made to read or write to the remote system. **Install** also asserts modem signals DTR (Data Terminal Ready) and RTS (Request To Send) which tells the remote system that you are “alive”.

CarrierDetect

PROCEDURE CarrierDetect():BOOLEAN;

This procedure tests the DCD (Data Carrier Detect) signal on the modem interface and returns **TRUE** if this signal is asserted. It is normally used to detect whether you are currently “online” to a remote system. This signal is most useful when the PC is attached to a physical modem and allows a communications program to detect when a dialup connection to a remote bulletin board system has been established.

UnInstall

PROCEDURE UnInstall;

This procedure forces a line drop (by dropping the DTR and RTS modem signals), and uninstalls the serial interrupt handler.

SerialWrite

PROCEDURE SerialWrite(b:ARRAY OF CHAR; Lnth:CARDINAL);

This procedure copies **Lnth** characters from the array **b** to the transmitter ring buffer. Once in the buffer, the characters are ready to be sent by the transmitter interrupt handler.

SerialRead

```
PROCEDURE SerialRead(VAR c:CHAR; t:CARDINAL):BOOLEAN;
```

This procedure is used to read characters from the receiver ring buffer. *t* is a timeout, and is the maximum time in tenths of seconds to wait when a character is not immediately available. The timeout may be zero, in which case the routine returns without waiting. The procedure returns **TRUE** if a character was actually read.

SetBreak

```
PROCEDURE SetBreak(Tenths:CARDINAL);
```

This procedure asserts the "break" condition on the modem interface for the desired length of time in tenths of seconds.

An Example

The following short example demonstrates how to use the features of module **RS232** to implement a (very) dumb terminal program. Although very simple it would be sufficient to, for example, dial up a remote BBS (Bulletin Board System). The example assumes that the defaults of port COM1 and RTS/CTS flow control are acceptable, and that 1200 baud is possible for the modem interface. Pressing **F10** causes the program to hang up the line and exit.

```
MODULE MiniTerm;
IMPORT RS232, Window, IO;
VAR c:CHAR;
BEGIN
  Window.Clear;
  RS232.InitPort(1200, 8, 1, RS232.None);
  RS232.Install;
  LOOP
    WHILE RS232.SerialRead(c, 0) DO (* test remote *)
      IO.WrChar(c)
    END;
    IF IO.KeyPressed() THEN
      c := IO.RdKey();
      IF c=0C THEN (* special key? *)
        c := IO.RdKey();
        IF c=CHR(68) THEN (* function key 10 *)
          EXIT
        END
      ELSE
        RS232.SerialWrite(c, 1);
      END
    END
  END
```



```
END  
END;  
RS232.UnInstall;  
END MiniTerm.
```

Chapter 3

File Transfer Modules

Introduction

File transfer is the task of sending stored information from one computer system to another. The two computer systems need not be of the same type. In fact, the two systems involved may be entirely unlike one another. In such a case, the RS-232 interface may be the only link between them.

The simplest file transfer method is "ASCII Transfer." This involves little or no intelligence on the part of the sender, and involves simply logging received text to a file on the receiving system. There are many problems with ASCII file transfer:

- it works only with ASCII text containing no unusual control characters
- it does not check the integrity of the file as it is being sent and received
- it is not normally possible to transfer raw binary files using plain ASCII file transfer

Generally, sender and receiver will coordinate their work by using a protocol to guide the transmission. A protocol is a means of identifying components of a transmission, as well as a way of making the interaction between sender and receiver run more smoothly. You'll find examples of several protocols in this chapter.

If the sender and receiver both agree on a protocol, it is possible to overcome these problems. There are many standard protocols which have been adopted, and the Toposend Communications Toolkit provides support for the most common of these.

Chapter 3

File Transfer Modules

Introduction

File transfer is the task of sending stored information from one computer system to another. The two computer systems need not be of the same type. In fact, the two systems involved may be entirely unlike one another. In such a case, the RS-232 interface may be the only link between them.

The simplest file transfer method is "ASCII Transfer." This involves little or no intelligence on the part of the sender, and involves simply logging received text to a file on the receiving system. There are many problems with ASCII file transfer:

- it works only with ASCII text containing no unusual control characters
- it does not check the integrity of the file as it is being sent and received.
- it is not normally possible to transfer raw binary files using plain ASCII file transfer

Generally, sender and receiver will coordinate their work by using a **protocol** to guide the transmission. A protocol is a means of identifying components of a transmission, as well as a way of making the interaction between sender and receiver run more smoothly. You'll find examples of several protocols in this chapter.

If the sender and receiver both agree on a protocol, it is possible to overcome these problems. There are many standard protocols which have been adopted, and the TopSpeed Communications Toolkit provides support for the most common of these.

Although some protocols are better than others, all of the many protocols have several things in common.

- The file to be transferred is broken into *chunks* of a particular size. These chunks have many names, including “blocks,” “packets,” and “frames.” The task of transferring a file then becomes the task of sending one packet at a time until the file transfer is complete.
- Each packet sent has an information “frame” surrounding it. This frame is useful to the protocol, and normally consists of a sequence number used to identify any packet uniquely, and a “frame check sequence” (FCS) which is used to protect the integrity of the packet. The FCS is often a checksum, but is preferably a CRC (cyclic redundancy check).
- This check is accomplished by having the sender calculate a frame check and add the result of the calculation to the information transmitted. The receiver examines the received packet and recalculates the frame check using the received data. If the receiver’s result agrees with the sender’s then the packet is accepted and the data are written to disk, otherwise the data in the packet are assumed to be incorrect, and must be retransmitted. This continues until every packet in the file is successfully transmitted.
- The start of each frame usually consists of a one or more “sync bytes” which simply tell the receiver that a new frame is about to start. The receiver would normally ignore any bytes it receives before it gets the sync bytes.

Here is a typical layout for a single packet.

Sync	Seq No	<i>n</i> Data Bytes	FCS
------	--------	---------------------	-----

This layout is similar to that used by the simpler protocols, such as Xmodem. Such packets are a constant length. More complex protocols may allow packets to have a variable length. In that case, the frame information may need to include a length byte, or may use some unique marker to denote the end of a packet. Additionally, some protocols use packets to send information *about* the file (such as its name and size), instead of just information *from* the file. This may require giving each packet a “type” byte, to tell the receiver how to interpret the packet.

Once the format of a single packet has been specified, there is still a good deal of variation in how those packets are transmitted to the receiving system. Simple Xmodem-type protocols (sometimes called “start-stop” or “half duplex” protocols), send a single packet and wait for an acknowledgement (which may be negative). Having received the acknowledgement, the protocol either sends the last packet again, or sends the next one. (Half duplex transmission is essentially one way. Only one computer can be sender at a time. Thus, while one machine is sending, the other must be receiving. The two machines may trade roles at a later time.)

At high bit rates or over slow networks the time needed by Xmodem to “wait for acknowledge” can become a significant portion of the total time required for the file transfer — and time, as any glance at a phone bill will tell you, is money.

There are three possible solutions to this problem:

- use larger packets
- use a windowing protocol, or
- use a streaming protocol

The second and third solutions are faster because they are “full duplex” protocols. (In full duplex protocols, both computers can be senders and receivers at the same time. Essentially, full duplex protocols can carry information in both directions at the same time.)

Larger packets can reduce the per packet overhead by reducing the number of packets that need to be sent. This also has drawbacks however. If a packet is found to be corrupted then it must be sent again, and it takes longer to retransmit larger packets. Thus, large packets are appropriate only where line noise is typically rare.

Ymodem and its variants are examples of “large packet” protocols. An Xmodem packet sends 128 bytes of user data, while a Ymodem packet normally sends 1024 bytes, although standard Ymodem allows for a mixture of 1024 and 128 data-byte packets.

A windowing protocol can send several packets before it must wait for an acknowledgement. The number of packets it can send is called the protocol’s window size. You can think of Xmodem as a degenerate windowed protocol, with a window size of one. At first glance it may seem that all we’ve done is increase the size of the packet. Notice, however, that the sender can send up to “window-size” packets without waiting for an acknowledgement. If the receiver simply acknowledges each packet as it comes in, then there will never be enough unacknowledged packets to stop transmission. Thus, with such a protocol the sender need not stop sending until the file transfer is complete. WXXmodem (Windowed Xmodem) is an example of a windowed protocol. Its window size is always four packets.

A streaming protocol does not use acknowledgments at all, unless something was wrong with the last packet sent. When that happens the receiver sends a negative acknowledgement to the sender, interrupting it and forcing it to start again from the bad packet. Depending on how long the turnaround is, the sender may have sent several more packets before it notices that it must go back to the bad one. None of the protocols supplied with the TopSpeed Communications toolkit are streaming protocols, although YModem-g might perhaps be considered a special case of a streaming protocol.

YModem-g is an example of a new breed of file transfer protocol in which the need for integrity checking has been decreased as new error correcting modems have been introduced. (Many of these modems support the MNP link level error correction protocol.) When such modems are used the integrity checking part of the protocol has been taken over by the modem, so that the file transfer protocol itself has a lot

less to worry about. YModem-g does not bother with acknowledgements because the data is assumed to be correct anyway. Packets are still assembled with frame checks, but a detected error will cause the entire file transfer to be aborted, since it normally means that something is wrong with the modem interface.

The TopSpeed Communications Toolkit makes available various file transfer protocols. The following sections discuss the toolkit's support for these protocols.

Module FTSupport

The Communications toolkit provides separate modules for each of the different file transfer protocols that it supports. Many of the tasks involved in file transfers are shared by all of the protocols, however. Module **FTSupport** provides a base module that supports these shared functions.

FTSupport - Constants, Types and Vars

```
CONST (* Characters common to all Xmodem/Ymodem Variants *)
  ACK = CHR(6);
  NAK = CHR(21);
  SOH = CHR(1);
  STX = CHR(2);
  EOT = CHR(4);
  CAN = CHR(24);

  (* Extra chars required by some of the variants *)
  SYN = CHR(22);
  CRCREQ = 'C';      (* Request CRC frame check instead of checksum *)
  WNDREQ = 'W';      (* Request Windowed Xmodem *)

VAR Aborted:BOOLEAN;
    Errors:CARDINAL;
    SizeInBytes,Transferred:LONGCARD;
    XferMode,LastMessage:ARRAY[0..30] OF CHAR;
    FileSpec:ARRAY[0..80] OF CHAR;
    CRCTable:ARRAY[0..255] OF CARDINAL;
    FBuff:ARRAY[0..1025] OF CHAR;
```

The constants and variables declared in **FTSupport.DEF** are used by most, if not all, of the file transfer protocols supported. The character constants declared have special meaning to many of the protocols.

The variables provide information that a program may find useful during the course of a file transfer.

Aborted is true if the user pressed **[Esc]** during the file transfer; this variable will also be true if the remote (that is, receiving) end cancelled the transmission.

Errors is used to count the number of errors encountered while trying to transmit the current packet.

SizeInBytes is the size of the file being transferred. This will be 0 if the size is not known.

Transferred represents the actual number of bytes successfully transferred so far.

XferMode depends on the protocol being used, but is normally "CRC" or "CHK," meaning that the frame check type is either a CRC or checksum.

LastMessage contains general information about the state of the file transfer — for example, the cause of the last error, or perhaps that the last file transfer is completed.

FileSpec is the complete name of the file being transferred.

CRCTable contains the data for the table driven CRC algorithm.

FBuff contains file data which will become, or has been extracted from, a packet.

FTSupport also contains several procedures and functions of use to the modules supporting the individual file transfer protocols.

OpenStatusWindow

```
PROCEDURE OpenStatusWindow(Title:ARRAY OF CHAR);
```

This procedure opens a window on the screen. This window is used to display the status of the file transfer while it is in progress.

NewFilename

```
PROCEDURE NewFilename;
```

This procedure updates the **Filename** field in the file transfer status window. The variable **FileSpec** is examined and the "tail" of the name (that is, the name *not* including the drive and path) is displayed.

CompleteFilename

PROCEDURE CompleteFilename;

This procedure is called only when downloading a file — that is, transferring the file from a remote source to your computer. The procedure takes the variable **FileSpec**, which is assumed to contain a filename as supplied either by the user or by the remote system. It then concatenates this name with the default pathname for downloaded files to form a completed file name. If the procedure discovers that the file name already has a path then **FileSpec** will be left unchanged. This can happen only when the user has supplied the name for the downloaded file.

UpdateStatus

PROCEDURE UpdateStatus;

This procedure updates the file transfer status window with the contents of the various status variables defined in **FTSupport.DEF**. The variables are kept up to date by the individual protocol modules, who then call **UpdateStatus** at regular intervals to update the display.

CloseStatusWindow

PROCEDURE CloseStatusWindow;

As its name implies, this procedure closes the file transfer status window. It also causes an alarm to be sounded, alerting the user (who may not be watching) that the transfer(s) are complete. If the boolean variable **Aborted** is true when procedure **CloseStatusWindow** is called, then a message (“aborted”) is displayed.

Cancel

PROCEDURE Cancel;

This procedure causes a sequence of eight ^X characters (ASCII CAN) to be sent. Most of the protocols use this as a signal to the remote to stop sending and to abort the file transfer. The eight CANs are followed by eight backspace characters to clean up unused CANs.

AbortRequested

```
PROCEDURE AbortRequested():BOOLEAN;
```

This function checks the keyboard status and reads a character if one is available. If this character is Escape (ordinal 27), then the boolean variable **Aborted** is set to **TRUE**. The function returns the state of the **Aborted** variable.

