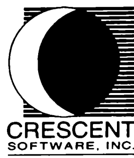
Professional Programming Tools
for BASIC Compilers

Crescent Software's

PDQComm

Owner's Manual



PDQComm
Communications Support for P.D.Q. And Microsoft BASIC

Entire contents Copyright ©1990-1994 by David Cleary, Ethan Winer and Crescent Software.

PDQComm was written by David Cleary, with contributions by Nash Bly and Ethan Winer. This manual was written by Ethan Winer and Dave Cleary, and was designed and typeset by Jacki W. Pagliaro.

No portion of this software or manual may be duplicated in any manner without the written permission of Crescent Software.

All tradenames referenced herein are property of their respective owners.

Crescent Software
11 Bailey Avenue
Ridgefield, CT 06888
(203) 438-5300

2nd Reprint – March, 94

LICENSE AGREEMENT

Crescent Software grants a license to use the enclosed software and printed documentation to the original purchaser. Copies may be made for back-up purposes only. Copies made for any other purpose are expressly prohibited, and adherence to this requirement is the sole responsibility of the purchaser. However, the purchaser does retain the right to sell or distribute programs that contain PDQComm routines, so long as the primary purpose of the included routines is to augment the software being sold or distributed. Source code and libraries for any component of the PDQComm program may not be distributed under any circumstances. This license may be transferred to a third party only if all existing copies of the software and documentation are also transferred.

WARRANTY INFORMATION

Crescent Software warrants that this product will perform as advertised. In the event that it does not meet the terms of this warranty, and only in that event, Crescent Software will replace the product or refund the amount paid, if notified within 30 days of purchase. Proof of purchase must be returned with the product, as well as a brief description of how it fails to meet the advertised claims.

CRESCENT SOFTWARE'S LIABILITY IS LIMITED TO THE PURCHASE PRICE. Under no circumstances shall Crescent Software or the authors of this product be liable for any incidental or consequential damages, nor for any damages in excess of the original purchase price.

Table of Contents

Table of Contents

Chapter 1: Introduction

Introduction 1-1
About This Manual 1-2
Installing PDQComm 1-2
PDQComm Overview 1-2
What's On The PDQComm Disk 1-3
P.D.Q. Routines In PDQComm 1-4
Differences Between PDQComm 2.x And Previous Versions 1-5
Enhancements 1-5
New Routines 1-5

Chapter 2: Using PDQComm

Using Integers 2-1
PDQComm Functions 2-1
Compiling and Linking With PDQComm 2-1
Combining Quick Libraries 2-2
Differences Between BASIC and PDQComm 2-3
The BIN, ASC, and LF Options 2-3
The TB[n] Option 2-4
The CD[n], CS[n], DS[n], and OP[n] Options 2-4
The RS Option 2-4
PDQComm Syntax 2-4
Handshaking and The Receive Buffer 2-6
Writing A BBS Program with PDQComm 2-8
ComInput$ Vs. ComLineInput 2-10
Printing Delays When Using P.D.Q. 2-10
Using PDQComm In A P.D.Q. TSR Program 2-11
Pdqcomm Terminal Emulations 2-11
Locate 2-15
Color 2-15
Cls 2-15
Using Multiple Windows 2-16
Transferring Files With PDQComm 2-17

Chpater 3: Functions and Subroutines

AdjustRecBuffer (Subroutine) 3-1
ASCIIReceive (Function) 3-2
ASCIISend (Function) 3-3
BIOSInkey (Function) 3-3
BIOSPrint (Subroutine) 3-4
Carrier (Function) 3-4
Checksum (Function) 3-5
CloseCom (Subroutine) 3-6
ComEof (Function) 3-6
ComInput$ (Function) 3-7
ComLineInput (Subroutine) 3-8
ComLoc (Function) 3-9
ComPrint (Subroutine) 3-9
CRC16 (Function) 3-10
DTR (Subroutine) 3-10
FlushBuffer (Subroutine) 3-11
GetComPorts (Subroutine) 3-11
GetLineStatus (Subroutine) 3-12
GetPortConfig (Subroutine) 3-13
OneColor (Function) 3-14
OpenCom (Subroutine) 3-14
OpenComX (Subroutine) 3-16
OverRun (Function) 3-17
ParseComParam (Subroutine) 3-18
Pause (Subroutine) 3-19
PDQExist (Function) 3-19
PDQParse (Function) 3-20
PDQPrint (Subroutine) 3-21
PDQRestore (Subroutine) 3-22
PDQTimer (Function) 3-22
PDQValI and PDQValL (Functions) 3-23
RTS (Subroutine) 3-23
ScanCodes% (Function) 3-24
SendBreak (Subroutine) 3-24
SetActivePort (Subroutine) 3-25
SetCom (Subroutine) 3-26
SetComPrintTO (Subroutine) 3-27
SetDelimitChar (Subroutine) 3-28
SetFIFO (Subroutine) 3-28
SetHandshaking (Subroutine) 3-29
SetMCRExit (Subroutine) 3-29

SetxxxxWindow (Subroutines) 3-30
XModemReceive (Function) 3-30
XModemSend (Function) 3-31
UARTType% (Function) 3-32
XOff (Function) 3-33
xxxxInit (Subroutines) 3-33
xxxxPrint (Subroutines) 3-34

Chapter 4: Communications Tutorial

An Introduction To Serial Communications 4-1
Synchronous And Asynchronous Communications 4-1
Communicating By Bits 4-2
The Parity Bit 4-2
Bidirectional Communications 4-3
Baud Versus Bits Per Second 4-3
RS-232C 4-4
DCE And DTE Devices 4-4
Cables, Null Modems, And Gender Changers 4-7
MODEMS 4-10
UARTs 4-11
The INS8250-B UART 4-11
The INS8250A And NS16450 UARTS 4-13
The NS16550/NS16550A UART 4-13
Summary 4-14

Appendices

Appendix A

Standard Hayes “AT”Commands A-1
S-Registers A-3

Appendix B

Terminal Emulation Control Codes B-1
TTY Control Codes Recognized by TTYDISP.BAS B-1
ANSI Control Codes Recognized by ANSIDISP.BAS . . . B-1
Data General D215 Codes Recognized by D215DISP.BAS B-2
VT52 Control Codes Recognized by VT52DISP.BAS . . . B-3
VT100 Control Codes Recognized by VT10DISP.BAS . . B-3

If VT52 Compatible Mode B-3
If ANSI Compatible Mode B-4

INDEX

Chapter 1: Introduction

Introduction

Thank you for purchasing PDQComm. We have made every effort to provide you with software and documentation that is both effective and easy to use. If you have a comment, a complaint, or perhaps a suggestion for another QuickBASIC-related product, please let us know. We want to be your *favorite* software company.

Before we begin discussing the contents of the PDQComm disk and manual, please take a few moments to fill out the enclosed registration card. Doing this entitles you to free technical support by phone, as well as ensuring that you are notified of possible enhancements and new products. Many upgrades are offered at little or no cost, but we can't tell you about them unless we know who you are! Note, however, that if you purchased PDQComm directly from us, the mail-in portion of the registration card may have been removed. In that case, you are already registered.

Please mark the PDQComm product serial number on your disk label or manual cover. License agreements and registration forms have an irritating way of becoming lost, and doing this will insure that the number is handy if you need to contact us. You may also want to note the product version number in a convenient location; this is stored on the distribution disks in the volume label. If you ever have occasion to call us for assistance, we will need to know your serial number, and probably the version you are using as well. To determine the version number for any Crescent Software product, simply use the DOS VOL command, which will display the disk volume label:

```
VOL A:
Volume in drive A is PDQComm 2.XX
```

Although you can call for assistance, we also maintain a bulletin board system (BBS), which is operated by Dave Cleary, the author of PDQComm. You are encouraged to log on by calling 203-426-5958 at any baud rate (up to 2400 baud) using a N, 8, 1 protocol.

We are constantly improving all of our products, and you may want to call us periodically and ask for the current version number. Major upgrades are always announced, however minor additions or fixes are generally not. If you are having any problems at all—even if you are sure it is not caused by one of our products—please call us. We support all versions of QuickBASIC, and can often provide better assistance than Microsoft.

About This Manual

This manual is divided into four sections—an overview which describes what PDQComm is all about, a brief tutorial about writing communications programs in general, a tutorial section on using PDQComm, and a reference section that describes each of the PDQComm routines. Besides the information in this manual, there are several files that contain additional information you may find useful. Perhaps most important, we include a few different BASIC terminal programs that show the PDQComm routines in context. We also provide all of the assembly language source code for the entire package. You will find this invaluable if you are interested in learning about communications in general, and about assembly language in particular.

Installing PDQComm

The .EXE files on the distribution disk are self-extracting archive programs which contain all of the files for PDQComm. To extract the various PDQComm program files you should place this disk into drive A, and then log onto the drive and directory in which you want the files to be installed. If you are using PDQComm with our P.D.Q. product, we suggest that you install these files into the same directory that contains P.D.Q. Once you have created and changed to that directory, simply run the PDQCOMM.EXE program from Drive A as follows:

```
A:PDQCOMM
```

This will extract all of the libraries, sample programs, and .MAK files, and write them to the current directory. If you also want to extract the assembly language source files you should create a separate directory, and run the SOURCE.EXE program as well.

Please note that there are a few "empty" files with names like "-" and "--" that serve as separators between logically grouped sets of files. Because these files have a zero length they do not take up any disk space other than the 32 bytes used by all directory entries. However, most hard disk tune-up programs will not move zero-length files, so you may want to delete them after you have installed PDQComm.

PDQComm Overview

PDQComm is a set of low-level routines that add communications capabilities to programs written using Microsoft QuickBASIC version 4.0 or later. PDQComm was originally designed to be used with our P.D.Q. replacement link library, because the P.D.Q. OPEN statement does not support a communications argument. Therefore, these tools are necessary

to obtain that feature with P.D.Q. However, PDQComm is also ideal for use with regular QuickBASIC programs, to avoid the need for ON ERROR. Many programmers prefer to avoid ON ERROR when possible, because of the code size and speed penalties that ON ERROR imposes.

Further, PDQComm offers several enhancements not present in QuickBASIC, such as both hardware and software handshaking, support for COM3 and COM4, file transfer protocols, and various terminal emulations. PDQComm also lets your program change the baud rate and other parameters while the communications port is open. Finally, all of the PDQComm routines have been designed to emulate the syntax of the QuickBASIC routines they replace as closely as possible. For example, to determine the current buffer location you would use the ComLoc function, as opposed to BASIC's LOC.