FileExists

```
PROCEDURE FileExists(VAR filename:ARRAY OF CHAR):BOOLEAN;
```

This function returns **TRUE** if the file **filename** exists, and **FALSE** if it does not.

FlushInput

```
PROCEDURE FlushInput;
```

This procedure throws away any characters that have been received from the remote, but that have not yet been processed.

ASCIISend

```
PROCEDURE ASCIISend(VAR Filename:ARRAY OF CHAR):CARDINAL;
```

The ASCII file transfer protocol is the only one which is not given its own module, mostly because the protocol is quite trivial. **Filename** contains the name of the file you wish to send. The function returns an error code that should be zero if the file was transferred successfully.

In the next few sections, you'll find discussions of the modules for handling the various transfer protocols supported by the TopSpeed Communications Toolkit.

Xmodem File Transfer

This section first describes the Xmodem protocol, and then discusses the implementation provided in the communications toolkit.

Xmodem is one of the oldest and simplest of file transfer protocols. It was designed by Ward Christensen in 1977 and was placed in the public domain. The protocol has gained in popularity over the years, to the point that now it is difficult to find a BBS or communications package that does not support Xmodem.

The reason for the protocol's popularity is almost certainly its simplicity. Xmodem does have its drawbacks, however. The main ones are that Xmodem requires a completely transparent eight-bit medium, and that its performance on timesharing systems and packet switching networks can be poor. Given these shortcomings, however, Xmodem does its job very well.

A standard Xmodem packet is laid out as follows:

SOH	Seq	Seq Compl	128 data bytes	checksum
-----	-----	-----------	----------------	----------

SOH is the ASCII character, ordinal 1. **Seq** is a sequence or block number, starting at one for the first packet, and incrementing by 1 (modulo 256) for every new packet sent. **Seq Compl** is the complement of the block number; it can be calculated using the relation

$$\text{Compl} = 255 - \text{Seq}$$

and is used as an integrity check for the block number.

Next, the packet contains 128 bytes of data from the file being transferred. Finally, there is a **checksum byte**, which is a modulo 256 sum of the ordinal values of the 128 data bytes.

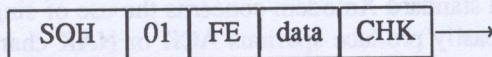
An Xmodem file transfer is entirely controlled by the receiver. The following is an example of a simple (no errors) Xmodem session, starting from the point where both sender and receiver are started on their respective systems.

SENDER

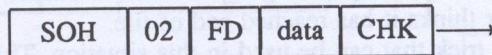
RECEIVER

wait ten seconds...

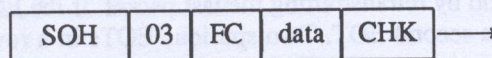
← NAK



← ACK



← ACK



← ACK

EOT →

← ACK

(session ends)

The signals that the receiver is sending are:

- NAK (^U, ordinal 21) which means "The last packet was bad." This signal is also used to tell the sender to transmit the first block.
- ACK (^F, ordinal 6), which means that "The last packet was good."

The signals sent by the sender are the packets and, if all packets have been sent, EOT (^D, ordinal 4) which means "End of File has been reached." The receiver ACKs that signal, and the file transfer is complete.

Over the years several weaknesses of Xmodem have been pointed out. The checksum is not a particularly good way of protecting data integrity. Because it is a simple sum of the data it is quite possible for double bit errors to pass undetected. (For example, noise might make the least significant bit of one byte into a zero when it was one, and then change the same bit in a later byte to one when it was supposed to be zero — the sum remains the same, so the errors would pass undetected). The solution to this problem is to use a CRC instead of a checksum. To do this, however, the Xmodem implementations at either end must be capable of using CRCs.

When CRC checking is to be used, the initial exchange between sender and receiver is altered slightly. The receiver sends 'C' instead of NAK, to tell the sender to transmit the first packet. If the sender supports Xmodem/CRC it will reply by sending the first packet. If no packet is sent, the receiver will try sending 'C' twice more, before sending NAK (that is, dropping down to Xmodem/Checksum). (See the example which is included after the next few paragraphs.)

Another objection to standard Xmodem is the long pause (ten seconds) before the receiver sends its first NAK (or 'C'). This was supposed to allow time for the sender to get his machine ready. However, there is no reason why the receiver should not simply send the NAK (or 'C') right away, and retry every few seconds until a packet arrives, or some timeout is reached.

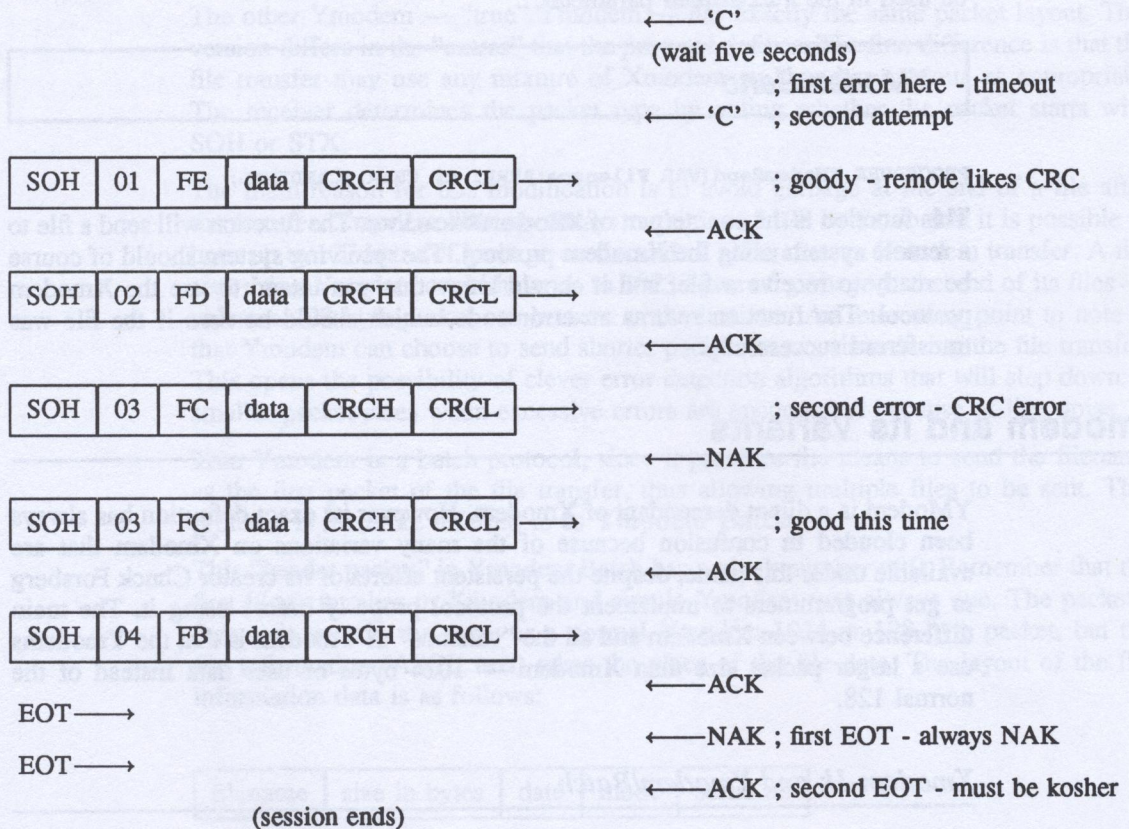
Yet another problem with standard Xmodem concerns the use of single character responses. Line noise can easily produce spurious ACK or NAK characters. Unfortunately, nothing has been done about this problem. The EOT character suffers from the same weakness. This is especially damaging as an EOT causes the file transfer to abort since the receiver thinks it has reached end of file.

However, there is a good trick that can be used in this situation. The solution is for the receiver to always NAK the first EOT that it receives. If the EOT was spurious then the sender will respond by retransmitting the last packet. If the EOT was genuine then the sender will send a second EOT. Two spurious EOTs in a row are extremely unlikely, so such a transmission can usually be trusted.

Here is another example of an Xmodem session, this time demonstrating all of the above improvements, and this time also adding a couple of faults. . .

SENDER

RECEIVER



Having described the Xmodem protocol in detail the module itself may be a bit of an anticlimax. There are only two very simple procedures defined in **XModem.DEF**.

XModemReceive

```
PROCEDURE XModemReceive (VAR Filename:ARRAY OF CHAR):CARDINAL;
```

This function will receive a file using the Xmodem protocol, with either checksum or CRC (although the function will attempt to negotiate for CRC). The function returns an error code whose actual value is important only to the internal code of the module, except that the value should be zero if the file was received successfully.

The Xmodem protocol does not make provision for passing the name of the file to be transmitted to the other system. Instead it is necessary to pass the name that should be used in the **Filename** parameter.

XModemSend

PROCEDURE XModemSend (VAR Filename:ARRAY OF CHAR):CARDINAL;

This function is the counterpart of **XModemReceive**. The function will send a file to a remote system using the Xmodem protocol. The receiving system should of course be ready to receive a file, and it should know that you intend to use the Xmodem protocol. The function returns an error code, which should be zero if the file was transferred successfully.

Ymodem and its Variants

YModem is a direct descendant of Xmodem. However its exact definition has always been clouded in confusion because of the many variations on Xmodem that are available under this name, despite the persistent efforts of its creator Chuck Forsberg to get programmers to implement the protocol properly before using it. The main difference between Xmodem and all the "versions" of Ymodem is that the Ymodems use a larger packet size than Xmodem — 1024 bytes of user data instead of the normal 128.

Xmodem-1k and Ymodem/Batch

There are two main variants of this protocol which are consistently called Ymodem. The simplest of these is merely Xmodem with the few minor changes required to use 1024 byte packets. The layout for this kind of Ymodem packet is shown in the following illustration:

STX	Seq	Seq Compl	1024 data bytes	FCS
-----	-----	-----------	-----------------	-----

Note that the sync byte has changed from SOH (Xmodem) to STX (^B, ordinal value 2), and the user data is 1024 bytes. FCS is the frame check sequence, and can be a CRC or checksum, using exactly the same rules as the improved Xmodem. Apart from the fact that it uses the modified packet layout, the file transfer algorithm

is indistinguishable from Xmodem. In fact Forsberg prefers to call this protocol “Xmodem-1k”.

The other Ymodem — “true” Ymodem — has exactly the same packet layout. This version differs in the “extras” that the protocol defines. The first difference is that the file transfer may use any mixture of Xmodem or Ymodem packets as appropriate. The receiver determines the packet type by noting whether the packet starts with SOH or STX.

The main reason for this modification is to avoid garbage at the end of a file after the transfer. Xmodem files are always multiples of 128 bytes, so that it is possible to have up to 127 bytes of garbage on the end of a file after an Xmodem transfer. A file sent using Ymodem could have up to 1023 bytes of garbage at the end of its files — unless a 128-byte packet can be sent as the last one. An interesting point to note is that Ymodem can choose to send shorter packets at any time during the file transfer. This opens the possibility of clever error detection algorithms that will step down to smaller packet sizes when excessive errors are encountered because of line noise.

True Ymodem is a batch protocol, since it provides the means to send the filename as the first packet of the file transfer, thus allowing multiple files to be sent. This protocol is therefore referred to as **Ymodem Batch**.

This “header packet” in Ymodem Batch has a block number of 0. Remember that the first block number in Xmodem and simple Ymodem was always one. The packet’s format is exactly the same as a normal Ymodem 1024 or 128 byte packet, but the file information (ASCII text) takes the place of the file data. The layout of the file information data is as follows:

filename	size in bytes	date	mode	serial no
----------	---------------	------	------	-----------

The **filename** field is a NUL terminated ASCII string. The remaining fields are ASCII strings separated by spaces. All fields after the filename are optional. Unused spaces in the packet data field are reserved for future expansion and should be NUL filled. A blank **filename** field instructs the receiver that there are no more files to be transferred.

The **filename** contains the name of the file being sent. It may contain a pathname (which may be ignored by the receiver), but the pathname must use Unix style separators ‘/’ instead of the DOS style ‘\’. The JPI implementation of Ymodem Batch does not send file paths.

The “size in bytes” field is self explanatory. The sender is allowed to supply the file size without supplying any of the remaining fields. This is exactly what the TopSpeed

implementation does, since all of the remaining fields are really only relevant in a Unix environment.

The "date" field is the date the file was last modified, based on the number of seconds since the first of January, 1970, GMT. For some reason, Forsberg chose to use an octal representation for this field. DOS does not make it easy to convert GMT to local time, so the TopSpeed implementation does not make use of this field.

The "mode" field refers to a Unix file mode. This is obviously inapplicable to DOS.

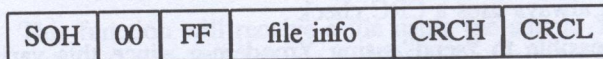
The serial number is a string of octal digits. Again the TopSpeed implementation will not use this field, and will ignore it if it is included by the sending program.

The following illustration shows a simple example of a Ymodem/Batch session. For simplicity, the session assumes no errors.

filename	size in bytes	date	mode	serial
testfile	1024	19700101	0644	000000

SENDER

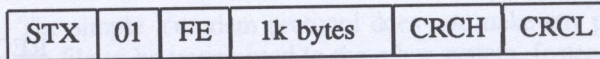
RECEIVER



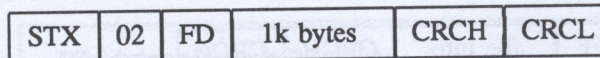
← 'C'

← ACK

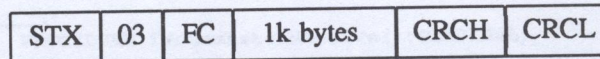
← 'C'



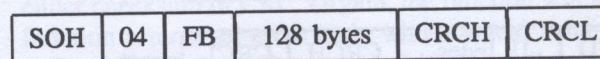
← ACK



← ACK



← ACK



← ACK

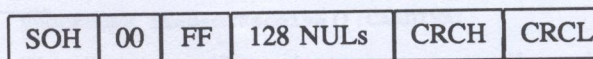
EOT →

← NAK ; first EOT

EOT →

← ACK

← 'C'



← ACK

(session ends)

Ymodem-g

There is one final variation on Ymodem, called Ymodem-g. This protocol is intended for use on error free connections (for example, on an error corrected modem link). The only difference between Ymodem-g and the other Ymodem versions is that no ACKs are sent in Ymodem-g, except for the ACK reply to EOT. Any error will abort a Ymodem-g transfer.

Ymodem-g is requested by the receiver using exactly the same technique used to request CRC transfer, except that the request character used is 'G'. This has two consequences:

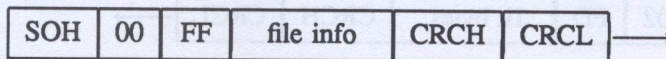
Ymodem-g always uses a CRC check

It is not possible to "send" using Ymodem-g, since this variant can only be requested by the receiver.

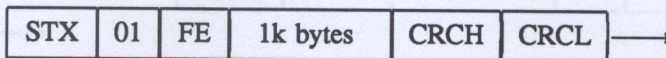
The following example transfers the same information as the previous Ymodem example, but uses Ymodem-g instead of the more usual Ymodem/Batch.

SENDER

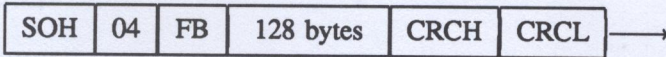
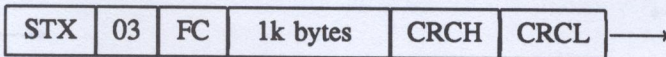
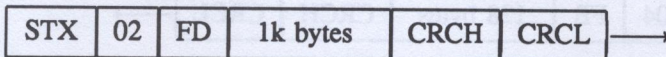
RECEIVER



← 'G'



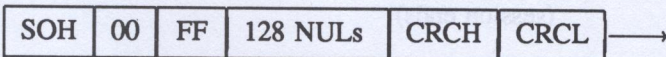
← 'C'



EOT →

← ACK

← 'C'



(session ends)

As in earlier examples, all of the complexity involved in transferring files between machines is hidden within the Ymodem module. The following sections describe the procedures declared in **YModem.DEF**.

YModemReceive

```
PROCEDURE YModemReceive (VAR Filename:ARRAY OF CHAR):CARDINAL;
```

This function will receive a file using the simple Ymodem (Xmodem-1k) protocol, and using either checksum or CRC (although the function will attempt to negotiate for CRC). The function returns an error code whose actual value is important only to the internal code of the module, except that the value should be zero if the file was received successfully.

The simple Ymodem protocol does not make any provision for passing the name of the file to be transmitted to the other system. Instead it is necessary to pass the name that should be used in the **Filename** parameter.

YModemBatchReceive

```
PROCEDURE YModemBatchReceive():CARDINAL;
```

This function will receive one or more files using the Ymodem/Batch protocol, using either checksum or CRC. (Again, the function will attempt to negotiate for CRC.) The function returns an error code whose actual value is important only to the internal code of the module. The code is 0 if all the files were transmitted successfully, and a nonzero value to indicate an error. The file will be stored under the filename supplied by the sender.

YModemGReceive

```
PROCEDURE YModemGReceive():CARDINAL;
```

This function will receive one or more files using the Ymodem-g protocol. The function returns an error code of 0 if the file was transmitted successfully, and a nonzero value if there was an error. This value is important only to the internal code of the module. The file will be stored under the filename supplied by the sender.

YModemSend

PROCEDURE YModemSend (VAR Filename:ARRAY OF CHAR):CARDINAL;

This function will send a file using the simple Ymodem (Xmodem-1k) protocol. The function returns an error code whose actual value is important only to the internal code of the module, except that the value should be zero if the file transfer was successful.

YModemBatchSend

PROCEDURE YModemBatchSend (VAR Wild:ARRAY OF CHAR):CARDINAL;

This function will send file(s) using the Ymodem/Batch protocol. The function returns an error code whose actual value is important only to the internal code of the module, except that the value should be zero if the file was received successfully. The **Wild** parameter is any valid DOS file specification. You can use the DOS wildcard characters, '*' and '?' to specify and send a group of files.

Windowed Xmodem (WXModem)

WXmodem is a simple extension of Xmodem that greatly improves the protocol's performance, especially on slow networks. WXmodem is still rare except on Opus BBS systems; however, WXmodem is catching on, mostly because it is very easy to implement (like Xmodem). The WXModem protocol may be the easiest of the many windowing protocols.

WXmodem packets are almost indistinguishable from those of Xmodem/CRC, as shown in the following illustration.

SYN	SOH	Seq	Seq Compl	128 data bytes	CRCH	CRCL
-----	-----	-----	-----------	----------------	------	------

As you see, the only changes are the SYN character at the front (^V, ordinal 22), and the two CRC bytes at the end. WXmodem always uses a CRC. (The WXmodem definition actually says that there can be one or more SYN characters at the start of the packet, and that two are recommended.)

The purpose of the SYN character is to improve synchronization. With normal Xmodem there is a problem if an error is detected in mid-packet. The protocol must enter a "flush" mode and throw away all characters, unless it detects the start of a new

packet — which can be tricky. WXmodem makes it easier to detect the start of a packet by using the SYN char for synchronisation at the start of the packet, and by making the character illegal anywhere else.

But suppose a SYN crops up in the data you want to send, or elsewhere in the frame? You can do what is called escaping to get around the preceding restriction. In this way, you can send the information in a legal manner.

In the case of WXmodem (and most other protocols which use the technique) *escaping* involves replacing the illegal single character with a two-character code that is legal. This code will be picked up by the receiving program and the original character can be restored.

The escape coding that WXmodem uses for illegal characters is fairly standard. The first character in each code sequence is DLE (^P, ordinal 16), and the next character is the illegal control character with 64 added to its ordinal value, thus making it a printable character. (In actual fact, the character has been XORed with 64 to get the desired value.)

For example, to use the SYN character (^V, or ordinal 22) anywhere except at the start of a packet, the protocol would send DLE followed by 'V' ($22 + 64 = 86 = 'V'$). The receiving program spots the DLE and subtracts 64 from the following character, thus restoring the SYN character's original value: $V - 64 = 86 - 64 = 22 = ^V$. Again, you can get the same result by XORing the character with 64, thus toggling the "control" bit.

SYN is not the only character that needs to be sent by escaping in this way. For example, you may already have noticed the need to escape DLE. WXmodem also escapes the XON and XOFF characters (^Q and ^S), so that WXmodem can be used over connections that use software flow control.

That finishes the discussion of the packet layout. WXmodem also changes the ACK and NAK responses used by Xmodem. In WXmodem, each of these becomes a two-byte sequence, with the second byte being the number of the packet being ACKed or NAKed. (The "packet number" used is not exactly the block number sent in the packet frame; rather, the number is the block number modulo 4.) If the sender receives an ACK or NAK for a particular block, the sender assumes that all of the previous blocks were received correctly. Thus, the receiver is not required to respond to every block — although it should try to do so, since fast responses reduce the chances of the window closing.

The ACK or NAK responses to the EOT signal are the only exceptions to this two-byte format. These responses remain single-character replies.

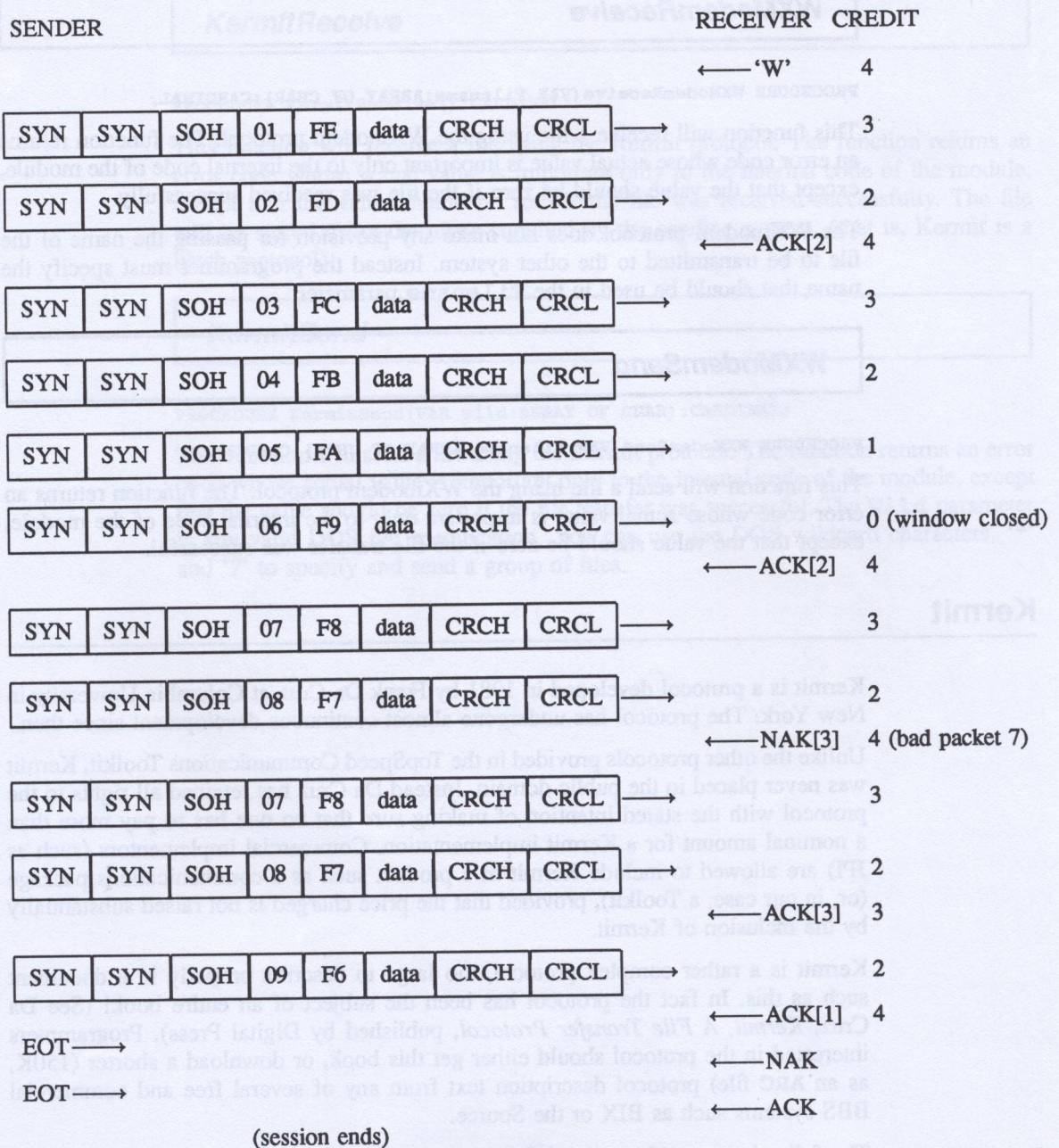
WXmodem is a "windowing" protocol with a window size of four. This means that the protocol can at most have four packets unACKed at any time. If this limit is reached then WXmodem will wait for an ACK before continuing. (In such a situation, it is said that the "window is closed.") In the sample WXmodem session below you'll

find the concept of "credit," which refers to the maximum number of packets that can be sent before the window is closed at that time. When the credit is 0, the sender must wait for an ACKnowledgement from the receiver.

Showing a sample WXmodem session is difficult, because the notation used so far really implies a half duplex protocol — in which only one machine is sending at any time. WXmodem, on the other hand is a full duplex protocol, since both machines can be sending at the same time. For example, the receiver can send acknowledgements as each packet comes in. If the window is still open, the sender will be transmitting the next packet while this acknowledgement is being sent.

Despite this, the same notation must be used. Keep in mind, however, that WXmodem does not stop and wait for a reply unless the window closes. Rather, the reply sent interrupts the sender who adjusts the credit accordingly (or sends previous packets again, whichever is required).

A final point to note is that the receiver requests Windowed Xmodem by using an initial 'W' similar to the request for CRC checking in normal Xmodem. This is why WXmodem always uses CRC.



The following descriptions summarize the routines declared in `WXModem.DEF`.

WXModemReceive

```
PROCEDURE WXModemReceive (VAR Filename:ARRAY OF CHAR):CARDINAL;
```

This function will receive a file using the WXmodem protocol. The function returns an error code whose actual value is important only to the internal code of the module, except that the value should be zero if the file was received successfully.

The WXmodem protocol does not make any provision for passing the name of the file to be transmitted to the other system. Instead the programmer must specify the name that should be used in the **Filename** parameter.

WXModemSend

```
PROCEDURE WXModemSend (VAR Filename:ARRAY OF CHAR):CARDINAL;
```

This function will send a file using the WXmodem protocol. The function returns an error code whose actual value is important only to the internal code of the module, except that the value should be zero if the file transfer was successful.

Kermit

Kermit is a protocol developed in 1981 by Frank Da Cruz at Columbia University in New York. The protocol has undergone almost continuous development since then.