What's On The PDQComm Disk

A number of files are provided with PDQComm, such as libraries, Include files, and BASIC demonstration programs. All of the files that are present on the distribution disk are described briefly in the section that follows.

README, if present, contains important information that is not in this printed manual. In many cases, this information is about new features or other enhancements you should know about.

PDQCOMM.LIB is the linking library that contains all of the assembly language routines in PDQComm.

PDQCOMM.RSP is the LIB.EXE response files we used to create PDQCOMM.LIB.

COMMQB4.RSP is the LIB.EXE response file we used to create COMMQB4.LIB.

COMMBC7.RSP is the LIB.EXE response file we used to create COMMBC7.LIB.

COMMQB4.LIB contains the same routines that are in PDQCOMM.LIB, but some of the routines have slight changes to allow them to operate in a Quick Library, with regular QuickBASIC, and with BASIC 7 when using near strings.

COMMQB4.QLB is a quick library for use in the QB 4.x environment.

COMMBC7.LIB contains the same routines that are in PDQCOMM.LIB, but they are meant for use with BASIC 7 when using far strings.

COMMBC7.QLB is a quick library for use in the QBX environment that comes with BASIC 7 PDS.

ASMINFO.DOC provides information about PDQCOMM that is of interest to assembly language programmers.

COMMDECL.BAS contains DECLARE and TYPE statements for the various PDQComm routines, and is meant to be loaded as an Include file in your programs.

DEMOTERM.BAS is a simple terminal program that illustrates the PDQComm routines in context. DEMOTERM.BAS was written by Crescent friend Nash Bly.

TSRTERM.BAS is a TSR version of DEMOTERM.BAS, and it shows how to implement communications in a TSR program.

MSTERM.BAS is a modified version of the TERMINAL.BAS example program provided by Microsoft, but adapted to use the PDQComm routines. This program serves as an example of translating BASIC communications statements for use with PDQComm, and it also may be run within the QB editor.

ANSITERM.BAS is another more elaborate program by Dave Cleary, who also wrote PDQComm. AnsiTerm offers ANSI terminal emulation along with XMODEM file transfer capabilities. AnsiTerm requires modules from PDQComm's QuickBASIC routines, as specified in the ANSITERM.MAK file.

***xxxx*DISP.BAS** are the terminal emulation routines. *xxxx* is either TTX, ANSI, D215, VT52, or VT10.

ASCIIXFR.BAS contains the ASCII file transfer routines.

XMODEM.BAS contains the XModem file transfer routines.

XSTAT.BAS is a routine that the file transfer routines call to report the status of the transfer.

All of the remaining files with an .ASM or .INC extension comprise the assembly language source code for PDQComm.

P.D.Q. Routines In PDQComm

Although PDQComm was written primarily for use with our P.D.Q. product, it will work quite nicely with regular QuickBASIC and BASIC 7 PDS. However, we have included several P.D.Q. extensions with this

library for those people who do not own P.D.Q. A complete discussion of these routines is given in the reference section that describes the PDQComm routines.

Differences Between PDQComm 2.x And Previous Versions

All programs written for PDQComm 2.0x will work correctly under 2.5 without requiring any changes to your existing programs. We have added many new routines, and also added several enhancements. These are described in this section.

Enhancements

PDQComm now supports having two ports open at once. The only restriction is that both ports must be using separate IRQ (interrupt request) lines. This is not a software restriction, but rather a hardware limitation with the PC bus design. Therefore, you cannot open both COM1 and COM3 at the same time because they both share IRQ 4. Likewise, COM2 and COM4 share the same line (IRQ 3) and may not be used simultaneously.

PDQComm now restores the state of the communications port to what it had been when the port was opened, except the baud rate. This allows PDQComm to work correctly with programs such as FOSSIL drivers, even if the FOSSIL is "hot" when PDQComm opens the port.

XON/XOFF handshaking is now handled automatically by ComPrint in the same way RTS/CTS handshaking is. Therefore, the ClearXOff routine included with earlier versions of PDQComm is no longer necessary.

PDQComm now supports the NS16550 in FIFO mode. Please see SetFIFO for more information on this.

PDQComm now supports ports with non-standard interrupts. Please see OpenComX for more information about this.

PDQComm now correctly handles situations where COM2 is the only com port installed in the machine. If address 2F8h is located in the BIOS data area where COM1 should be, PDQComm will use IRQ 3 instead of 4. However, you must still use COM1 when opening the port.

New Routines

The following describes new routines that have been added to PDQComm since version 2.0. Several of these routines are from our P.D.Q. and

QuickPak Professional products, and are included here because of their usefulness.

AdjustRecBuffer (Subroutine) - Changes the size of the receive buffer along with the NearEmpty and NearFull points used in handshaking.

BIOSPrint (Subroutine) - This is similar to PDQPrint except it uses the BIOS to print instead of directly writing to screen memory.

GetLineStatus (Subroutine) - Allows you to obtain the status of all port lines with one routine.

OneColor% (Function) - OneColor% is from QuickPak Professional, and it combines a foreground and background color into a single value.

OpenComX (Subroutine) - Allows you to open ports that have non-standard addresses or use non-standard IRQs. This routine also provides a small-code way to open a standard port.

PDQExist% (Function) - File exist function from P.D.Q.

PDQPrint (Subroutine) - Fast print routine from P.D.Q. that writes directly to screen memory.

PDQTimer& (Function) - Small-code TIMER replacement from P.D.Q. that returns the number of timer ticks as a long integer.

ScanCodes% (Function) - This is used by the terminal emulator routines. It is like BASIC's INSTR, except it returns the position of control characters (ASCII values less than 32) in a string.

SetActivePort (Subroutine) - This routine changes the currently active port. That is, it specifies which port subsequent commands such as OpenCom and ComPrint are to use.

SetFIFO (Subroutine) - Used to enable or disable the NS16550A FIFO mode.

UARTType% (Function) - Returns the type of UART at a specified address.

SetComPrintTO (Subroutine) - Used to set the time-out value of the ComPrint routine.

Chapter 2: Using PDQComm

Using Integers

All of the PDQComm routines that accept numeric arguments expect integer values or variables. Since BASIC uses single precision variables by default, it is very important that you add either an explicit percent sign to each variable name, or place the statement DEFINT A-Z at the beginning of your program.

PDQComm Functions

Many of the PDQComm routines have been designed as functions as opposed to called subroutines, where returning a value is sensible. However, it is imperative that you declare these routines. Unlike called routines where a DECLARE statement is optional, external functions written in assembly language must always be declared. In the routines description portion of this manual, each is identified as either a subroutine or function. By including COMMDECL.BAS in your programs, this is all done for you automatically.

Compiling and Linking With PDQComm

Once you have compiled your BASIC program using the BC.EXE compiler, you must link it with the assembler routines in the PDQCOMM.LIB library file. This is very easy to do, and the example below shows how PDQComm would be linked to a P.D.Q. program.

```
LINK /noe /nod program , , nul , pdqcomm pdq
```

If you are using PDQComm with regular QuickBASIC then you would specify the COMMQB4.LIB file instead:

```
LINK program , , nul , commqb4
```

And if you plan to use PDQComm with BASIC 7 and want to take advantage of Far Strings, you would specify COMMBC7.LIB instead:

```
LINK program , , nul , commbc7
```

If you do not compile your program with BASIC 7 far strings (/FS) you must use COMMQB4.LIB.

Of course, LINK lets you specify more than one library on its command line, so to link with our QuickPak Professional library you would specify that as well:

```
LINK program , , nul , commqb4 pro
```

Combining Quick Libraries

If you need to combine PDQComm with another library to create a single quick library, you must use LINK as shown below. This example shows how to combine PDQComm and QuickPak Professional for use with QuickBASIC:

```
LINK /q /seg:500 commqb4.lib pro.lib , , nul , bqlb45.lib
```

The /seg switch is required because of the number of routines contained in QuickPak Professional. If you are using QuickBASIC 4.0, the last file on the link line would be BQLB40. If you are creating a quick library for the QBX environment, you would use COMMBC7.LIB and PRO7.LIB respectively, and QBXQLB instead of BQLB45 as the support file at the end of the LINK command line.

Finally, if you want to create a Quick Library that includes all of the P.D.Q. extensions as well as the PDQComm routines, you should edit the QUICKLIB.RSP file and add COMMQB4.LIB to the top of the list as follows:

```
COMMQB4.LIB +
Absolute    +
BIOSInky    +
BIOSInpt    +
  .
  .
```

Then, simply run the QUICKLIB.BAT batch file to create the Quick Library. Please ignore the duplicate definition errors you receive. These are due to the same routine names being present in both the PDQCOMM and PDQ library files.

Regardless of the BASIC version you are using, you would start the QB editor using the /L (Library) option, specifying the Quick Library you just created. For QuickBASIC 4 and 4.5 do this:

```
QB [program] /L commqb4.qlb
```

And for BASIC 7 (QBX) do this:

```
QBX [program] /L commbc7.qlb
```

Differences Between BASIC and PDQComm

The OpenCom routine in PDQComm emulates the syntax of QuickBASIC's OPEN "COM" as closely as possible. However, there are some arguments that OpenCom does not support. To open a communications port in BASIC you use the OPEN statement specifying a port, a baud rate, the type of parity, and the number of data and stop bits. You may also provide one or more optional parameters as shown below.

```
OPEN "COM1:9600,N,8,1[,parameters]"
```

The relationship between these optional parameters and the equivalent PDQComm arguments is described below.

The BIN, ASC, and LF Options

These parameters are used in QuickBASIC to specify the mode the port is opened with. The "BIN" (binary mode) option is BASIC's default, and this is the only mode that PDQComm supports. In binary mode the data is transmitted verbatim—Tab characters are not expanded to blanks, and a CHR$(26) is not recognized as an EOF (end of file) marker.

In BASIC, using an "ASC" argument opens the port for ASCII mode which, in addition to expanding Tabs and recognizing EOF, also sends a carriage return after every 80 characters automatically.

The BASIC "LF" option is used in conjunction with "ASC", and causes a CHR$(10) line feed to be sent after every carriage return. PDQComm does not support either ASC or LF because of their limited usefulness.