Unlike the other protocols provided in the TopSpeed Communications Toolkit, Kermit was never placed in the public domain. Instead Da Cruz has retained all rights to the protocol with the stated intention of making sure that no one has to pay more than a nominal amount for a Kermit implementation. Commercial implementors (such as JPI) are allowed to include Kermit in a product such as a communications package (or, in our case, a Toolkit), provided that the price charged is not raised substantially by the inclusion of Kermit.

Kermit is a rather complex protocol, too large to describe properly in a document such as this. In fact the protocol has been the subject of an entire book! (See Da Cruz, *Kermit, A File Transfer Protocol*, published by Digital Press). Programmers interested in the protocol should either get this book, or download a shorter (150K, as an ARC file) protocol description text from any of several free and commercial BBS systems such as BIX or the Source.

The following procedures are defined in **Kermit.DEF**.

KermitReceive

```
PROCEDURE KermitReceive():CARDINAL;
```

This function will receive a file using the Kermit protocol. The function returns an error code whose actual value is important only to the internal code of the module, except that the value should be zero if the file was received successfully. The file will be stored using the name supplied by the sending system, (that is, Kermit is a batch protocol).

KermitSend

```
PROCEDURE KermitSend(VAR wild:ARRAY OF CHAR):CARDINAL;
```

This function will send a file using the Kermit protocol. The function returns an error code whose actual value is important only to the internal code of the module, except that the value should be zero if the file transfer was successful. The **Wild** parameter is any valid DOS file specification. You can use the DOS wildcard characters, '*' and '?' to specify and send a group of files.

Script Syntax and Commands

The script system is quite simple. The main elements of a script file are commands. These commands may take arguments of various sorts. The commands can be controlled using IF and SWITCH statements. (You'll learn more about such control elements in the next chapter.)

The following shows a systematic definition of the script language:

Chapter 4

Scripts

In communications, a script is normally used to automate the process of dialing up a remote system, although it also can be used for controlling a communications package in other ways. A **script** is a file containing the commands and information needed to accomplish certain aspects of a communication automatically.

The script language provided with the TopSpeed Communications Toolkit is intended to be roughly compatible with that used by the popular ProComm shareware communications package. However, unlike ProComm, the TopSpeed script language is compiled rather than interpreted.

Script source files have the extension *.CMD. The first 10 characters of any identifier are significant, and strings have a maximum length of 80 characters. Numbers have a maximum value of 32767 (although particular commands may specify a lower limit).

Script Syntax and Commands

The script syntax is quite simple. The main elements of a script line are commands. These commands may take arguments of various sorts. The commands can be controlled using **IF** and **SWITCH** statements. (You'll learn more about such control elements in the next chapter.)

The following shows a systematic definition of the script language:

- **script** ::= **command** { **command** }
- **command** ::= **simple_cmd** | **if_stmt** | **switch_stmt**
- **simple_cmd** ::= [**label** ":"] **ident** { **args** }
- **if_stmt** ::= "IF" **condition** { **command** }
["ELSE" { **command** }]
["ENDIF"]
- **condition** ::= [NOT] **flag**
- **flag** ::= "SUCCESS" | "FAILURE" | "FOUND"
| "CONNECTED" | "WAITFOR"
- **switch_stmt** ::= "SWITCH" **string_var**
case { **case** }
"DEFAULT" **command**
"ENDSWITCH"
- **case** ::= "CASE" **string_lit**
{ **command** }
"ENDCASE"
- **label** ::= **ident**
- **args** ::= **argument** { **separator argument** }
- **separator** ::= " " | ","
- **ident** ::= **Letter** { **Letter** }
- **argument** ::= **number** | **string** | **string_lit**
- **number** ::= **digit** { **digit** }
- **string** ::= **string_var** | **string_lit**
- **string_lit** ::= "" { **char** } ""
- **string_var** ::= "S" **digit** (eg S0, S1)

The condition flags are tested using the IF statement.

- The "SUCCESS" and "FAILURE" flags are complements of each other, and they reflect the success of the last script command to be executed. Only certain commands affect the status of these flags, however. Those that do are identified in the list below.
- "FOUND" is the result of a FIND command, as described below.
- "CONNECTED" reflects the current status of the modem carrier (DCD) signal.

- “WAITFOR” is the result of the last “WAITFOR” command - see the description below.

String comparisons are performed for certain commands (such as for case statements, or the FIND command). Such comparisons are *not* case sensitive.

The following is a complete list of the commands supported by the current script language:

Command	Description
ALARM number	Sound an audible alarm for number seconds.
ASSIGN string_var string	string_var := string. The first argument can be any of 10 string variables, named S0, S1, ... S9.
CLEAR	Clear the terminal screen.
CWHEN	Terminates the “WHEN” monitor (see WHEN command).
EXECUTE string	Load and execute the new script named in the string argument. The script processor <i>does not</i> return to the original script.
EXIT	Terminate the current script, and return to terminal mode.
FIND string_var string_lit	Test whether the literal string (string_lit) is part of string_var , and set the flag “FOUND” to the result of the test. The test is not case sensitive.
GET string_var [number]	Read a string from the keyboard, placing it in string_var . If the optional argument, number , is supplied, then this value is used as the maximum length for the string. If no value is supplied, then the maximum length is 80 characters.
GETFILE ident [string]	This command initiates a download using one of the protocols supported by the toolkit. string is the name of the file to be downloaded, and need not be supplied for batch protocols (Ymodem/Batch and Kermit). The possible values for ident are KERMIT, XMODEM, YMODEM, WYMODEM and BYMODEM. The “SUCCESS” and “FAILURE” flags are set based on the result of the download attempt.

GOSUB label

A return address is pushed and the program counter is loaded with the address of the **label**. Execution continues from the new program counter address. See RETURN.

GOTO label

Similar to GOSUB, except that no return address is pushed.

HANGUP

Hang up the modem line by dropping the DTR signal and then raising it again.

ISFILE string

Test whether the file named in **string** exists, and sets the "SUCCESS" and "FAILURE" flags accordingly.

KFLUSH

Flush the keyboard buffer.

LOCATE Row Column

Positions the cursor at the specified row and column address on the screen. **Row** and **Column** are numbers; **Row** is in the range 0..23 and **Column** is in the range 0..79. This command is equivalent to a GOTOXY(**Column**+1,**Row**+1).

LOG OPEN [string]

Open a logging (capture) file. The optional argument, **string**, specifies a name for the file. If no name is supplied, a default name of "term.log" is used. The "SUCCESS" and "FAILURE" flags are set, based on the outcome of the file opening.

LOG CLOSE

Close the logging file.

LOG SUSPEND

Suspend logging temporarily. Logging stops, but the logging file remains open.

LOG RESUME

Resume a suspended logging operation. Characters received after logging was suspended but before logging was resumed are lost.

MACRO digit

Transmit the macro specified by **digit**. A macro is a named sequence of commands and actions.

For example, you may use different logon names and passwords for different BBS. You can write macros to initialize userid and password variables for the different BBS.

MGET string_var [number]

Same as GET, except that this command echoes each "*" in response to each character typed. This command is used when entering passwords, etc.

PAUSE number	Suspend script execution for number seconds.
PRINTER ON	Turn printer logging on.
PRINTER OFF	Turn printer logging off.
QUIT	Terminate the script and exit the entire program.
RETURN	Pop an old program counter address from the stack. Script execution continues from the new address. RETURN should not be used unless there was a preceding GOSUB.
RFLUSH	Flush the serial input buffer.
RGET string_var [max] [tim]	Similar to GET, except that this command reads characters from the serial input buffer. Characters are read into string_var . The first number (max) is the maximum length of the string, and defaults to 80. The second number (tim) is the timeout in seconds. If a timeout is specified then a maximum length must also be specified. The "SUCCESS" and "FAILURE" flags are set depending on whether a string was read.
SENDFILE ident string	This command initiates an upload using one of the protocols supported by the toolkit. ident is the protocol to be used, and can be any of: KERMIT, XMODEM, YMODEM, WXMODEM and BYMODEM. string is the name of the file to be uploaded. The "SUCCESS" and "FAILURE" flags are set based on the result of the upload attempt.
TRANSMIT string	Transmit the specified string to the modem interface.
WAITFOR string [delay]	Wait for a string to arrive from remote. If the optional delay is set, then the script processor waits that long (in seconds). If no delay is specified, the default timeout is 30 seconds. If a delay of zero is specified then WAITFOR actually waits forever. The "WAITFOR" flag is set if the string was received within the wait period, or reset if it was not. The comparison performed is not case sensitive.
WHEN target action	This command sets up a monitor that runs for the duration of the script. target and action are strings. The script processor looks for occurrences

of the **target** string arriving from the remote system. If this string is received then the **action** string is transmitted. The comparison is not case sensitive.

Script Processing Modules

Script processing is performed by three modules. The module **Scanner** takes the source file of a script and breaks it into a stream of tokens for the compiler. Module **ScriptCompiler** compiles the script into a pseudo-code ready for execution. Module **ScriptProcessor** is the important module, since it is the interface to the compiler and pseudo-code interpreter.

Unless you feel like tinkering there is no need for you to study either the **Scanner** or **ScriptCompiler** modules. The module to study is **ScriptProcessor**, as this is the only module you will use directly.

The following listing shows the types and variables declared in **ScriptProcessor.DEF**.

```

TYPE ScriptStates = (Off,Running,Transmitting,Waiting,Pausing,WaitSuccess);
ScriptFlags = (Linked,Success,Failure,Found,Connected,WaitFor);
String80 = ARRAY[0..80] OF CHAR;

VAR TxPtr: CARDINAL;
    WhenProcessing: BOOLEAN;
    ScriptState: ScriptStates;
    Flags: ARRAY ScriptFlags OF BOOLEAN;
    WaitString, WhenString, WhenAction, TransmitString: String80;
```

TxPtr is used when a script is transmitting text. It is an index into the **TransmitString** variable.

WhenProcessing is **TRUE** when monitoring is active.

ScriptState describes the current state of the pseudocode interpreter. This state may be any of the following:

"Off"	No script is currently executing.
"Running"	A script is being executed.
"Transmitting"	The string in TransmitString is being sent via the modem interface. The script state will return to "Running" when this is complete.
"Waiting"	The script processor is currently waiting for a string to arrive, or for a timeout to elapse. Either way the script processor will return to "Running" when this is completed. The string variable

WaitString contains the string for which the script processor is waiting.

“Pausing”

The script processor is currently waiting for a timeout to elapse. The script processor will return to “Running” when this is completed.

“WaitSuccess”

The script processor is actually placed in this state by the external monitor which performs the “WHEN” processing when it receives the string contained in **WhenString**. The script processor’s response is to transmit the string in **WhenAction** and then return to its “Running” state.

Flags contains the current state of the (script) processor flags.

Strings that are transmitted using the TRANSMIT or MACRO commands may contain characters which are interpreted in a special way. For example, if the character “|” appears in the string, then it will be transmitted as a carriage return character. Other control characters may be transmitted by prefixing the character with “^”. For example, to transmit the sequence ^A^A<cr> the appropriate string would be “^A^A|.” If the character following the “^” is in the range ‘@’ to ‘_’ then it will be transmitted unchanged. This is useful if the character to be transmitted is itself either “|” or “^.”

The following procedures are found in the **ScriptProcessor** definition module.

InitScriptProcessor

PROCEDURE InitScriptProcessor;

This procedure performs a one-time initialization of the **ScriptProcessor** module. You should call it from the main initialization procedure in your program.

LoadScript

PROCEDURE LoadScript (Filename:ARRAY OF CHAR):BOOLEAN;

This function causes a script source file to be loaded, compiled and then prepared for execution. The function returns **TRUE** if all of this is done successfully. **Filename** is the name of the script file to be loaded.

SingleStep

PROCEDURE SingleStep;

The script is meant to appear to execute in parallel with other functions of the communications package. To achieve this you need to call **SingleStep** at frequent intervals. Each time this procedure is called, one pseudocode instruction is executed.

SingleStep creates a time-slice within which the script executes. The script is then run in a loop, along with carrier checking and abort-key checking. There is no other way provided to run a script.

TerminateScript

PROCEDURE TerminateScript;

This procedure will terminate a currently executing script.

An Example Script

The following listing contains a very brief example, which you can use to dial up to a particular BBS, such as BIX. In fact, the following listing shows how to call BIX. Where the listing says things such as "YourName," you need to substitute the appropriate values yourself.

```
clear
message "          Dialing BBS (tel # = 1234567)      |"
message "=====|"
transmit "ATDT 12345767|"
waitfor "connect 1200",60
if waitfor
  pause 1
  transmit "|"
  pause 1
  transmit "|"
  waitfor "login"
  transmit "BBSName|"
  waitfor "Name?",100
  transmit "YourName|"
  waitfor "Password:"
  transmit "YourPassword|"
else
  message "Timed out"
endif
exit
```


To invoke a script, you would need to call **InitScriptProcessor** to initialize things properly, **LoadScript** to prepare the desired script for execution, and then **SingleStep** frequently, to carry out a script instruction on each call.

You'll find an example program that lets you execute a script in Chapter 4.

Support Modules

The **CommDemo** program, supplied with the toolkit, shows how you can use the features of the TopSpeed Communications Toolkit to produce your own communications package. Some of the support modules for the **CommDemo** program may also be of interest; these are described in this chapter, beginning with the **CommDemo** program itself.

Module CommDemo

The **CommDemo** program provides a good example of how the various toolkit modules are used. The program lets you call another computer, and lets you send files back and forth between your computer and the remote system. You can keep a log of everything that happens during your session, and you can save this log for later study. When you want to transmit files, you can specify the protocol to be used — that is, **XModem**, **YModem**, **Kermit**, and so forth.

You can automate some of the communication process by using scripts. To execute a script in the **CommDemo** program, you need to select the script from the Dial Menu.

To run the program, type

```
commdemo
```


Chapter 5

Support Modules

The **CommDemo** program, supplied with the toolkit, shows how you can use the features of the TopSpeed Communications Toolkit to produce your own communications package. Some of the support modules for the **CommDemo** program may also be of interest; these are described in this chapter, beginning with the **CommDemo** program itself.

Module CommDemo

The **CommDemo** program provides a good example of how the various toolkit modules are used. The program lets you call another computer, and lets you send files back and forth between your computer and the remote system. You can keep a log of everything that happens during your session, and you can save this log for later study. When you want to transmit files, you can specify the protocol to be used — that is, XModem, YModem, Kermit, and so forth.

You can automate some of the communication process by using scripts. To execute a script in the **CommDemo** program, you need to select the script from the Dial Menu.

To run the program, type

```
commdemo
```


at the DOS command line. On startup, the program looks for a configuration file named **TERM.CFG** in the current directory. If this file is not found, then the program uses a default configuration. To change the default settings, simply press **ALT[S]** to enter setup mode. After making the desired changes, select Save setup to save the new configuration in file **TERM.CFG**.

While running, the program displays various information on a status line at the bottom of the display. This line provides the following information.

logo	The program name.
carrier	The state of the carrier. If there is a carrier, the symbol "DCD" appears in this position, otherwise the position is blank.
script	This specifies the name of the currently executing script file. If no script is being executed then the slot will be blank.
capture	This slot contains the name of the currently open log file. If no log file is open then this position will be blank.
emul	This position shows which terminal type is currently being emulated. This value will be either VT100, VT52 or ANSI.
time	This slot shows the elapsed time since the last log on. The time is represented in an hours:minutes.seconds format. This information is useful if you are logged onto a bulletin board that has a time limit for logons. The value is updated every second while you are on line. When the carrier is dropped the clock stops.

Commands to the CommDemo program

Commands are entered using the **ALT** key in combination with another character. The exceptions to this format are the upload and download commands (**PgUp** and **PgDn**, respectively) and the command to display the main **CommDemo** menu (**F1**).

The **CommDemo** program currently understands the following commands:

PgUp	Upload a file. The user is offered a menu of protocols. After selecting a protocol, the user is prompted for the name of a file to upload.
PgDn	Download a file. The user is again offered a menu of protocols, but is prompted for a filename only if the protocol chosen is not a batch protocol (that is, neither Ymodem-g nor Ymodem Batch nor Kermit).

[ALT][0..9]

Transmit Macro. In Setup mode the user can enter up to 10 macros which then can be transmitted using this command. To send the desired macro, hold down the **[Alt]** key in combination with the numbers key corresponding to the desired macro.

[ALT][B]

Transmit Break. This causes a break signal to be asserted on the modem interface for a period of $\frac{1}{2}$ second. On many mainframe or mini systems (such as the VAX), this signal is used to tell the host computer that the terminal is now "live."

[ALT][D]

Dial Menu. This command causes a menu to be displayed. The menu contains a list of scripts available in the default script directory (as defined in setup). Choose a script by number or move the highlight bar to the desired script. Then press **[Return]**. The selected script will be loaded and executed.

[ALT][H]

Hang up line. This command attempts to drop the line by lowering DTR and keeping it low for half a second.

[ALT][L]

Toggle Logging. **Logging** is the capture and storage of received text to disk. The "Toggle Logging" command will turn logging on if it is currently off, and will turn it off if it is currently on.

When logging is turned on you will be prompted for the name of the file in which the text will be stored. If you press **[Return]**, a default name of **TERM.LOG** is used. Press **[Esc]** to cancel the command.

[ALT][S]

Change Setup. This option is entirely menu driven and should be self explanatory.

You'll find that you can change various types of settings. These include general communications parameters such as baud rate and parity, the number of data and stop bits, and so forth. You can also specify default values for the Kermit and ASCII transfer protocols, and can specify the type of terminal you want the program to emulate. Finally, you can edit macros by making the appropriate selection from the Setup menu.

[ALT][X]

Exit program. You are prompted for confirmation. If this is forthcoming, the program ends.

Module ComSetup

The **ComSetup** module deals with configuration information for the demo communications package. Among other things, the module makes it possible to display the

Setup menu. This module is also needed to allow the user to change serial port parameters, general parameters, and various file transfer protocol options.

All of the configuration information is stored in the record variable **Setup**. Expand this variable definition as necessary for your own needs. The module does most of its work using modules **MenuLib** and **SPRINTF** which are described later.

InitSetup

PROCEDURE InitSetup;

This procedure attempts to initialize the **Setup** record by first attempting to open a configuration saved in the file **TERM.CFG**. If this is unsuccessful the **Setup** record is initialized to a default configuration.

ChangeSetup

PROCEDURE ChangeSetup;

This procedure displays the "Setup" menu and allows you to change the program configuration while running the program.

SaveSetup

PROCEDURE SaveSetup;

This procedure writes the current configuration to the file **TERM.CFG** in the current directory.

Module Filenames

Filenames is a utility module for processing DOS file specifications. In particular, the module contains routines for determining the components of a complete file name and for creating a complete file name, given the component values.

ParseFilename

```
PROCEDURE ParseFilename(f:ARRAY OF CHAR;  
                        VAR drive,path,name,ext:ARRAY OF CHAR):BOOLEAN;
```

This procedure attempts to break its first argument (**f**) up into its constituent parts: **drive**, **path**, **name**, and **ext**. Any constituent for which no value is specified will be returned as a null string. The function returns **TRUE** if **f** is a valid filename.

Normally this function is used in conjunction with **MakeFilename**, to break up a filename entered by the user, and to fill in the default parts (such as a default path or extension).

MakeFilename

```
PROCEDURE MakeFilename(drive,path,name,ext:ARRAY OF CHAR;  
                       VAR f:ARRAY OF CHAR);
```

This procedure takes the constituent parts (**drive**, **path**, **name**, and **ext**) of a DOS file specification and concatenates these components. The procedure is reasonably forgiving. For example, if the drive name does not have a colon (:) attached then one will be added. Similarly if the path does not have a separator (\) at the end, then one will be added. Finally, if the extension does not start with a period (.) then one will be inserted — unless, of course, any of these parameters were null strings.

Module Keyboard

Module **Keyboard** is a replacement driver for the keyboard, and provides services that are compatible with the Int 16H BIOS services. This module is needed because of the VT100 emulation, which requires more keys (especially from the numeric keypad) than are recognized either by DOS or the standard BIOS.

For example, this driver enables the programmer to differentiate between numeric keys on the keypad and on the main keyboard. The driver also returns codes for *all* the keys on the keypad in cursor key mode (that is, when NumLock is off). In contrast, the normal PC BIOS does not return a code for the central (5) key of the numeric keypad.

Although this driver works fine on standard PCs and ATs it is possible that some clones will not work correctly. In that case, you may wish to stop using this keyboard driver. However, that will also limit VT100 emulation. You should still be able to provide normal IBM-style ANSI emulation, however.

The **Keyboard** driver is installed automatically when any program **IMPORTs** it. Once installed the driver works invisibly in the background, just as if you were still using the normal BIOS routines. In fact, you can continue to use **IO.RdKey**, as this does not notice any change. The keyboard codes returned are exactly identical to those of the standard BIOS, except that more keys return codes under the alternative driver.

Keyboard.DEF does define two functions and one variable, which you may use directly if you wish. The two routines are **KeyPressed** and **RdKey**, and are identical — in their use — to the routines of the same names in the standard **TopSpeed IO** module.

The variable defined in the driver is

```
VAR ScanCode:CHAR;
```

and contains the scan code of the last key read.

Module MenuLib

The **MenuLib** module is responsible for creating and controlling the kind of popup menu that the **CommDemo** program uses for its Setup, File transfer, and Dial menus. The module is completely stand-alone, and could be used for other programs if you wished.

This module is not intended to provide extensive Menu capabilities; rather, the module's main feature is simplicity. Menus using this module are restricted in size to a maximum 30 characters wide by 23 lines (menu options) deep.

Here is the type and variable definition part of module **MenuLib**:

```
IMPORT Window;
```

```
CONST MaxOption = 24;
```

```
TYPE MenuOption = RECORD
```

```
    Name:ARRAY[0..30] OF CHAR;
```

```
    XPos,YPos,FirstNonSpace:CARDINAL;
```

```
END;
```

```
MenuRec = RECORD
```

```
    Title:ARRAY[0..30] OF CHAR;
```

```
    Option:ARRAY[0..MaxOption] OF MenuOption;
```

```
    MenuX,MenuY,Width,NumUsed:CARDINAL;
```

```
    EscapeOK,ExitOnChar,HelpOK,
```

```
    ExitSideWays,PageKeysOK:BOOLEAN;
```

```
    Normal,Selected,Highlite,SHighlite:SHORTCARD;
```

```
    Selection:CARDINAL;
```

```
    winptr:Window.WinType;
```

```
END;
```



```

ColorRec = RECORD
    NormFore, NormBack,
    FrameFore, FrameBack,
    HighFore, HighBack,
    SelFore, SelBack,
    HSelFore, HSelBack : Window.Color;
END;

MenuColors : ColorRec;

```

In many ways, the use of **MenuLib** is analogous to the use of the standard **TopSpeed Window** module. In both cases, you declare a variable of type **MenuRec**, and you initialize it using procedure **InitMenu** (All procedures mentioned will be described fully below.)

You can then use procedure **AddOption** to add options to the menu. Procedure **MenuDisplay** is used to display the menu, and **MenuDrive**, the menu driver function, is called to animate the menu and return a result. Finally, procedure **KillMenu** is called to close the window containing the menu, thus restoring what was underneath.

Some menus have special features that you can control using the boolean variables defined in the menu record. All of these boolean fields default to **FALSE**.

- If **EscapeOK** is **TRUE**, then the user can exit from the menu by pressing **[Esc]**.
- If **ExitOnChar** is **TRUE**, then pressing the first character of a menu option should be equivalent to moving to the option with the cursor keys and pressing **[Return]**. If this is the case, then the first character of the menu option will be highlighted.
- Variable **HelpOK** is **TRUE** if pressing **[F1]** should exit from the menu. In this case your program would provide help and information about the menu option which was highlighted when the user pressed **[F1]**. (This position is specified by the **Selection** field of the menu record.)
- Variable **ExitSideWays** is **TRUE** if you want the menu driver (animator) to exit if the **[←]** or **[→]** keys are struck. Otherwise the driver will simply treat these as if they were cursor movement keys, and will assume that the user pressed them in error.
- Finally, **PageKeysOK** is **TRUE** if you want the menu to exit when the **[PgUp]** or **[PgDn]** keys are pressed.

When the user chooses a menu option then the menu driver will exit and will return a result indicating what the choice was. Normally this result will be in the range of 0 .. n-1 (where n is the number of menu options).

However, the result also can be negative if any of several special exit conditions were met. The following list shows the possible negative results and their meanings:

- 1 The user pressed **[Esc]**, and **EscapeOK** is true.

- 2 The user pressed **PgUp**, and **PageKeysOK** is true.
- 3 The user pressed **PgDn**, and **PageKeysOK** is true.
- 4 The user pressed **F1**, and **HelpOK** is true.
- 5 The user pressed **⇒**, and **ExitSideWays** is true.
- 6 The user pressed **⇐**, and **ExitSideWays** is true.

The following procedures and functions are defined in **MenuLib.DEF**.

InitMenu

```
PROCEDURE InitMenu (VAR menu:MenuRec; menutitle:ARRAY OF CHAR;
                   MenuX,MenuY,menuwidth:CARDINAL);
```

This procedure initializes a variable of type **MenuRec**, giving values to the various fields which make up the record. **menutitle** is the name you wish to give to the window that contains the menu. This may be a null string if you don't want the menu to have a title.

MenuX and **MenuY** are the absolute coordinates of the upper left corner of the menu window. **MenuX** should be in the range 0..79, and **MenuY** should be in the range 0..23.

menuwidth is the maximum width of the menu options you will be adding. Procedure **InitMenu** *must* be called before any of the other procedures are applied to the menu.

AddOption

```
PROCEDURE AddOption (VAR menu:MenuRec; menutext:ARRAY OF CHAR);
```

This procedure adds an option to a menu. The menu must previously have been initialized using procedure **InitMenu**. **AddOption** should be called once for each option being added to the menu.

DispMenu

```
PROCEDURE DispMenu (VAR menu:MenuRec);
```

This procedure displays a menu which has been initialized and then completely defined by repeated calls to **AddOption**.

MenuDrive

```
PROCEDURE MenuDrive (VAR menu:MenuRec):INTEGER;
```

This function “animates” a defined menu, which already should be visible on the screen. See the description earlier in the module discussion for the result codes returned by this function.

KillMenu

```
PROCEDURE KillMenu (VAR menu:MenuRec);
```

This procedure closes the window containing a menu, restoring what was underneath the window.

RepaintMenu

```
PROCEDURE RepaintMenu (VAR menu:MenuRec);
```

This procedure redisplay the contents of a menu window, but assumes that the window is already visible onscreen. This procedure is normally used to update a menu onscreen when the options have been changed by a user routine.

Modules PRINTF/SPRINTF

The **PRINTF** and **SPRINTF** modules help you avoid some of the tedium of controlling output to the display and to strings in Modula-2. Each of these modules defines several procedures for controlling formatted output. Be careful not to confuse the module names (which are entirely uppercase) with the procedures of the same names — that is, **Printf** and **Sprintf** — which are mixed case.