ASC and LF are generally used when sending data to a serial printer. Of course, you can easily simulate those features by printing a carriage return and line feed manually when needed:

```
CRLF$ = CHR$(13) + CHR$(10)
CALL ComPrint("Test message" + CRLF$)
```

We recommend thay you define CRLF$ once at the beginning of your program, to avoid repeated calls to BASIC's CHR$ function later. Each use of CHR$ adds six bytes, and each string concatenation requires 13 more.

The TB[n] Option

This parameter specifies the size of the transmit buffer. PDQComm currently does not support buffered transmission, and therefore cannot honor this parameter.

The CD[n], CS[n], DS[n], and OP[n] Options

In BASIC these parameters are used to set the timeout period for the specified lines to become active when the port is opened. These parameters often cause more trouble than they are worth, so PDQComm leaves it up to the programmer to determine whether checking these lines is important. The OP option sets the timeout period for all the lines to become active, while the CD, CS, and DS options let you control each line individually. To check the state of these lines with PDQComm you would use the GetLineStatus subroutine after the port has been opened. You can then take appropriate action based on what the line states are.

The RS Option

This BASIC option causes the RTS line to not become active when the port is opened. It is hard to envision a situation in which this would be useful, but to achieve the same result using PDQComm, you would call the RTS subroutine after opening the port.

PDQComm Syntax

Because the syntax for the various PDQComm routines is modeled after the equivalent statements and functions in QuickBASIC, designing a communications program using PDQComm will be very similar to doing the same with QuickBASIC. The short program below shows the minimum steps necessary to create a fully functioning terminal program with PDQComm.

Please understand that the example programs we will be presenting herein are not intended to teach you everything you need to write a communications program. Rather, our purpose is to show how the PDQComm routines are used. If you are unfamiliar with communications programming in general, we suggest that you become familiar with the various PDQComm example programs.

Besides using a syntax as close to BASIC's as possible, PDQComm also returns error information in the BASIC ERR function. This lets you easily test the success or failure of the most recent operations, without requiring ON ERROR. Unlike other communications libraries you may have seen,

PDQComm uses the minimum number of parameters possible. This greatly reduces the size of your programs, and also improves their speed.

```
DEFINT A-Z                                  'all integers please

DECLARE SUB OpenCom (Action$)               'same as OPEN "COM..."
DECLARE SUB ComPrint (Work$)                'same as PRINT #n,
DECLARE SUB CloseCom ()                     'same as CLOSE #n
DECLARE FUNCTION BIOSInkey% ()              'similar to ASC(INKEY$)
DECLARE FUNCTION ComEof% ()                 'same as EOF(n)
DECLARE FUNCTION ComInput$(NumChars)        'same as INPUT$(n,#n)
CALL OpenCom("COM1:2400,N,8,1,RB512,XON") 'open the port
IF ERR THEN                                 'oops
  PRINT "Error opening the communications port."
  END
END IF

DO
  Char = BIOSInkey%                         'get what was typed
  IF Char = 27 THEN EXIT DO                 'exit if it was Escape
  IF Char THEN CALL ComPrint(CHR$(Char))    'anything else, send it
  IF NOT(ComEof%) THEN                      'was anything received?
     ComString$ = ComInput$(ComLoc%)        'yes, get the characters
     PRINT ComString$;                      'print them on the screen
  END IF
LOOP                                        'loop forever

CALL CloseCom                               'close the port and end
```

Here, the OpenCom routine is called specifying communications port 1 at a baud rate of 2400, no parity, 8 data bits, and 1 stop bit. RB512 specifies a receive buffer size of 512 bytes, and XON indicates that XON/XOFF handshaking is to be performed automatically. We will discuss handshaking as well as selecting an appropriate buffer size in a moment.

Once the communications port has been opened, an "endless" loop is entered that alternately checks for keyboard activity and characters being received. If a character has been entered at the keyboard it is sent through the port (unless it was the Escape key). And if any characters have been received and are waiting to be read, the ComInput$ function reads all of them from the receive buffer.

The ComEof% function is used to determine if any characters are waiting in the buffer, ComLoc% reports how many, and ComInput$ does the actual reading. Notice that these functions are identical to the equivalent BASIC functions EOF(), LOC(), and INPUT$ respectively.

The BIOSInkey function is much more efficient than the regular QuickBASIC INKEY$, because it returns the integer ASCII value of the

key that was pressed. This requires less code in the BASIC program, since integer assignments and comparisons are simpler than the equivalent string operations.

Handshaking and The Receive Buffer

In any communications program, data is sent in a continuous stream from one PC or terminal to another. As characters are received, they are placed into a special area of memory called a receive buffer. Then, when the receiving program is ready to read that data, it reads from the buffer rather than directly from the port. This is an important concept, since it frees the receiving program from having to continually poll the communications port. Of course, all of this is handled transparently for you by the various PDQComm routines. That is, as data is received by the port it is placed into the buffer automatically through a system of hardware interrupts. And when your program asks to read those characters, they are taken from the buffer.

A receive buffer of 1K is more than adequate if you are using a baud rate of 2400 bps or less, or if handshaking is employed. Handshaking means that the sender and receiver use a system of flags to tell each other when they are ready. Without handshaking, the sending program will simply continue to transmit, and the receiving program must be prepared to accept the characters as quickly as they arrive. When handshaking is employed, however, a smaller buffer may be used because the receiving program can tell the sender to stop transmitting before the buffer overflows.

If you are not using handshaking and there may be delays in your program, for example when you write data to a disk file, then you should specify a larger receive buffer. Microsoft uses 512 bytes as a default buffer size for QuickBASIC, but very high baud rates require a larger buffer. Some hardware interface devices do not support handshaking, so a larger buffer would be needed in that case as well.

PDQComm supports two types of handshaking—XON/XOFF and RTS/CTS—and each of these methods will be described in turn. Of course, regardless of the handshaking method you choose, the remote terminal you are communicating with must also be using the same method. Now let's look at the handshaking options PDQComm provides.

XON/XOFF is sometimes called software handshaking because it is done by sending special characters between systems. (Ctrl-S tells the sender to stop transmitting, and Ctrl-Q means it is okay to resume.) Software handshaking is used mostly with modems because there isn't a direct hardware connection between the two computers. When the receive buffer

becomes nearly full, the PDQComm routines will automatically send an XOFF character to the remote system to tell it to stop transmitting.

After you have emptied the buffer with the ComInput$ function, an XON character will be sent to the remote system, so it knows that it is okay to resume sending. PDQComm handles this transparently for you so you don't have to worry about how many characters are in the buffer at any given time. Likewise, if the remote system sends an XOFF character, PDQComm will set the PDQComm XOff function to TRUE to let you know that.

It is important not to use XON/XOFF handshaking if you are transferring binary files, because the binary information may happen to contain those control characters. You can enable and disable handshaking after the port is opened using the SetHandshaking routine.

The other handshaking method supported by PDQComm, RTS/CTS, is also called hardware handshaking. This is because it is implemented using two physical lines connected between both systems. When the receive buffer is almost full, the receiving computer will change the state of its RTS line, which tells the remote system to stop transmitting. As with the XON/XOFF method, PDQComm handles RTS/CTS handshaking transparently for you. If you try to print while the remote has its CTS line high, you will get a "Timeout" error (error 24). The only way to print if the remote has its CTS line high is to disable RTS/CTS handshaking with the SetHandshaking routine. If you get the error while printing a string, ComPrint will remember where in the string you were, and will resume from that point when you subsequently send the same string again later. The example that follows shows this in context.

```
Text$ = "This is being sent through the communications port."
Retries = 0

DO
  CALL ComPrint(Text$)            'attempt to print the string
  IF ERR  24 THEN EXIT DO         'anything but a time-out
                                  ' error is okay
  Retries = Retries + 1           'count how many times it timed out
  IF Retries  5 THEN              'more than five means it's
                                  ' hopeless
     PRINT "I couldn't print the string."
     EXIT DO
  END IF
LOOP
```

Presently, PDQComm does not support both hardware and software handshaking at the same time. If you actually need to do this, then use RTS handshaking and look for XON/XOFF characters manually.

Writing A BBS Program with PDQComm

The following is not intended to serve as a complete tutorial on writing a full-featured BBS (Bulletin Board System) program, however a number of useful tips and techniques will be discussed. PDQComm also includes several routines that will assist you in writing a BBS program, and these will be discussed as well. If you would like to see what is involved in writing a full-featured BBS program, the QuickBASIC source code to the popular RBBS-PC is available for download from the Crescent Software Support at BBS (203) 426-5958.

Perhaps most important, when writing a BBS it will be up to you to echo back everything you receive. For example, when you log on to a commercial BBS you will observe that everything you type also appears on your screen. This is because the characters you enter first go from your computer to the BBS, and then are sent back to your computer. Your PC displays everything that it receives, which of course includes those characters that you transmitted.

One important PDQComm helper is the GetPortConfig routine, which lets you determine the current port parameters. For example, if you are using a front end mailer such as BinkleyTerm and a human calls, BinkleyTerm will transfer control to your BBS program. But how do you know at what baud rate the caller logged on with? The GetPortConfig routine will retrieve the current configuration for any valid communications port. It uses a TYPE variable called ModemType, which is shown below. (This TYPE is defined in the Include file COMMDECL.BAS. You should always include this file in your programs, because it also contains DECLARE statements for all of the PDQComm routines.)

```
TYPE ModemType
     Baud AS LONG
     DBits AS INTEGER
     Parity AS STRING * 1
     SBits AS INTEGER
END TYPE
```

You would therefore call GetPortConfig with a variable that has been dimensioned as ModemType, and then read the current port parameters from that variable. This is shown in the routine description for GetPort-Config.

PDQComm also provides a routine called SetCom. Like GetPortConfig, SetCom uses the ModemType variable, and it lets you change the parameters of a port that is already opened, without having to close it first. SetCom is useful if your BBS answers the phone, and the caller logs on

with a different baud rate. You would merely look at the "CONNECT" string the sending modem transmits, and determine the baud rate from that. SetCom will then let you modify the baud rate you are using to accommodate the caller.