If you look at some of the code in module **ComSetup** you will realize that much of it would have been considerably longer were it not for the extended use of calls to procedure **Sprintf** to format output to menu option strings.

In case you do not recognize the names, **Printf** and **Sprintf** are the names of output procedures normally associated with the standard library of the C programming language. For some reason, the Modula-2 community has attached itself to an awkward set of I/O procedures, largely ignoring the solutions for similar problems already found in C. (C is similar to Modula-2 in that most of the special functions,

such as I/O, have been relegated to external libraries.) You may wish to experiment with these modules which provide something quite similar to the C method of I/O, but for Modula-2 programmers.

The modules which comprise the Modula-2 support for formatted output are as follows:

PrintFBase This base module is used by the other **PrintF**-related modules. Although the module contains only two public routines, it does the “dirty-work” for the formatted output routines. For most situations you won’t need to use this module directly.

PRINTF This module contains six routines for sending formatted output to the display. The routines differ primarily in the number of arguments they output.

SPRINTF The routines in this module are similar to those in module **PRINTF**, except that the routines in **SPRINTF** pass the output to a string argument, rather than displaying the output.

You may want to look in a good book on C for more extensive discussion of the definitions of the C versions of **Printf** and **Scanf**. Two possible sources are Harbison & Steele, *C: A Reference Manual*, 2nd edition (Prentice-Hall, 19878), or the original definition of the C language by Kernighan & Ritchie in *The C Programming Language* (2nd edition, Prentice-Hall, 1988).

The **Printf**-related modules provide a method of output which is totally unlike what you’ll find in the usual Modula-2 I/O modules. In the latter you are expected to import a different output procedure for each variable type that you wish to display. In the TopSpeed IO Library, these routines are all gathered in the same place, which at least makes it a bit more convenient to use them.

This can be incredibly tedious to use, and can turn what would have been a simple output statement in Pascal or C into a screenful of I/O calls. Needless to say, this has the effect of obscuring the program, even those parts which are not concerned with I/O.

Printf-related routines

```
PROCEDURE Printf (pat:ARRAY OF CHAR);
PROCEDURE Printf1(pat:ARRAY OF CHAR; arg1:ARRAY OF BYTE);
PROCEDURE Printf2(pat:ARRAY OF CHAR; arg1,arg2:ARRAY OF BYTE);
PROCEDURE Printf3(pat:ARRAY OF CHAR; arg1,arg2,arg3:ARRAY OF BYTE);
PROCEDURE Printf4(pat:ARRAY OF CHAR; arg1,arg2,arg3,arg4:ARRAY OF BYTE);
PROCEDURE Printf5(pat:ARRAY OF CHAR;
                  arg1, arg2, arg3, arg4, arg5:ARRAY OF BYTE);
```


The **PRINTF** module contains six different procedures for displaying formatted output. For simplicity, we'll speak of **Printf** when the discussion concerns these procedures in general, rather than a specific one under discussion. The differences between the procedures will become clear in the following discussion.

The basic parameter supplied to each of the **Printf**-related routines is called the **control string**. In the simplest case this string is merely a sequence of characters to be output. For example,

```
Printf('Hello, World');
```

would simply write the string 'Hello, World' to the display.

The control string is special however. In addition to being allowed to specify ordinary characters (which are written exactly as specified), you may also embed commands in the string. These commands serve a dual purpose:

- They serve as placeholders to indicate that a particular value will be substituted at that location when the string is actually output.
- They specify what type of value is to be output at that location.

If you have such a command in the control string, you must supply an appropriate value as an additional parameter when calling a **Printf**-related routine. The number of commands in the control string will determine which procedure you need to call. For example, consider the following call:

```
Printf1('I am %i today.', age);
```

This outputs the string 'I am <something> today'. The <something> is replaced in the output with whatever you supply as the second argument. The '%i' part of the control string says that this parameter *must* be an integer variable — although it can be of any integer type (including **SHORTINT**, **INTEGER** or **LONGINT**).

In the example given, if the integer variable **age** equaled 12, then the string

```
I am 12 today.
```

would be output.

You may have several of these %<char> directives in the control string, and each one must be matched by another parameter supplied to the appropriate **Printf**-related routine.

It is unfortunate but true that Modula-2 does not allow procedures to take a variable numbers of parameters. To get around this, the variants of **Printf** are supplied. These differ only in the number of parameters that each procedure takes. All versions of the **Printf** routine take at least the control string, so the number of parameters

referred to in the procedure names does not include this string. Thus, `Printf1` takes one parameter in addition to the control string.

The embedded commands in a control string are called **field specifiers**. All field specifiers have the following format:

`'%' [optional formatting directive] <type character>`

The type character can be any character from the set `'a'..'z'` or `'A'..'Z'`. `Printf` is case sensitive, so `'a'` and `'A'` are not treated as the same character. Currently, only the following type characters are supported by `Printf`:

- `'i'` output a *decimal* **INTEGER**, **SHORTINT** or **LONGINT**
- `'u'` output a *decimal* **CARDINAL**, **SHORTCARD** or **LONGCARD**
- `'c'` output a **CHAR**
- `'s'` output a string (that is, an **ARRAY OF CHAR**)
- `'h'` output the parameter value in "lower case hexadecimal" format (for example, `0fff` rather than `0FFF`). The parameter may be any of the types accepted by `'i'`, `'u'` or `'c'`.
- `'H'` same as `'h'`, except that the hexadecimal output is uppercase (for example, `0FFF`).
- `'x'` same as `'h'`.
- `'X'` same as `'H'`.

Here is a somewhat longer example of a call to one of these library functions:

```
Printf5('My name is %s, aged %u, born %u/%u/%u',
       name, age, day, month, year);
```

If `name = 'Sam'`, `age = 20`, `day = 10`, `month = 1`, and `year = 70`, then the output of `Printf` would be:

```
My name is Sam, aged 20, born 10/1/70
```

None of the examples shown so far specify any formatting of the object to be output. To do this you need to place the formatting directive between the `'%'` and the type character. For example, if you wanted the date in the example to be output in a different format (such as `dd/mm/yy`), then you would change that part of the control string to

```
%02u/%02u/%02u
```

The following list shows the valid formatting directives:

<any decimal number> This number specifies the minimum width of the field within which the appropriate value will be written. For example `%2u` specifies a minimum width of 2 characters. The field will be padded out with spaces to the left of the value — unless you put a leading zero in front of the width. In that case, the field would be padded out with zeroes. Thus, `%3u` would produce ' 1' and `%03u` would produce '001' when output.

Please remember that the field width specifies a *minimum* field width. This means that if the conversion produces a string whose length is less than the width then it will be padded. On the other hand, if the actual length is greater than the specified field width, then the value will *not* be truncated to the field width, but rather will be output as is.

'*' This directive tells `Printf` that the field width should be taken from a supplied parameter instead of from the control string. Examples of this are `*u` and `%.*r`.

'-' This directive tells `Printf` that the field should be *left* justified. This directive has no effect if no field width is specified.

'+' This directive indicates that all decimal output should be signed. Negative values are always signed, but positive values generally are not preceded by a sign. By including this directive positive numbers also would have a leading sign.

' ' A space character means that all decimal output should be signed if negative, or a space should be output if not.

The following list contains some examples using such directives:

'%-30s' A string parameter is supplied, and should be padded to a width of 30 characters. The output string should be left justified.

'%+i' An integer parameter is supplied and should be signed (either with '-' or '+') when output.

'%05i' An integer is expected, and should be output with a field width of 5, padded on the left with zeroes.

'%+05i' As above, except that the resulting value is always signed.

'%04H'

An **INTEGER**, **CARDINAL** or **CHAR** value is expected, and is output in upper case hexadecimal padded with 0s to a width of 4 digits.

'%%'

Write the character '%.'

There is another form of instruction which may be embedded in control strings: **switch characters**. These basically allow the programmer to embed control characters in literal strings. In C such instructions are interpreted by the compiler and replaced with actual control characters, however in our Modula-2 implementation the switch instruction is interpreted at runtime.

Switch characters provide a way of embedding non-printable characters in the control string so that they can be output by **PrintF**. All switch characters have the following form:

'\<Character Definition>'

"Character Definition" will be one of the following:

nnn A three digit *decimal* number that specifies the ASCII code of the character to be output. For example, '\001' means **Ctrl****A**. You should always supply exactly three digits, otherwise it is possible (though perhaps unlikely) for **PrintF** to become confused by unrelated digits immediately following the control character.

'n' A newline character sequence is output (CR LF).

'r' A single carriage return is output. This command moves the cursor to the first (leftmost) column of the *current* line. The cursor does not automatically move to the next line, and therefore behaves differently from a newline character.

't' Output a Tab character (ASCII 9).

'f' Output a formfeed character (ASCII 12).

'b' Output a backspace character (ASCII 8).

'\' Output a backslash (\) character (ASCII 92). (With the starting backslash, this command is '\\')

SPrintF-related routines

```

PROCEDURE SPrintF(pat:ARRAY OF CHAR; VAR dest:ARRAY OF CHAR);
PROCEDURE SPrintF1(pat:ARRAY OF CHAR; arg1:ARRAY OF BYTE;
    VAR dest:ARRAY OF CHAR);
PROCEDURE SPrintF2(pat:ARRAY OF CHAR; arg1,arg2:ARRAY OF BYTE;
    VAR dest:ARRAY OF CHAR);
PROCEDURE SPrintF3(pat:ARRAY OF CHAR; arg1,arg2,arg3:ARRAY OF BYTE;
    VAR dest:ARRAY OF CHAR);
PROCEDURE SPrintF4(pat:ARRAY OF CHAR; arg1,arg2,arg3,arg4:ARRAY OF BYTE;
    VAR dest:ARRAY OF CHAR);
PROCEDURE SPrintF5(pat:ARRAY OF CHAR;
    arg1,arg2,arg3,arg4,arg5:ARRAY OF BYTE;
    VAR dest:ARRAY OF CHAR);

```

Notice that **SPrintF** and its related functions have a format very similar to that for the **PrintF** collection. These functions work in exactly the same way as **PrintF** which has been described above. The only difference is that **SPrintF** stores its output in strings, which is why it takes the second parameter. Thus, each call to an **SPrintF**-related function takes an extra parameter. This additional argument is the output — or target — string.

Module Timer

Module Timer provides support for the “elapsed time” clock, which is displayed on the status line in the **CommDemo** program. The module also provides a general-purpose counter-timer for use by the file transfer protocols, etc. (This is referred to in the module as the “Application Timer” or “AppTimer”).

The module should be fairly portable, since it counts using the DOS system clock instead of using the PC hardware ticker. The only variable declared in the definition contains the current state of the clock, and an associated clock record type (**Clock-Rec**) is also defined:

```

TYPE ClockRec = RECORD
    Running:BOOLEAN;
    Hours,Minutes,Seconds:CARDINAL;
END;

VAR Clock:ClockRec;

```

These declarations should be fairly obvious. The following procedures are also defined in **Timer.DEF**:

StartClock

PROCEDURE StartClock;

This procedure starts the elapsed time clock ticking. The clock display on the status line is not updated automatically however. Rather, the clock needs to be read regularly and the status line must be updated externally. In the **CommDemo** program this is accomplished in module **CommUtil**.

StopClock

PROCEDURE StopClock;

This procedure stops the elapsed time clock.

SetAppTimer

PROCEDURE SetAppTimer(Seconds:CARDINAL);

This procedure is used to set the general-purpose "Application" counter/timer to expire in **Seconds** time. **Seconds** may be zero, which effectively turns the timer off.

AppTimerExpired

PROCEDURE AppTimerExpired():BOOLEAN;

This function tests whether the application timer has expired — that is, whether it has reached zero. Note that this function will always return **TRUE** if the timer is not actually running.

UpdateTimer

PROCEDURE UpdateTimer;

As mentioned above the timer module outlined here does not use the hardware ticker interrupt, and so is not a background routine. You can achieve the effect of background use, however, by calling procedure **UpdateTimer** regularly. Note that, although the update should be regular, it need not be frequent. Once a second or so will be

enough, since both the elapsed time clock and the application timer have a minimum granularity of one second.

Module CommUtil

The **CommUtil** module serves to gather in one place all those procedures, functions and variables that are either of general utility to a wide range of modules, or that do not fit well elsewhere.

The following types and variables are declared in **CommUtil.DEF**:

```

IMPORT Window;

TYPE String80 = ARRAY[0..80] OF CHAR;
StatusFields = (logo, carrier, script, capture, emul, time);
LogCommands = (OpenLogging, CloseLogging, SuspendLogging, ResumeLogging);

FuncKeys = (NotRecognised,
            BackSpace, Etx, Escape, Del, Cup, Cdown, Cleft,
            Cright, InsertKey, DeleteKey, Tab, BackTab,
            HomeKey, EndKey, PageUp, PageDown, FKey, AltKey);

VAR LogMode:LogCommands;
    Printing:BOOLEAN;
    Term, Stat:Window.WinType;
    FuncKeyTyped:FuncKeys;
    CaptureFile, TransferFile, ScriptFile:String80;

TYPE OutProc = PROCEDURE (CHAR);
VAR WrChar:OutProc; (* Normally from IO, but could be other,
                    for example, an emulator *)

```

Most of these declarations relate to a particular procedure, and will be described below.

- **LogMode** is the current state of text logging. See the enumerated type **LogCommands** for the possible states.
- **Printing** is **TRUE** if print logging is currently active.
- **Term** and **Stat** are the window variables for the main screen and status line windows, respectively.
- **CaptureFile** is the current log file, if one is open.
- **TransferFile** is the name of the last file to be uploaded or downloaded using one of the file transfer protocols.
- **ScriptFile** is the name of the currently executing script, if one is being executed.