Be sure that your modem is configured to return a "verbose" CONNECT string, so the program will be able to read the baud rate from it. (Our Practical Peripherals modems use the "ATX4" command for this, which is the factory default—please consult your modem owner's manual for the correct command.)

```
    DIM Config AS ModemType                       'ModemType is in
                                                  ' COMMDECL.BAS
    ComParam$ = "COM1:2400,N,8,1,RB1024,XON"      'This is our open string
                                                  'Parse Config
    CALL ParseComParam(ComParam$, Port, Config, BufLen, Hand$)

    CALL OpenCom(ComParam$)                       'Open the port

Start:
    DO
      CALL ComLineInput(Ring$, 45)                'Wait for the phone to
                                                  ' ring
      IF BiosInkey% THEN                          'Quit if they press any
        CALL CloseCom                             ' key (this is optional)
        END
      END IF

      IF ERR = 0 THEN                             'If no time-out
        IF Ring$ = "RING" THEN EXIT DO            'The phone rang so
                                                  ' answer it
      END IF
    LOOP
    CALL ComPrint("ATA" + CHR$(13))               'Tell modem to answer
    DO
      CALL ComLineInput(Connect$, 45)
      IF INSTR(Connect$, "CONNECT") THEN EXIT DO 'We have a
                                                  '  connection
      IF Connect$ = "NO CARRIER" THEN GOTO START 'No connection,
                                                  ' start over
      IF ERR THEN
        CALL ComPrint(" ")                        'This forces modem to
                                                  ' hang up
        GOTO Start                                'If time-out, no
                                                  ' connection
      END IF
    LOOP
    Config.Baud = VAL(MID$(Connect$, 8))          'Modem returns baud
                                                  ' after CONNECT
    IF Config.Baud = 0 THEN Config.Baud = 300     '300 baud not reported
    CALL SetCom(Config)                           'Change port baud
```

Finally, you should always use the XON/XOFF handshaking method in a BBS program, since that is what your callers will be using.

ComInput$ Vs. ComLineInput

There are two popular methods for retrieving data from a communications port in BASIC. One is to use the INPUT$ function to read a specified number of characters at one time; this provides you with full control over what action is taken for each character. The disadvantage is that your program may need to combine the characters into a single string, which requires more effort than simply using LINE INPUT.

The other method is to use the LINE INPUT statement. This is simpler to use because it returns an entire line at once, and also removes the carriage return and line feed. However, BASIC's LINE INPUT suffers from a serious shortcoming when used with a communications port: control is not returned to your program until the terminating carriage return is received. If the phone connection is disrupted, or the sender fails to press Enter, then your computer will literally be disabled until you reboot it. Therefore, although convenient, LINE INPUT is not well suited for this use. In the context of a BBS system, using LINE INPUT also precludes you from being able to echo characters back to the sender.

PDQComm includes replacement routines for BASIC's INPUT$ and LINE INPUT, and in the latter case, we have added a unique "time-out" feature that lets your program regain control if a terminating carriage return is not received within a specified amount of time. When using ComLineInput, you tell it the number of seconds to wait for a carriage return, before timing out and exiting back to your program. If a carriage return is in fact received, then the entire line is returned. Otherwise, whatever has been received thus far is returned, and BASIC's ERR function is set to 24 which is the code for "Device timeout". ComLine-Input has several other useful features, and these are described in the routine description portion of this manual.

Printing Delays When Using P.D.Q.

When using the default P.D.Q. PRINT statement (as opposed to PDQPrint), it is important that you not print a string that is very long. Since P.D.Q. uses DOS for its printing (which is slow), it is possible that some of the data being received will be lost. This is because the DOS interrupt that PRINT uses disables the communications port interrupt while it is printing. This is especially important at fast baud rates, where many characters are received every second. We have found that limiting the length of a single string printed in one statement to 200 characters will prevent this from occurring.

Using PDQComm In A P.D.Q. TSR Program

When you call the OpenCom routine, it uses standard DOS services to allocate memory for its receive buffer. PDQComm uses DOS "far" memory, so its buffer will not impinge on BASIC's 64K string memory area. When you are using PDQComm in a non-TSR P.D.Q. program, the necessary memory is available and there are no problems. (Unless, of course, your program is so large or you have so many device drivers and TSR programs loaded that there really is insufficient memory for the receive buffer.)

However, most programs, including those written in regular BASIC, request all available DOS memory when they are loaded. This prevents DOS from honoring a memory request once the program is running. P.D.Q. programs release the unneeded memory back to DOS as part of their startup code, so this is not a problem when you are using P.D.Q. But when PDQComm is used with a P.D.Q. TSR program, extra steps must be taken to ensure that enough DOS memory will be available to satisfy the request by OpenCom. There are two possible solutions.

One is to open the communications port before calling EndTSR to stay resident, and this is the simplest choice when that is practical. The other solution is to manually allocate memory using AllocMem before calling EndTSR, and then relinquish that memory using ReleaseMem prior to calling OpenCom. If you plan to close the port and then reopen it later within the TSR, then you should call AllocMem again immediately after calling CloseCom. This way your program always "owns" at least enough memory for the receive buffer.

PDQComm Terminal Emulations

PDQComm offers a variety of windowed terminal emulations. You simply define a rectangular window on the screen, and PDQComm will handle displaying and scrolling all of the data in that window. Terminal emulations allow you to handle special control codes you may receive over the serial port; for example, codes that affect cursor position, screen colors, and other display attributes. Perhaps the best-known set of control codes on IBM computers are those recognized by the ANSI.SYS device driver. Since these emulation routines interpret and act on these codes for you, this precludes the use of BASIC's screen functions such as COLOR or LOCATE. The text that follows discusses how to simulate these functions. Note, however, that the overwhelming value afforded by emulating the ANSI control codes is that ANSI.SYS is not required for your programs to operate.

PDQComm includes several .BAS and .BI (BASIC Include) files to implement terminal emulation. TERM.BI defines a type variable that is used by all the emulations, and this variable controls all aspects of the emulation routines. A discussion of each component in this variable is given later on. The other .BI files contain declarations for each particular emulation, and also allocate a named COMMON SHARED variable for use by the routines. The .BAS files contain the code that actually performs each emulation.

All of the terminal emulation routines work in the same way, and each is comprised of only three routines. The routines are named *xxxx*Init, *xxxx*Print and Set*xxxx*Window, where *xxxx* indicates the type of terminal emulation. PDQComm currently offers the following emulations:

TTY - Generic emulation
ANSI - ANSI.SYS emulation
VT52 - Digital Equipment VT52 emulation
VT100 - Digital Equipment VT100 emulation
D215 - Data General D215 emulation

The source code for each emulation is contained in a file named *xxxx*DISP.BAS, where *xxxx* is the emulation name. For example, ANSIDISP.BAS holds the routines that perform ANSI emulation. Appendix B provides a control code reference for each of these emulations. EMULATE.DOC, if present, contains information on other emulations that may have been added after the printing of this manual.

To use the terminal emulations, you must use the '$INCLUDE metacommand to add two files to your program. The first include file is TERM.BI. This file defines a TYPE structure that is used by all of the emulations to control the various screen characteristics. The TYPE is defined as follows:

```
TYPE TermType
     Monitor        AS INTEGER
     Bios           AS INTEGER
     Fore           AS INTEGER
     Back           AS INTEGER
     TRow           AS INTEGER
     BRow           AS INTEGER
     LCol           AS INTEGER
     RCol           AS INTEGER
     CurRow         AS INTEGER
     CurCol         AS INTEGER
     DefFore        AS INTEGER
     DefBack        AS INTEGER
END TYPE
```

When you call the *xxxx*Init routine associated with the emulation you are using, default values are placed into each component of this variable.

Monitor tells if you have a color or monochrome monitor. The *xxxx*Init routine checks low memory to see what type of monitor is in use, and assigns zero if it is a color system, or non-zero for a monochrome system. To force monochrome, you may manually set this value to -1. This allows you to force black and white text on a color system if necessary. This would be desirable when the program is being run on some laptop computers, or on an older Compaq portable PC that has a CGA adapter connected to a monochrome monitor.

Bios tells the routines whether you want printing to be performed using direct screen writes, or through the BIOS. Setting this value to -1 (True) forces screen writes through the BIOS. The default is 0 (False) for direct screen writing. This could be important when running under some multitaskers such as DesqView or DoubleDOS to prevent bleed "through".

Fore and Back are the current foreground and background colors; the default values for these parameters is 7 and 0 respectively. You can change these values manually to force a color change, as opposed to using the COLOR statement. For example, to change the foreground color to bright blue you would use *typename.Fore = 9*, where *typename* is named according to the emulation being used.

TRow, BRow, LCol, and RCol define the window coordinates. By modifying these values, you can change the portion of the screen that is used to contain the displayed text. These values are initialized to default values of 1, 25, 1, and 80, which establishes the entire screen as a display window.

CurRow and CurCol are the row and column that the cursor is currently located at, and these values change as text is displayed. To manually set a new cursor position simply assign new values to these parameters, instead of using LOCATE. Note that these values are an offset from the current window. For example, if you have defined a window smaller than full screen, then the location 1,1 will be in the upper left hand corner of that window, and not necessarily at absolute location 1,1.

The DefFore and DefBack TYPE components are the default screen colors. These are used by the emulations that require default values, and are initialized to 7 for foreground and 0 for background.

The next file that must be included depends on the emulation you are using, as shown in Table 2-1 below.

EMULATION	INCLUDE FILE
TTY	TTY.BI
ANSI	ANSI.BI
VT52	VT52.BI
VT100	VT100.BI
D215	D215.BI

Table 2-1: Emulation Include File Names

These include files allocate a COMMON SHARED variable defined as TermType, using the name of the emulation used. For example, ANSI.BI looks like this:

```
DECLARE SUB AnsiInit ()
DECLARE SUB AnsiPrint (Text$)
DECLARE SUB SetAnsiWindow (WinNum%)

COMMON SHARED /Ansi/ Ansi AS TermType
```

The syntax for all of the emulation routines is identical, and for clarity all of the discussions and code samples that follow will use the ANSI routines as an example. Please note that for every occurrence of the word *Ansi*, you may substitute the name of another emulation.

This first example shows how to use the ANSI emulation to create a window with screen coordinates of (1, 1, 24, 80). Specifying line 24 as the bottom allows you to reserve line 25 for status information. The following example shows how to define this window:

```
DEFINT A-Z
'$INCLUDE: 'TERM.BI'                    'required include files
'$INCLUDE: 'ANSI.BI'

CALL AnsiInit                           'set up defaults for an ANSI window

Ansi.BRow = 24                          'set line 24 as the bottom row
CALL AnsiPrint  (CHR$(12))              'this clears the screen
```

Since the window has already been initialized to 1, 1, 25, 80 by AnsiInit, only the bottom row needs to be changed. From this point on, any text that is displayed using AnsiPrint will be contained within the window and scrolled automatically. The PDQComm emulation routines also handle all of the ANSI escape sequences that are embedded within the text.