The procedure variable **WrChar** allows module output to be redirected — normally to a terminal emulation module, such as the **Term** module provided. All of the communications support modules perform output using this call, so that they can be used regardless of whether terminal emulation is implemented. The initialization procedure in **CommUtil** sets the default to **IO.WrChar**.

The following sections describe the procedures and functions declared in **CommUtil.DEF**.

InitUtil

```
PROCEDURE InitUtil;
```

This procedure performs a one-time initialization of module **CommUtil**. The procedure should be called from the initialization section of the main program.

Read

```
PROCEDURE Read(VAR c:CHAR);
```

This procedure reads and returns single characters from the keyboard. The procedure also takes some of the misery out of interpreting various standard keyboard functions by mapping them onto **FuncKeyTyped**. This value can be checked to determine whether one of the standard recognized functions was typed. This also makes things more readable in routines for editing lines, since it makes it unnecessary to use long case statements full of unintelligible PC scan codes.

StatusLine

```
PROCEDURE StatusLine(which:StatusFields);
```

This procedure is used to update specific fields on the **CommDemo** program status line.

CheckHistory

```
PROCEDURE CheckHistory(VAR s:ARRAY OF CHAR):BOOLEAN;
```

The **CommDemo** program maintains a 128-character “history” of characters recently received from the remote system. This procedure compares the last **Length(s)** characters of the history with string **s** to see if they match. (The comparison is not

case sensitive.) The function returns **TRUE** if the string matches the history. This is used to implement the “WaitFor” and “When” script functions.

GetHistoryString

```
PROCEDURE GetHistoryString(VAR s:ARRAY OF CHAR):CARDINAL;
```

This procedure returns the 10 most recent characters from the “history” described in the discussion of procedure **CheckHistory**. The **CARDINAL** value returned represents the number of history characters actually returned. This will generally be 10, but may be a smaller value. This procedure is not currently used by the **CommDemo** program.

StoreHistory

```
PROCEDURE StoreHistory(c:CHAR);
```

This procedure stores a character in the history buffer (see the discussion of **CheckHistory**). It is not necessary to call this procedure directly if characters are written to the display using procedure **Screen**, which is described elsewhere in this section.

StoreChar

```
PROCEDURE StoreChar(c:CHAR);
```

This procedure is used to store received characters in the current log file, if one is open. It is not necessary to call this procedure directly if characters are written to the display using procedure **Screen**, which is described elsewhere in this section.

SetLogMode

```
TYPE LogCommands = (OpenLogging, CloseLogging, SuspendLogging, ResumeLogging);
```

```
PROCEDURE SetLogMode(Cmd:LogCommands);
```

This procedure changes the current log mode. The first two options are used by the “Toggle Logging” command in the **CommDemo** program, and all four options are available when driven from a script.

Printer

```
PROCEDURE Printer(OnOff:BOOLEAN);
```

This procedure turns printer logging on or off.

Screen

```
PROCEDURE Screen(Ch:CHAR);
```

This procedure sends characters to the display, via the redirection procedure variable `CommUtil.WrChar`. The routine takes care of storing the character in the history buffer, the log file, and also sends the character to the printer, if necessary.

ErrorMessage

```
PROCEDURE ErrorMessage(s:ARRAY OF CHAR);
```

This procedure displays an error message in a window, where the user can read it. The procedure then waits for the user to press **Esc** before erasing the window.

SendString

```
PROCEDURE SendString(s:ARRAY OF CHAR);
```

This procedure sends a formatted string to the modem interface. See the section on transmitted strings in Chapter 4, on Scripts.

TransmitMacro

```
PROCEDURE TransmitMacro(MacroNo:CARDINAL);
```

This procedure transmits one of the ten macros created by the user while using the Setup mode in the `CommDemo` program.

SelectFile

```
PROCEDURE SelectFile(MenuX,MenuY:CARDINAL; Title,Wild:ARRAY OF CHAR;  
VAR f:ARRAY OF CHAR):INTEGER;
```

This function is normally used to implement the dialing menu, but has been written to be quite general-purpose. The function offers a list of files in the form of a menu.

- **MenuX** and **MenuY** are the x- and y-coordinates of the window containing the menu.
- **Title** is the name to be given to the menu window.
- **Wild** is a wildcard specification (such as ***.CMD**) for the directory search that will yield the list of files that will be displayed. The list is sorted.
- The user chosen file name is returned in **f**.

The function returns the result of the menu call. This value is not normally important unless it is -1, in which case the user pressed **Esc** and no action should be taken. (Do not attempt to execute the chosen file in such a situation).

DoUpload

```
PROCEDURE DoUpload;
```

This procedure causes a menu of available upload protocols to be displayed. After one is chosen, the procedure prompts the user for the name of the file to be uploaded. The upload is then carried out.

DoDownload

```
PROCEDURE DoDownload;
```

This procedure causes a menu of available download protocols to be displayed. After one is chosen, the procedure prompts the user for the name of the file to be downloaded — unless the chosen protocol is a batch protocol (that is, Ymodem-g, Ymodem Batch or Kermit). In that case, no filename prompt is offered; rather, the file name is supplied by the remote system. The download is then carried out.

ReadLine

```
VAR SpacesOK:BOOLEAN;

PROCEDURE ReadLine(VAR s:ARRAY OF CHAR; Max:CARDINAL);
```

This is a general string reading and editing procedure. **Max** is the maximum allowed length for the string. If **SpacesOK** is **TRUE**, then spaces will be allowed in the string. **SpacesOK** may be **FALSE**, for example, when entering or editing file or path names.

PushBack

```
PROCEDURE PushBack(c:CHAR);
```

This procedure “unReads” a character previously read using the **Read** procedure (described elsewhere in this section). This makes it possible to read the character again – using procedure **ReadLine** for example. This technique is useful when the programmer wants a character that has already been read to be included in a string yet to be read. The **ReadLine** procedure must be used to read the string.

ReadString

```
PROCEDURE ReadString(x,y:CARDINAL; Title,Prompt:ARRAY OF CHAR;
VAR Result:ARRAY OF CHAR);
```

This procedure is used like **ReadLine**, but adds some extra “frills” that are used in the **CommDemo** program. The string is read in a window with an optional title, and with an optional prompt. **x** and **y** are the coordinates of the window to be displayed, which is always three lines high, counting the frame.

Title is the name to be given to the window and may be a null string. Similarly, **Prompt** is the prompt to be offered to the user, and may also be a null string.

VT100/VT52/ANSI Terminal Emulation

Term, the terminal emulation module, emulates a VT100 terminal. This means that, as a byproduct, the module also supports VT52 emulation — since real VT100 terminals also support the VT52 terminal’s commands. A subset of the ANSI standard commonly found on bulletin boards systems also is supported.

Emulation of both screen and keyboard sequences is provided, although certain requirements of keyboard emulation are problematical and have required rather strange solutions. These will be explained below.

The VT100 terminal is a complex device, and is difficult to emulate completely without using machine dependant resources such as graphic modes, and so forth. The emulation provided will normally be enough to interact with BBS systems that send and expect ANSI or VT100 sequences. For more demanding applications (such as using the PC as a VAX terminal, for example), you may want to expand the capabilities of the emulator yourself. To accomplish this, you would have to sacrifice program portability.

Emulation of the VT100 keyboard is difficult to do on the standard PC because of the shortage of keys, the placement of keys, and the dual nature of the PC keypad. This has been resolved to some extent in the toolkit by means of a special keyboard driver. This driver takes over the hardware keyboard interrupt, allowing the emulator to differentiate more keys.

When using the VT100 emulation you should keep the keypad in cursor key mode — that is, you should keep NumLock mode off. If you hold down the **[Shift]** key while pressing keypad characters then it can be used as a numeric keypad. Otherwise the four cursor keys will generate ANSI cursor key sequences in normal mode, or DEC alternate key sequences in application mode. If you press keypad keys while holding down the **[Ctrl]** key then the keys will produce numbers in DEC “Numeric Keypad Mode,” or DEC alternate sequences in “Alternate Keypad Mode.”

The following procedures are defined in the **Term.DEF** emulation module:

WrChar

PROCEDURE WrChar (c:CHAR);

The VT100 emulation has been implemented in such a way that the programmer may imagine that an actual VT100 has been attached to the PC, using a spare serial port. Procedure **WrChar** writes to this imaginary VT100 using this virtual port. The VT100 emulator (which is a black box for the user) responds to the characters written to it using this interface, just as a real VT100 terminal would respond to characters received from its serial port.

RdKey

PROCEDURE RdKey():CHAR;

This function returns keys which are read from the imaginary VT100 attached to your PC. The keys returned are mostly VT100 key sequences or normal ASCII characters. Those that are not (such as any key in combination with **[ALT]**) are preceded with a **NUL** byte in the traditional IBM fashion. The key code following this byte is a standard IBM PC special key code.

KeyPressed

PROCEDURE KeyPressed():BOOLEAN;

This function returns **TRUE** if characters are waiting to be read from the VT100 emulator.

ClearScreen

PROCEDURE ClearScreen;

This procedure clears the screen, when you're using VT100 emulation. You must use this procedure if you wish to clear the display while in VT100 emulation. This is because the VT100 must make special changes to internal tables when the screen is cleared.

Emulation

TYPE Emulations = (EmANSI,EmVT100,EmVT52);

PROCEDURE Emulation():Emulations;

This function can be used to test whether the emulator is currently emulating a VT100,VT52 or an ANSI device.

EmulateVT100

PROCEDURE EmulateVT100;

This procedure is used to initialize the emulator module for VT100 emulation. This is the default emulation chosen by the initialization code of the module.

EmulateVT52

PROCEDURE EmulateVT52;

This procedure is used to initialize the emulator module for VT52 emulation.

EmulateANSI

PROCEDURE EmulateANSI;

This procedure is used to initialize the emulator module for ANSI emulation.

Appendix A

The 8250 UART

The standard serial chip in the IBM PC is the 8250 UART (Universal Asynchronous Receiver/Transmitter). As its name implies, this chip provides only asynchronous and not synchronous communications. Some clones may use other chips, but almost all of these will be compatible with the 8250.

The 8250 UART has 10 registers which can be read or modified using the I/O Port whose address is mapped at a fixed offset from the known base address. The base address of port 1 (for example) is normally 3F8H; for port 2 it's normally 2F8H. Ports 3 and 4 are at 3E8H and 2E8H, respectively.

Thus, to access a register such as the the **Interrupt Identification Register** (IIR) of port 1 (COM1), the address would be computed as: base address + offset, that is $3F8H + 2 = 3FAH$. The offsets of the registers on the 8250 are always the same on any hardware, regardless of the base address.

The following sections describe each of the 8250 registers. In some cases, two registers will have the same offset. In such a case, the registers are mutually exclusive. One or the other can be used. Which register is valid will depend on settings in other registers.

Receive Buffer Register (RBR) - I/O Address: Base+0

The **Receive Buffer Register** is a read only register. It can be accessed only when bit seven of the **Line Control Register** (LCR) is reset. (This bit is the **Divisor Latch Access Bit**, or **DLAB**.) Received data bytes are read from this register.

Transmit Holding Register (THR) - I/O Address: Base+0

This is the **Transmitter Holding Register**. Bytes written to this register are sent out through the serial port. It can be accessed only when the **DLAB** is reset.

Divisor Latch Least (DLL) - I/O Address: Base+0

Divisor Latch Least represents the least significant byte of the clock divisor which provides the bit rate for the serial port. Notice that the offset is the same as the **THR** register, and so can only be accessed when **DLAB** is set. See also **DLM**.

Divisor Latch Most (DLM) - I/O Address: Base+1

Divisor Latch Most represents the most significant part of the clock divisor. Again, the **DLAB** must be set in order to access this register. **DLL** and **DLM** together make up the sixteen bit divisor that provides the serial bit rate for the port. The appropriate divisor is calculated by dividing 115200 by the required bit rate. For example, to achieve a rate of 1200 bps the appropriate divisor is $115200 / 1200$ which equals 96.

Interrupt Enable Register (IER) - I/O Address: Base+1

This is the **Interrupt Enable Register**. The 8250 is capable of raising any of four different types of interrupt, each of which can be individually enabled or disabled by setting or resetting the correct bit in this register. The bits are as follows:

Bit	Interrupt On:
0	Received Data Available
1	Transmit Holding Register empty (so ready to receive another transmission)
2	Receiver Line Status change — interrupt when there is any change in the Line Status Register (see p. 74)
3	Modem Status Change — interrupt when there is any change in the Modem Status Register (see p. 74)
4-7	Unused.

Interrupt Identification Register (IIR) - I/O Address: Base+2

When an interrupt occurs, its type can be identified by reading this register and checking individual bits.

Bit	Description
0	Set if <i>no</i> interrupt is pending.
1-2	Two bit interrupt type code (see below)
3-7	Unused

The possible codes for specifying interrupt types are:

Code	Description	Cleared By:
0	Modem Status Change (see MSR)	Reading MSR
1	Transmitted Data Register Empty	Reading IIR or writing THR
2	Received Data Available	Reading RBR
3	Received Data Line Status	Reading LSR

Line Control Register (LCR) - I/O Address: Base+3

This register is used to control the format of data transmitted or received (for example, parity type, number of stop bits).

Bit	Description
0-1	Two bit code for word length (0=5, 1=6, 2=7, 3=8 bits).
2	Number of stop bits (0=1, 1=2).
3-5	Parity Type (0=None, 1=Odd, 3=Even, 5=Mark, 7=Space; other values unused).
6	Asserts Break Signal when set.
7	DLAB bit. Allows access to bit rate divisor when set. When reset access is allowed to RBR , THR and IER registers.