Because these emulation routines need to operate independently of BASIC, you must use them to locate the cursor, set display colors, and clear the screen, instead of BASIC's LOCATE, COLOR, and CLS statements. This is accomplished by using the same escape code the emulation routines are designed to act on, or by modifying the TYPE variable as described following.

Locate

To locate the cursor at a specified row and column, set the CurRow and CurCol components of the Ansi TYPE variable. The next time AnsiPrint is called, the cursor will be moved there prior to printing. The following example sets the cursor to 10,40 by assigning the TYPE variable and then printing an empty string. Of course, it is not mandatory to print a null string unless you want the cursor to actually be moved. Simple assigning CurRow and CurCol will force subsequent printing to be performed at the specified location.

```
Ansi.CurRow = 10                    'set row 10
Ansi.CurCol = 40                    'and column 40
CALL AnsiPrint("")                  'move the cursor
```

Note that you may still use BASIC's LOCATE to turn the cursor on and off. Also, you may use LOCATE and PRINT to manually print information outside of the current window. In this case you would do that to print the status information on line 25. Even though BASIC will temporarily relocate the cursor to line 25, PDQComm remembers the correct cursor position within the ANSI window for subsequent calls to AnsiPrint.

Color

To change the current screen colors you will modify the Fore and Back portions of the Ansi TYPE variable. For example, the following code changes the current colors to a black foreground and white background (inverse):

```
Ansi.Fore = 0
Ansi.Back = 7
CALL AnsiPrint("This is printed in inverse video")
```

Cls

All of the emulation routines recognize CHR$(12) as a form feed character, and printing that will clear the current window and position the cursor at the top left corner:

```
CALL AnsiPrint(CHR$(12))
```

Using Multiple Windows

PDQComm allows you to define as many different windows on the screen as you want, up to the limits of available memory. Creating more than one window is accomplished by using the SetAnsiWindow routine. Presently, the code in these routines is commented out to avoid wasting space if you need only one window. If you intend to use multiple windows, please examine the comments contained in the ANSIDISP.BAS program file.

To create multiple windows you must first activate the remarked-out code in ANSIDISP.BAS, and also set the MaxWindows% constant to the maximum number of windows you intend to use. Like the SetCom routine that is used to identify which open communications port will be accessed, the emulation routines use a similar method. Simply call SetAnsiWindow indicating which of the windows you plan to address:

```
CALL SetAnsiWindow(WinNumber%)
```

The following routine shows how to create four ANSI windows occupying the four screen quadrants:

```
DEFINT A-Z

'$INCLUDE: 'TERM.BI'                  'required include files
'$INCLUDE: 'ANSI.BI'

FOR I = 1 TO 4
  CALL SetAnsiWindow(I)               'set the active window
  CALL AnsiInit(I)                    'initialize to default values
NEXT

CALL SetAnsiWindow(1)                 'address the first window
Ansi.TRow = 1                         'assign coordinates of (1,1,12,39)
Ansi.BRow = 12
Ansi.LCol = 1
Ansi.RCol = 39
CALL AnsiPrint(CHR$(12))              'clear the screen window

CALL SetAnsiWindow(2)                 'as above
Ansi.TRow = 1                         'using coordinates of (1,41,12,80)
Ansi.BRow = 12
Ansi.LCol = 41
Ansi.RCol= 80
CALL AnsiPrint(CHR$(12))              'clear the screen

CALL SetAnsiWindow(3)                 'as above
Ansi.TRow = 14                        'using coordinates of (14,1,25,39)
Ansi.BRow = 25
Ansi.LCol = 1
```

```
Ansi.RCol = 39
CALL AnsiPrint(CHR$(12))

CALL SetAnsiWindow(4)                'as above
Ansi.TRow = 14
Ansi.BRow = 25
Ansi.LCol = 41
Ansi.RCol = 80
CALL AnsiPrint(CHR$(12))
```

The program DEMOWIN.BAS demonstrates using multiple windows and multiple emulations at the same time. Again, all of the other emulation routines operate in exactly the same way as the ANSI routines described here, and the only difference is the routine and file names.

TRANSFERRING FILES WITH PDQCOMM

PDQComm currently supports ASCII file transfers for text, and XMODEM file transfers for binary data. The routines that perform these files transfers are in the ASCIIXFR.BAS and XMODEM.BAS files respectively. All of these routines have been designed as functions, and return a completion code to indicate their success or failure as shown in Table 2-2 below.

Code		Meaning
0	-	File sent successfully
1	-	DOS file error
2	-	Timeout error
3	-	Too many errors
4	-	Receiver cancelled
5	-	Transmitter cancelled

Table 2-2: File Transfer Return Codes

The file transfer routines call a subprogram named XStat which opens a window on the screen and reports the status of the file transfer. XStat is located in the file XSTAT.BAS. If you prefer to create your own reporting method or have no reporting, simply replace XStat with your own routine.

When XStat is first called at the beginning of a transfer, it saves the underlying screen contents and then draws a window on the screen. The first string passed to XStat is printed on the first two lines of the window, and it is divided into two parts separated by a vertical bar (|). For instance:

```
CALL XStat("This is line 1 | And this is line 2")
```

All subsequent calls to XStat print text on the bottom line of the window. Calling XStat with a null string ("") tells it that the transfer is complete. XStat will then close the window and restore the underlying screen. For more information on file transfers, please see the reference section for the appropriate routines.

Chapter 3:
Functions and Subroutines

AdjustRecBuffer (Subroutine)

- **Purpose**

AdjustRecBuffer changes the size of the receive buffer, as well as the near-empty and near-full points used for handshaking.

- **Syntax**

```
CALL AdjustRecBuffer(BuffSize%, NEmpty%, NFull%)
```

- **Where**

BuffSize% is the new size of the receive buffer. If this value is 0, the buffer is not adjusted but the NearEmpty and NearFull points are changed. NEmpty% is the near-empty value. If handshaking is used, this is the point at which the receiver tells the transmitter to resume sending after having previously told it to stop. NFull% is the near-full value. If handshaking is being used, this is the point at which the receiver tells the transmitter to stop sending.

- **Comments**

This routine is used to change the default buffer size given when you have used OpenComX. It is also used to change the handshaking points. The handshaking points specify when the sender is to stop and resume transmitting. For example, if the buffer size is 1,024 bytes, you could tell AdjustRecBuffer that it is to send a Stop signal when 900 characters have been received. This affords some measure of safety, instead of allowing the buffer to fill completely. Likewise, you can specify at what point a Resume signal is sent. Rather than wait for the buffer to become empty, you could tell AdjustRecBuffer to have the sender resume when 100 bytes remain.

Note that if you change the buffer size, the current contents of the receive buffer will be lost. You can specify a receive buffer length between 128 and 32,767 bytes. 1,024 bytes is adequate for most purposes. The only error checking done on the near empty and full values is to ensure they are greater than 0 and less than the receive buffer size. If you set NEmpty% greater than NFull%, your program will not work correctly. These values matter only when handshaking is used.

PDQComm defaults to a near-empty value of 32 and a near-full value of BuffSize% - 64. While this is more than adequate for direct and modem connections, it is not sufficient for delayed connections such as satellite communications. When the receiver sends an XOFF in this instance, there is a slight delay before the transmitter receives it. Thus, the receiver buffer could overflow. For communications where such delays are possible, we

recommend a near-empty point of 1/3 and a near-full point of 2/3 of the buffer size.

AdjustRecBuffer will set the BASIC ERR function to 7, "Out of memory" if there isn't enough DOS memory for the buffer. It will set ERR to 5, "Illegal function call" if you provide incorrect parameters.

ASCIIReceive (Function)

Purpose

ASCIIReceive receives a text file over an open serial port using no protocol.

Syntax

```
RetCode = ASCIIReceive%(FileName$)
```

Where

FileName$ is the name of the file you want to receive, and RetCode is assigned a return code as follows:

ASCIIReceive Return Codes

0 - File sent successfully
1 - DOS file error
2 - Timeout error
3 - Too many errors
4 - Receiver cancelled
5 - Transmitter cancelled

Comments

Because ASCIIReceive has been designed as a function, it must be declared before it may be used.

ASCIIReceive uses XON/XOFF handshaking and closes the file and returns when a CHR$(26) EOF (end of file) character is received. Therefore, it is not appropriate for receiving binary data that may contain embedded CHR$(26) characters. Also note that ASCIIReceive overwrites filename$ if it already exists.

ASCIIReceive is contained in the ASCIIXFR.BAS file. Please see the section entitled "Using PDQComm" for more information on file transfers.

ASCIISend (Function)

- **Purpose**

ASCIISend sends a text file over an open serial port using no protocol.

- **Syntax**

```
RetCode = ASCIISend%(FileName$)
```

- **Where**

FileName$ is the name of the file you want to send, and RetCode is assigned a return code as follows:

ASCIISend Return Codes

0 - File sent successfully
1 - DOS file error
2 - Timeout error
3 - Too many errors
4 - Receiver cancelled
5 - Transmitter cancelled

- **Comments**

Because ASCIISend has been designed as a function, it must be declared before it may be used.

ASCIISend uses XON/XOFF handshaking, and sends a CHR$(26) EOF (end of file) character to signal the end of transmission. Therefore, it is not appropriate for transmitting binary data that may contain embedded CHR$(26) characters.

ASCIISend is contained in the ASCIIXFR.BAS file. Please see the section entitled "Using PDQComm" for more information on file transfers.

BIOSInkey (Function)

- **Purpose**

BIOSInkey is similar to BASIC's native INKEY$ function, except it returns an integer result.

- **Syntax**

```
KeyHit = BIOSInkey%
```

- **Where**

KeyHit receives 0 if no key is pending in the keyboard buffer, a positive number that represents a normal key's ASCII code, or a negative value

that corresponds to an extended key code. For example, the "A" key has a value of 65, and the F1 key is returned as -59.

- **Comments**

Because BIOSInkey has been designed as a function, it must be declared before it may be used.

In general, integer functions such as BIOSInkey require less setup and processing by the BASIC compiler than do string functions. Also, integer comparisons are much faster and require less code than string comparisons.

BIOSPrint (Subroutine)

- **Purpose**

BIOSPrint is similar to PDQPrint except it uses the BIOS to print instead of directly writing to screen memory.

- **Syntax**

```
CALL BIOSPrint(Text$, Row%, Col%, Colr%)
```

- **Where**

Text$ is the text to be printed, Row% and Col% are the row and column where the text is to be printed, and Colr% is the combined foreground and background color.

- **Comments**

BIOSPrint is used to avoid "bleed-through" problems when a program is being used with a multi-tasker such as Windows or DesqView. Unless the program is running on a 80386 or later, direct screen writes from a background operation will appear on the currently running foreground program. Using BIOSPrint avoids that problem, because the multi-tasker can monitor all BIOS calls, and handle the characters correctly. BIOSPrint is used by the PDQComm terminal emulator modules.

Note that this routine is much slower than PDQPrint. Please see PDQPrint and OneColor for additional information.

Carrier (Function)

- **Purpose**

The Carrier function returns the status of the CD hardware line.

- **Syntax**

```
CDLine = Carrier%
```

- **Where**

CDLine is assigned to -1 (True) if the CD line is currently active, or 0 (False) if it is not.

- **Comments**

Because Carrier has been designed as a function, it must be declared before it may be used.

On modems that reflect the actual state of the carrier, Carrier will tell you if the remote system has disconnected. The following example shows a typical use for this function.

```
IF Carrier% THEN
    PRINT "Carrier Detected"
ELSE
    PRINT "No Carrier Detected"
END IF
```

Carrier sets the BASIC ERR function to error 52, "Bad file number" if the communications port is not open.

Checksum (Function)

- **Purpose**

Checksum returns the checksum value for a string.

- **Syntax**

```
Check$ = Checksum$(Work$)
```

- **Where**

Check$ receives the checksum of Work$.

- **Comments**

Because Checksum has been designed as a function, it must be declared before it may be used.

Checksums are used to detect if data has been corrupted. In the context of a communications package, a checksum can be used to ensure that the data that was sent is the same as that received. There are many ways to calculate a checksum, and this routine is among the simplest. (But also see the CRC16 function elsewhere in this section, which uses a much more

sophisticated method.) Communications programs that use a checksum simply calculate the value before transmitting, send the data, and then send the checksum. The receiving program does likewise, and compares the received checksum with the value it calculates locally. If the two match, then the data is assumed to have arrived intact.

In this case, the ASCII values of all the characters being sent are added together, with any excess beyond 256 discarded. A BASIC equivalent of the PDQComm Checksum algorithm is shown below.

```
Check = 0
FOR X = 1 TO LEN(Work$)
     Check = Check + ASC(MID$(Work$, X, 1))
NEXT
Check = Check AND 255
```

Checksum is designed to return a string rather than an integer value, to simplify passing it as a parameter to ComPrint.

CloseCom (Subroutine)

- **Purpose**

CloseCom closes the COM port and restores the original interrupt vector. It also deallocates the memory that was used by the receive buffer.

- **Syntax**

```
CALL CloseCom
```

- **Comments**

You should call CloseCom before ending your program.

ComEof (Function)

- **Purpose**

ComEof indicates if there are any characters waiting to be read from the receive buffer.

- **Syntax**

```
IF ComEof% THEN
  ' no characters are waiting to be read.
END IF
```

- **Comments**

Because ComEof has been designed as a function, it must be declared before it may be used.

ComEof is identical to the regular BASIC EOF() function when using the COM port. If there are no characters waiting to be read in the receive buffer, ComEof will return -1 (True). If the receive buffer does contain characters, then ComEof returns 0 (False).

ComEof sets the BASIC ERR function to error 52, "Bad file number" if the communications port is not open.

The example below shows ComEof in a typical context.

```
IF NOT(ComEof%) THEN
     Text$ = ComInput$(ComLoc%)
     PRINT Text$;
END IF
```

ComInput$ (Function)

- **Purpose**

ComInput$ reads a specified number of characters from the receive buffer.

- **Syntax**

```
Text$ = ComInput$(NumChars%)
```

- **Where**

Text$ receives the next NumChars% characters that are available to be read.

- **Comments**

Because ComInput$ has been designed as a function, it must be declared before it may be used.

This function reads characters from the receive buffer, and returns them as a single string. You tell ComInput$ how many characters you want to read, and it returns either that number of characters, or the number of characters that are currently in the buffer, whichever is less. If you call this function when the buffer is completely empty, it will return a null string and set ERR to error 62, "Input past end". In general, however, you will use ComInput$ in conjunction with the ComLoc function, to simply read however many characters there are available. This is shown in the example below.

```
Text$ = ComInput$(ComLoc%)
```

ComInput$ sets the BASIC ERR function to error 52, "Bad file number" if the communications port is not open.

ComLineInput (Subroutine)

- **Purpose**

ComLineInput reads an entire line of text from the receive buffer in a single operation.

- **Syntax**

```
TimeOut% = 5                          'use a timeout period of
                                      ' 5 seconds
CALL ComLineInput(Text$, TimeOut%)'input the line of text received
```

- **Where**

Text$ receives the next full line of text from the receive buffer, but only if it arrives within the next five seconds.

- **Comments**

ComLineInput serves the same purpose as BASIC's regular LINE INPUT # statement when using a communication port. It works by examining all of the characters in the receive buffer (those that have been received but not yet read by your program), until it finds a carriage return. It then checks if the next character in the buffer is a line feed, and discards that if it is. If no carriage return is found, ComLineInput will wait for one, up to the number of seconds specified byTimeOut%.

The TimeOut parameter specifies the number of seconds ComLineInput will wait for a carriage return before returning to your program. For example, if the sending program does not issue a carriage return within, say, 5 seconds, then ComLineInput simply returns with however many characters have been received thus far. To disable the time-out feature entirely, you should set TimeOut% to 0. However, your program will never regain control if the sender fails to transmit a CHR$(13) carriage return. Please notice that text lines are limited to 256 characters.

ComLineInput$ sets the BASIC ERR function to error 52, "Bad file number" if the communications port is not open. If ComLineInput does not find a carriage return within the first 256 characters, it returns the first 256 characters and sets ERR to error 62, "Input past end". If a carriage return is not received within the specified timeout period, ComLineInput sets ERR to error 24, "Device timeout".

ComLoc (Function)

- **Purpose**

ComLoc reports the number of characters currently waiting to be read in the receive buffer.

- **Syntax**

```
NumChars = ComLoc%
```

- **Comments**

Because ComLoc has been designed as a function, it must be declared before it may be used.

ComLoc serves the same purpose as BASIC's LOC() function when using communications, and it is generally used in conjunction with the ComInput$ function as shown below.

```
Text$ = ComInput$(ComLoc%)
```

This example uses ComLoc% to retrieve all of the characters that are currently in the receive buffer.

ComLoc sets the BASIC ERR function to error 52, "Bad file number" if the communications port is not open.

ComPrint (Subroutine)

- **Purpose**

ComPrint replaces the QuickBASIC function PRINT # statement when printing to the communications port.

- **Syntax**

```
CALL ComPrint(Work$)
```

- **Where**

Work$ is sent through the communications port.

- **Comments**

ComPrint sets the BASIC ERR function to error 52, "Bad file number" if the communications port is not open. If there is a timeout due to handshaking, ComPrint will set ERR to error 24, "Device timeout".

If a timeout does occur, ComPrint remembers where in the string it left off. Thus, you would call it again later using the exact same string to

finish transmitting it. Please note that if you are using handshaking and receive a timeout error, you will need to receive a resume from the remote system or disable handshaking with the SetHandshaking routine to allow printing to resume.

CRC16 (Function)

- **Purpose**

CRC16 (Cyclical Redundancy Check, 16 bit method) returns the CRC-16 checksum calculation of a string.

- **Syntax**

```
Check$ = CRC16$(Work$)
```

- **Where**

Check$ receives the CRC-16 calculation of Work$.

- **Comments**

Because CRC16 has been designed as a function, it must be declared before it may be used.

CRC16 is similar in concept to the Checksum routine described elsewhere in this section, except is uses a much more sophisticated technique. Please see the comments that accompany the Checksum function for a brief discussion of how checksums are used.

CRC16 is designed to return a string, to simplify passing it as a parameter to ComPrint.

CRC16 is shown in context in the XMODEM.BAS file that is used in the PDQTERM.BAS example program.

DTR (Subroutine)

- **Purpose**

The DTR routine toggles the state of the hardware DTR line.

- **Syntax**

```
State% = 0
CALL DTR(State%)
```

- **Where**

If the State% argument to DTR is True (-1), then the DTR line is set to active (low). If State% is False, then the DTR line is set to inactive (high).

- **Comments**

Using the DTR routine to switch the DTR line false is often used to force a modem to hang up the line.

DTR sets the BASIC ERR function to error 52, "Bad file number" if the communications port is not open.

FlushBuffer (Subroutine)

- **Purpose**

The FlushBuffer routine resets the receive buffer, flushing all of the characters that it currently contains.

- **Syntax**

```
IF OverRun% THEN CALL FlushBuffer
```

- **Where**

The receive buffer is purged of all data if a "buffer overrun" condition has occurred.

- **Comments**

You must call this routine if your program receives a "Buffer overrun" error, as detected by the PDQComm Overrun function. When this happens the characters in the receive buffer are invalid, so you must flush the buffer and begin again. Note that you will not need to use FlushBuffer or Overrun if you are using either of the two handshaking methods provided in PDQComm.

GetComPorts (Subroutine)

- **Purpose**

GetComPorts examines low memory and returns the addresses of all of the installed communications ports.

- **Syntax**

```
CALL GetComPorts(Port1%, Port2%, Port3%, Port4%)
```

- **Where**

Port1%, Port2%, Port3%, and Port4% are returned holding the respective port addresses.

- **Comments**

GetComPorts may be used to determine which communications ports are actually present in a PC. If a Port parameter is returned as non-zero, then that port does in fact exist.

Notice that some older PC's do not report Com3 or Com4, even when they are present. In that case, and if you are sure they really do exist, the correct port addresses are &H3E8 and &H2E8 for Com3 and Com4 respectively.

GetLineStatus (Subroutine)

- **Purpose**

GetLineStatus allows you to retrieve the status of all port lines in one operation.