Modem Control Register (MCR) - I/O Address: Base+4

This register is used to control the state of various modem control signals, such as the **DTR** and **RTS** signals.

Bit	Description
0	DTR (Data Terminal Ready). Signals modem that PC is active.
1	RTS (Request to Send). Signals modem that you can accept data
2	OUT1 not used by PC.
3	OUT2 must be set to enable interrupts from the UART.
4	LOOP is used to place UART in diagnostic loopback mode.
5-7	Unused.

Line Status Register (LSR) - I/O Address: Base+5

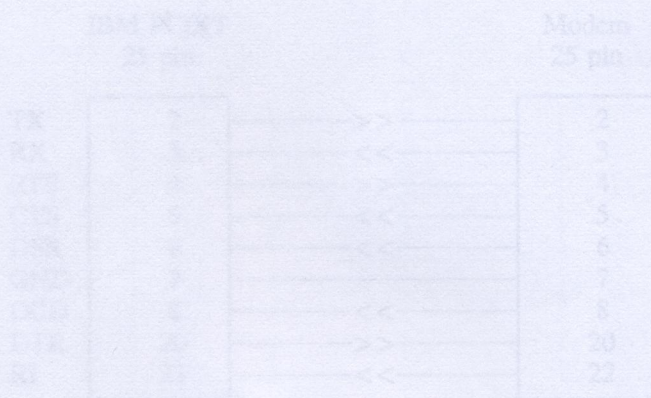
This register is used to test the status of various line conditions (and errors). The register's bits represent the following information.

Bit	Description
0	Data Ready. If set, data is available to be read from RBR .
1	Overrun Error
2	Parity Error
3	Framing Error
4	Received Break Signal
5	THR Empty. If set, data may be placed in THR for transmission.
6	Transmitter Shift Register Empty.
7	Unused.

Modem Status Register (MSR) - I/O Address: Base+6

This register is used to test the status of various modem signals. The register's bits represent the following:

Bit	Description
0	Delta CTS. The CTS signal has changed state
1	Delta DSR. The DSR signal has changed state
2	Trailing Edge Ring Indicator. The modem detected a ring signal.
3	Delta DCD. The DCD signal has changed state
4	Clear to Send (CTS). The current state of the CTS signal.
5	Data Set Ready (DSR). The current state of the DSR signal.
6	Ring Indicator (RI). The current state of the RI signal.
7	Data Carrier Detect (DCD). The current state of the carrier.



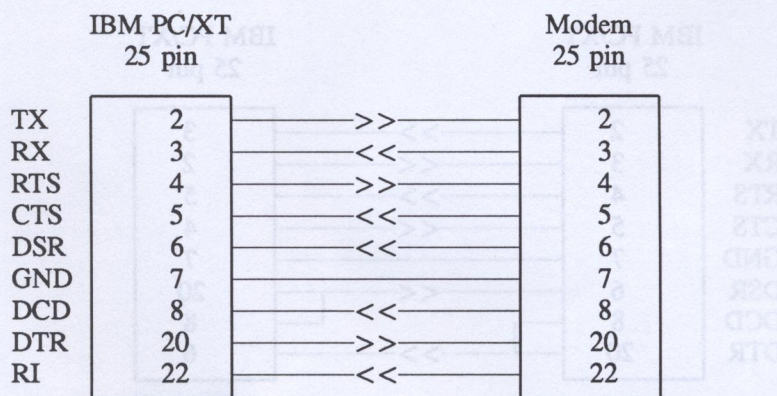
Connecting an AT to a modem is more difficult, because no modems on the market come with AT style 9-pin connectors. The following cable will do the trick, however.

Appendix B

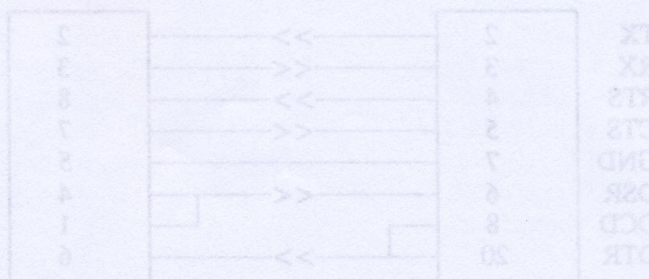
Cabling Information

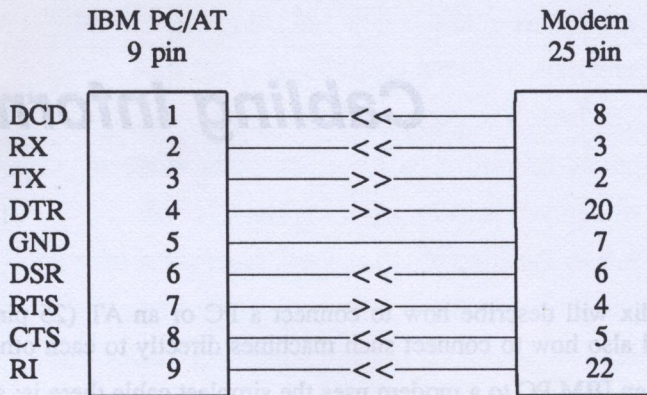
This appendix will describe how to connect a PC or an AT (25 pin or 9 pin) to a modem, and also how to connect such machines directly to each other.

Connecting an IBM PC to a modem uses the simplest cable there is: a straight ribbon cable with snap on connectors. If you don't have such a cable available. Then the following cable should be the minimum full specification cable for a PC. The arrows on the connectors show the direction of the signal.

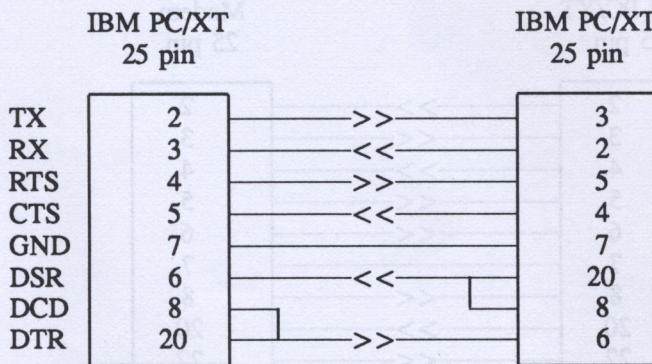


Connecting an AT to a modem is more difficult, because no modems on the market come with AT-style 9-pin connectors. The following cable will do the trick, however.

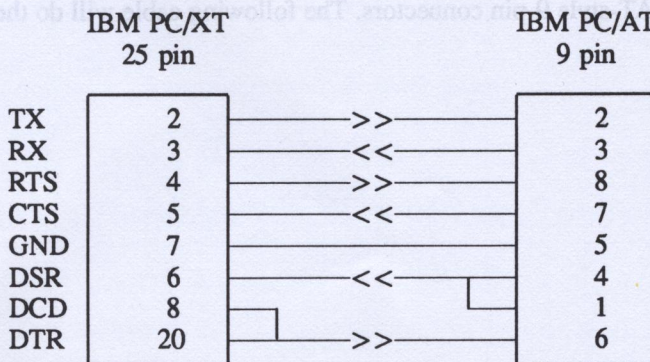




The remaining cables are all “null modem” cables, for connecting computers together directly. The first diagram shows how to connect a PC to a PC.



The next cable can be used to connect a PC to an AT.



Index

8250 UART, 71

AbortRequested, 17

AddOption, 52

ANSI emulation, 66

AppTimerExpired, 60

ASCII Protocol, 11

ASCIISend, 17

Block, 12

Cabling, 77

Cancel, 16

CarrierDetect, 7

ChangeSetup, 48

CheckHistory, 62

Checksum, 12

Checksum Byte, 18

ClearScreen, 68

CloseStatusWindow, 16

CommDemo

Commands, 46

Module, 45

Program, 45

CommUtil Module, 61

CompleteFilename, 16

ComSetup Module, 47, 53

Configuration File, 46

Control String, 55

CRC (Cyclic Redundancy Check),
12

Data Bits, 6

DEC Key Sequences, 67

DispMenu, 52

Divisor Latch Least, 72

Divisor Latch Most, 72

DoDownload, 65

DoUpload, 65

Downloading, 16

EmulateANSI, 69

EmulateVT100, 69

EmulateVT52, 69

Emulation, 68

Emulation

ANSI, 66

VT100, 49, 66, 67

VT52, 66

ErrorMessage, 64

FCS (Frame Check Sequence),
12

Field specifiers, 56

FileExists, 17

FileNames Module, 48

Flow control

Hardware, 6

RTS/CTS, 6

Software, 6

XON/XOFF, 6

FlushInput, 17

Formatting Directive, 56

Frame Check Sequence (FCS),
12

FTSupport Module, 14

Full Duplex, 13

GetHistoryString, 63

Half Duplex, 12

InitMenu, 52

InitPort, 6

InitScriptProcessor, 41

InitSetup, 48

InitUtil, 62

Install, 7

Interrupt Enable Register, 72

Interrupt Identification Register,
73

Kermit, 32

KermitReceive, 33

KermitSend, 33

Keyboard Driver, 49

Keyboard Module, 49

KeyPressed, 68

KillMenu, 53

Line Control Register, 73

Line Status Register, 74

LoadScript, 41

Logging, 47

MakeFilename, 49

MenuDrive, 53

MenuLib Module, 48, 50

Menu

 Popup, 50

Modem Control Register, 74

Modem Status Register, 74

Module

 CommDemo, 45

 CommUtil, 61

 ComSetup, 47, 53

 Filenames, 48

 FTSupport, 14

 Keyboard, 49

 MenuLib, 48, 50

 PRINTF, 53, 54

 RS232, 5

 Scanner, 40

 ScriptCompiler, 40

 ScriptProcessor, 40

SPRINTF, 48, 53

Term, 66

Timer, 59

Window, 51

NewFilename, 15

OpenStatusWindow, 15

Output

 Formatting Directives, 56

Packet, 12

 Layout, 12

 WXmodem, 28

 Xmodem, 18

Parity, 6

ParseFilename, 49

Pin Connections, 77

Popup Menu, 50

Printer, 64

PRINTF Module, 53, 54

PrintF-Related Routines, 54

Protocol, 11

 Full Duplex, 13

 Half Duplex, 12

 Large Packet, 13

 Streaming, 13

 Windowing, 13, 29

 Wmodem, 13

 Xmodem, 12, 13, 17

 Xmodem-1k, 22

 Ymodem, 13

 YModem-g, 13

 Ymodem/Batch, 22

PushBack, 66

RdKey, 68

Read, 62

ReadLine, 66

ReadString, 66

Receive Buffer Register, 71

Remote System, 6

RepaintMenu, 53

RS232 Module, 5

RTS/CTS Flow Control, 6

- SaveSetup**, 48
- Scanner Module**, 40
- Screen**, 64
- Script**, 35
 - Commands, 37
 - Example, 42
 - Processing Modules, 40
 - Syntax and Commands, 35
- ScriptCompiler Module**, 40
- ScriptProcessor Module**, 40
- SelectFile**, 65
- SelectPort**, 5
- SendString**, 64
- SerialRead**, 8
- SerialWrite**, 7
- SetAppTimer**, 60
- SetBreak**, 8
- SetFlowControl**, 6
- SetLogMode**, 63
- SingleStep**, 42
- SPRINTF Module**, 48, 53
- SPrintF-related routines**, 59
- StartClock**, 60
- StatusLine**, 62
- Stop Bits**, 6
- StopClock**, 60
- StoreChar**, 63
- StoreHistory**, 63
- Streaming Protocol**, 13
- Switch Character**, 58
- Term Module**, 66
- TERM.CFG**, 46
- TerminateScript**, 42
- Timer Module**, 59
- Transmit Holding Register**, 71
- TransmitMacro**, 64
- UART**, 71
- UART Register**
 - Divisor Latch Least, 72
 - Divisor Latch Most, 72
 - Interrupt Enable, 72
 - Interrupt Identification, 73
 - Line Control, 73
 - Line Status, 74
 - Modem Control, 74
 - Modem Status, 74
 - Receive Buffer, 71
 - Transmit Holding, 71
- UnInstall**, 7
- UpdateStatus**, 16
- UpdateTimer**, 60
- VT100**
 - Emulation, 49, 66, 67
 - Terminal, 66
- VT100/VT52/ANSI Terminal**
 - Emulation, 66
- VT52**
 - Emulation, 66
 - Terminal, 66
- Window Module**, 51
- Windowed Xmodem**
 - (WXModem), 28
- Windowing Protocol**, 13, 29
- Wmodem Protocol**, 13
- WrChar**, 67
- WXmodem Packet**, 28
- WXModemReceive**, 32
- WXModemSend**, 32
- Xmodem Packet**, 18
- Xmodem Protocol**, 12, 13, 17
- Xmodem-1k Protocol**, 22
- XModemReceive**, 21
- XModemSend**, 22
- XON/XOFF Flow Control**, 6
- Ymodem and its Variants**, 22
- Ymodem Batch**, 23
- Ymodem Packet**, 22
- Ymodem Protocol**, 13
- YModem-g Protocol**, 13
- Ymodem-g**, 25
- Ymodem/Batch Protocol**, 22
- YModemBatchReceive**, 27
- YModemBatchSend**, 28



Jensen & Partners UK Ltd.
63 Clerkenwell Road
London EC1M 5NP
England

Jensen & Partners International, Inc.
1101 San Antonio Road, Suite 301
Mountain View, CA 94043
USA