- **Syntax**

CALL GetLineStatus(PortStat)

- **Where**

PortStat is a TYPE variable that has been dimensioned using the LStatType structure defined in COMMDECL.BAS. It is constructed as follows:

```
TYPE LStatType
                            CTS AS INTEGER
                            DSR AS INTEGER
                            RI  AS INTEGER
                            DCD AS INTEGER
END TYPE
```

Here, CTS is the Clear To Send line, DSR is the DataSet Ready line, RI is the Ring Line, and DCD is the Carrier Detect line.

- **Comments**

This routine is useful for determining the status of the remote device. For instance, if you want to open a port to a serial printer but want to make sure the printer is turned on and on-line, you would use code similar to this:

```
DIM PrinterStat AS LStatType            'create the TYPE variable

CALL OpenCom("Com1:2400,N,8,1,XON")     'open Com1
Start& = PDQTimer                       'start timeout counter
DO
```

```
  CALL GetLineStatus(PrinterStat)          'get the status of the port
  IF PDQTimer& - Start& > 90 THEN          'after 5 seconds, give up
     PRINT "Printer not on-line"
     END
  END IF
LOOP UNTIL PrinterStat.CTS                 'continue until CTS is active
```

You may have to check PrinterStat.DSR or both PrinterSta.DSR and PrinterStat.CTS together, depending on the printer. If you merely want to check the status of the DCD line, you should use the Carrier% function.

GetLineStatus will set the BASIC ERR function to 52, "Bad file number" if the communications port is not opened.

GetPortConfig (Subroutine)

- **Purpose**

GetPortConfig retrieves the parameters for a currently open communications port, and returns them in a TYPE variable.

- **Syntax**

```
CALL GetPortConfig(Port%, PortConfig)
```

- **Where**

Port% is the communications port (1, 2, 3, or 4), and PortConfig is a special TYPE variable that holds the various configuration information.

- **Comments**

GetPortConfig relies on the ModemType TYPE variable, which is defined in the COMMDECL.BAS include file. ModemType is designed as follows:

```
TYPE ModemType
     Baud AS LONG
     DBits AS INTEGER
     Parity AS STRING * 1
     SBits AS INTEGER
END TYPE
```

Baud is the baud rate of the port, DBits are the number of data bits, Parity is either "E", "O", or "N" for even, odd, or none respectively, and SBits is the number of stop bits. This routine is particularly useful when you need to determine the current parameters for an already-open communications port.

GetPortConfig sets the BASIC ERR function to error 52, "Bad file number" if the communications port is not valid. This is a useful way to see if the port is present, as shown in the example below.

```
DIM PortConfig AS ModemType
Port% = 1
CALL GetPortConfig(Port%, PortConfig)
IF ERR = 52 THEN PRINT "Port not there"
```

OneColor (Function)

- **Purpose**

OneColor accepts separate foreground and background color values, and returns them combined in a single byte for use with the QuickPak Professional video routines.

- **Syntax**

```
Colr% = OneColor%(FG%, BG%)
```

- **Where**

FG% and BG% are the intended foreground and background colors, and Colr% receives the combined value.

- **Comments**

Because OneColor is implemented as a function, it must be declared before it may be used.

All of the PDQComm video routines expect a single value to specify both the foreground and background colors. The colors are in fact stored this way by the PC's hardware, and providing them in this format allows the video routines to operate that much faster. Further, this saves variable memory in your programs by eliminating an extra parameter.

The PC's hardware uses a convoluted method to combine the foreground and background components of a color, and OneColor will save you that much additional code and effort.

The formula used by OneColor is:

```
Colr = (FG AND 16) * 8 + ((BG AND 7) * 16) + (FG AND 15)
```

OpenCom (Subroutine)

- **Purpose**

OpenCom is used to open a specified COM port, and it is equivalent to BASIC's OPEN "Com" statement.

- **Syntax**

```
CALL OpenCom("comN:BBBBB,P,D,S,[rbFFFF],[HHH]")
```

- **Where**

N is the communications port to open; valid port numbers are 1 through 4 inclusive.

BBBBB is the baud rate; Valid numbers are 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, and 115200.

P is the parity; valid characters are "N", "E", and "O" for None, Even, and Odd respectively.

D is the number of data bits; valid numbers are 7 and 8.

S is the number of stop bits; valid numbers are 1 and 2.

FFFF is the receive buffer size; this value may range from 128 to 32767 bytes, however you may omit this parameter and accept the default buffer size of 128 bytes.

HHH specifies the handshaking method; valid strings are "XON", "RTS", and "NON". Using "XON" causes PDQComm to use XON/XOFF handshaking, "RTS" means RTS/CTS handshaking is to be used, and "NON"specifies no handshaking. This parameter is also optional, and omitting it will default to no handshaking.

- **Comments**

OpenCom expects a string containing all of the parameters as shown above, except for the last two which are optional. Note that capitalization is ignored. Also, note that you must include the port number, parity, data and stop bit parameters, and they must be in the order shown below. (This is the same requirement as with BASIC's OPEN.) However, the receive buffer and handshaking parameters are optional, and may be listed in either order. If omitted, the buffer size defaults to 128 bytes, and the handshaking defaults to "NON".

OpenCom sets the BASIC ERR function to error 52, "Bad file number" if the communications port is already open. It sets ERR to error 5, "Illegal function call" if you provide incorrect parameters. If you attempt to open a non-existent port, ERR will be set to error 54, "Bad file mode".

OpenCom will also set ERR to error 7, "Out of memory" if there isn't enough DOS memory for the receive buffer.

The following complete example shows OpenCom in action:

```
Param$ = "COM2:19200,N,8,1,RB1024,RTS"
CALL OpenCom(Param$)
IF ERR = 54 THEN
     PRINT "COM port not available"
     END
END IF
```

This opens port COM2 at 19.2K baud with no parity, 8 data bits, 1 stop bit, a receive buffer of 1K, and RTS/CTS handshaking.

OpenComX (Subroutine)

- **Purpose**

OpenComX allows you to open ports that have non-standard addresses or use non-standard IRQ (Interrupt Request) lines. It is also a small-code way to open a standard port, because it avoids the string parsing overhead required in OpenCom.

- **Syntax**

```
CALL OpenComX(Address%, IRQ%)
```

- **Where**

Address% is the base address of the port, and IRQ% is the IRQ level.

- **Comments**

This routine should be used only if you are familiar with the hardware installed in your PC. Using incorrect values will probably cause the PC to lock up. OpenComX currently supports using IRQs 1 to 7. Please understand that this does *not* mean you may use any IRQ value between 1 and 7. IRQs are usually assigned as follows:

IRQ 0 - System Timer
IRQ 1 - Keyboard
IRQ 2 - Redirected IRQ 9 on IBM AT and AT compatables
IRQ 3 - Communications port 2
IRQ 4 - Communications port 1
IRQ 5 - Hard disk on IBM XT
IRQ 6 - Floppy disk
IRQ 7 - Reserved for printer

If you call OpenComX with an IRQ in use by another device, you most surely will have to cold-boot the PC.

OpenComX makes some assumptions in order to reduce the code size. These assumptions are as follows:

- The Baud rate is maintained at the current setting

- The receive buffer size is set to 512 bytes
- Handshaking is not being used

If you need to change any of these default values, PDQComm offers the following routines:

- To change the baud rate, use SetCom first.
- To change the buffer size, use AdjustRecBuffer.
- To change handshaking, use SetHandshaking.

OpenComX makes these assumptions to minimize the amount of code that is added to your program. For example, the following program instructs the modem to take the phone off hook.

```
DEFINT A-Z

 DECLARE SUB OpenComX(Address, IRQ)
 DECLARE SUB ComPrint(Text$)

 CALL OpenComX(&H3F8, &HC)              'open COM1
 CALL ComPrint("ATH1" + CHR$(13))       'tell modem to access the line
```

When this program is compiled and linked with P.D.Q., it produces an .EXE file size of 2,592 bytes. By substituting OpenCom for OpenComX, the program size swells to 4,134 bytes.

OpenComX will set the BASIC ERR function to 52, "Bad file number" if the communications port is already open. It sets ERR to 5, "Illegal function call" if you provide it incorrect parameters. It will also set ERR to 54, "Bad file mode" if there is no communications port at the address you specified. Finally, OpenComX will set ERR to 7, "Out of memory" it there is not enough DOS memory for the receive buffer.

OverRun (Function)

Purpose

OverRun checks the receive buffer for an "overrun", or overflow condition.

Syntax

```
IF OverRun% THEN CALL FlushBuffer
```

- **Where**

The receive buffer is purged of all data if a "buffer overrun" condition has occurred.

- **Comments**

Because OverRun has been designed as a function, it must be declared before it may be used.

OverRun returns -1 (True) if an overrun has occurred, or 0 (False) if it has not. An overrun condition simply means that data has continued to arrive at the communications port when the receive buffer was already full, and your program did not remove the characters quickly enough.

Note that you must call FlushBuffer if you receive an overrun error. You will lose all of the data in the buffer if it becomes overrun anyway, because there is no assurance that the characters present are valid. However, the buffer will never overrun if you are using either of the handshaking methods we provide in PDQComm. OverRun sets the BASIC ERR function to error 52, "Bad file number" if the communications port is not open.

ParseComParam (Subroutine)

- **Purpose**

ParseComParam accepts a string in the format used with OpenCom, and unpacks it into a ModemType structure variable.

- **Syntax**

```
CALL ParseComParam(Parm$, Port%, PortConfig, BufferLength%, HandShake$)
```

- **Where**

Parm$ is a setup string such as

```
"COM1:2400,N,8,1,RB1024,NON"
```

and the remaining parameters are returned set to the equivalent values. PortConfig is a special ModemType TYPE variable that is defined in the COMMDECL.BAS include file. (Also see the GetPortConfig routine for a description of the ModemType variable.)

- **Comments**

ParseComParam is useful if you want to change the parameters using the SetCom routine while a port is open. You would call ParseComParam with the same string that was used with OpenCom, and then change what is necessary. You would then call SetCom to alter just those portions in

the TYPE variable you want changed. The example below shows this in context.

```
DEFINT A-Z

DIM PortConfig as ModemType
Parm$ = "Com1:2400,N,8,1,RB1024,NON"
CALL OpenCom(Parm$)
CALL ParseComParam(Parm$,Port,PortConfig, Buflen, Hand$)
PortConfig.Baud = 1200
CALL SetCom(PortConfig)
```

PortConfig now contains the correct parameters from the original Parm$. That is, Port contains the port number, BufLen holds the length of the receive buffer, and Hand$ reflects the handshaking method currently in use ("RTS", "XON", or "NON").

Notice that the only parameters you may change using SetCom are the baud rate, parity, and data and stop bits. These are the values returned in the PortConfig TYPE variable. To change the handshaking method you must use the SetHandshaking routine. In this example, only the baud rate is being changed, from 2400 to 1200.

Pause (Subroutine)

- **Purpose**

Pause will delay a program for a specified number of 18ths of a second.

- **Syntax**

```
CALL Pause(Ticks%)
```

Ticks% is the number of system timer ticks (18th if a second) to delay.

- **Comments**

P.D.Q. does not support QuickBASIC's TIMER function, because that function requires floating point operations. Pause therefore provides a simple way to delay your program, while providing a finer resolution than BASIC's SLEEP command.

PDQExist (Function)

- **Purpose**

PDQExist provides a simple way to determine if a file exists.

- **Syntax**

```
There = PDQExist%(FileName$)
```

- **Where**

FileName$ is either a file name, or a file specification such as "*.BAS", and There receives -1 if the file exists, or zero if it does not.

- **Comments**

Because PDQExist has been designed as a function, it must be declared before it may be used.

A drive and path specification may be optionally used to specify other than the current defaults.

PDQParse (Function)

- **Purpose**

PDQParse lets you extract individual portions from a delimited string.

- **Syntax**

```
ThisItem$ = PDQParse$(Work$)
```

- **Where**

ThisItem$ receives the next delimited portion of Work$.

- **Comments**

Because PDQParse has been designed as a function, it must be declared before it may be used.

PDQParse serves two important purposes. First, it can be used to parse delimited information in a string such as the current DOS PATH. It also provides an easy way to simulate READ and DATA, to reduce the amount of code that is added to your programs.

There are actually three related routines in this group. The first is PDQParse, which is designed as a string function with a single string argument. Each time PDQParse is invoked, it returns the next successive item in the string. PDQRestore may then be used to reset the routine to start at the beginning again—either with the same string or with a new one. The last routine is SetDelimitChar, and it lets you establish any arbitrary character as a delimiter. The short program below shows this in action.

```
Work$ = "One; Two; Three"
     FOR X = 1 TO 3
     PRINT PDQParse$(Work$);
NEXT

Result on the screen: OneTwoThree
```

By default, PDQParse uses a semicolon (;) as a delimiter, since parsing the DOS PATH is a fairly common operation. But you may also change the delimiter to, say, a comma, which is what BASIC uses when reading DATA items. If your data has a comma in it, then you could change the delimiter to some other character, for example a "|" or even a Ctrl-A. The delimiter can also be changed mid-read if necessary. Like QuickBASIC's READ statement, PDQParse strips leading blanks and Tab characters from each item.

PDQPrint (Subroutine)

- **Purpose**

PDQPrint is a "quick print" routine that bypasses DOS and writes directly to screen memory.

- **Syntax**

```
CALL PDQPrint(Work$, Row%, Column%, Colr%)
```

- **Where**

Work$ is the string to be printed, Row% and Column% specify where on the screen to print, and Colr% is the combined foreground and background color to use.

- **Comments**

PDQPrint assumes an 80 column display using text page zero. PDQPrint fully supports the 25, 43, and 50 line modes available with EGA and VGA adapters.

Because PDQPrint accepts row and column parameters, your program should use BASIC's built-in CSRLIN and POS(0) functions if you intend to print at the current cursor location. This is shown below.

```
CALL PDQPrint(Work$, CSRLIN, POS(0), Colr%)
```

The foreground and background colors must be combined into a single value using the following formula:

```
Colr = (FG AND 16) * 8 + ((BG AND 7) * 16) + (FG AND 15)
```

The simplified formula below does not accommodate flashing:

```
Colr = FG + 16 * BG
```

The first time PDQPrint is called, it examines the type of display adapter installed using a BIOS service, and saves that information internally. Thus, subsequent calls to PDQPrint will be extremely fast. This also

makes PDQPrint ideal for use within a TSR program. As long as it is called once before the program becomes resident, it may be used at any time without regard to whether the BIOS is in an "interruptable" state.

Also see BIOSPrint which prints using the BIOS, and OneColor which combines the foreground and background colors into one byte automatically for you.

PDQRestore (Subroutine)

- **Purpose**

PDQRestore is intended to be used with PDQParse, to force that routine to begin reading from the beginning of the string.

- **Syntax**

```
CALL PDQRestore
```

- **Where**

The internal pointer PDQParse uses is reset to the beginning of the string.

- **Comments**

See the PDQParse routine elsewhere in this section.

PDQTimer (Function)

- **Purpose**

PDQTimer returns the number of timer ticks stored in the BIOS data area in low memory.

- **Syntax**

```
NumTicks& = PDQTimer&
```

- **Where**

NumTicks& receives the contents of the four-byte system timer.

- **Comments**

Because PDQTimer has been designed as a function, it must be declared before it may be used.

PDQTimer is provided as a way to measure elapsed time. Please note that when the clock passes midnight, the timer is reset to zero:

```
Start& = PDQTimer&                'start the timer
FOR X = 1 to 10000                'we want to time how long this
     .                            '  takes
     .
```

```
NEXT
Done& = PDQTimer                         'done timing
IF Done& < Start& THEN
  PRINT "The clock passed midnight so I'm totally lost."
ELSE
  PRINT Done& - Start&; "clock ticks have elapsed"
END IF
```

PDQValI and PDQValL (Functions)

- **Purpose**

PDQValI returns an integer that represents the value of a string, and PDQValL returns a long integer.

- **Syntax**

```
Value = PDQValI%(Work$)
```

or

```
Value = PDQValL&(Work$)
```

- **Where**

Work$ is a string containing a number such as "1234", and Value receives its value.

- **Comments**

Because PDQValI and PDQValL have been designed as functions, they must be declared before they may be used.

These function are useful because the BC.EXE compiler generates floating point interrupts whenever VAL() is used, and these integer and long integer functions do not. Because floating point operations are not required, PDQValI and PDQValL are also extremely fast.

RTS (Subroutine)

- **Purpose**

RTS sets the state of the RTS line.

- **Syntax**

```
CALL RTS(State%)
```

- **Where**

State% is either 0 to clear the line, or anything else to set it.

- **Comments**

The RTS subroutine lets you manually control the state of the RTS (Request To Send) line on a serial port. Although PDQComm features automatic control of this line as part of its support for hardware handshaking, there may be situations where you need to control the line manually. For example, if you are receiving data and want to Shell to DOS, you can tell the sender to halt prior to using the BASIC SHELL command. Then, once the person using your program types EXIT and you regain control, you can reenable the RTS line status telling the sender that it's okay to resume.

To halt the flow of data as described here, call RTS with an argument of 0. Then to resume sending again, use -1 (or any other value).

ScanCodes% (Function)

- **Purpose**

ScanCodes is used by the terminal emulators. It is similar to INSTR except it returns the position of the first control character it finds in the string. (Control characters are those with an ASCII values less than 32.)

- **Syntax**

```
Found = ScanCodes%(Start%, Text$)
```

- **Where**

Start% is the position in Text$ to begin looking. This must have a value between 1 and LEN(Text$). Text$ is the text to search.

- **Comments**

Since ScanCodes is a function, it must be declared before you may use it.

ScanCodes is declared in COMMDECL.BAS so if you include that file, you do not need to declare it yourself. This routine is used by the emulation routines to search for escape sequences and control codes, and it is documented here in the interest of completeness.

SendBreak (Subroutine)

- **Purpose**

This routine transmits a break signal. A break signal is a logic 0 that is asserted for a time period longer than one character.

- **Syntax**

```
CALL SendBreak(Ticks%)
```

- **Where**

Ticks% is the number of timer ticks (1/18th second) that the break signal will be sent for.

- **Comments**

Some mainframe and minicomputer systems require that a break be sent to cancel a session. Consult that system's manual for the recommended break duration.

SendBreak will set the BASIC ERR function to 52, "Bad file number" if the communications port is not opened.

SetActivePort (Subroutine)

- **Purpose**

This routine changes the active port, to allow access to any two ports at one time. The active port is the one that commands such as OpenCom and ComPrint work with.

- **Syntax**

```
CALL SetActivePort(PortNum%)
```

- **Where**

PortNum% is either 1 or 2, to specify either the first port that was opened or the second. The 1 or 2 have no relation to the actual port number.

- **Comments**

Using this routine allows you to open two ports at once. Older versions of PDQComm allowed only one port to be open at one time, and therefore did not require a port number argument. For example, BASIC's communications routines need a file number to specify the port, which adds extra code each time they are used. PDQComm avoids that problem and also maintains compatibility with the original version by letting you manually switch the port that is to be addressed.

Due to the hardware design of the ISA (Industry Standard Architecture) bus, you cannot open two ports that use the same IRQ. This means you cannot open ports 1 and 3 at the same time, or ports 2 and 4 together.

PDQComm will not allow you to open two ports on the same IRQ, but it won't stop you from opening a port that shares an IRQ opened by another program.

For instance, if you have a serial mouse connected to COM2, PDQComm will allow you to open COM4. But in that case neither the mouse nor the port will operate properly.

The MCA and EISA buses do allow different boards to share the same IRQ, but PDQComm does not support IRQ sharing at this time. Also, you can buy special boards that have a number of ports and allow you to share IRQs, but PDQComm does not currently support this either.

The following code fragment opens both COM3 and COM2, and simply redirects everything that comes into COM3 out through COM2.

```
DEFINT A-Z

'$INCLUDE: 'COMMDECL.BAS'

CALL OpenCom("Com3: 19200,N,8,1,RB2048,XON")'open first port
CALL SetActivePort (2)                       'set second port active
CALL OpenCom("Com2: 2400, N,8,1,XON")        'open second port
CALL SetActivePort(1)                        'make first port active again

DO
     IF ComLoc THEN                          'see if we have characters
        ComOneIn$ = ComeInput$(ComLoc)       'we do so get them
        CALL SetActivePort(2)                'make the second port active
        CALL ComPrint(ComeOneIn$)            'output the characters
        CALL SetActivePort(1)                'make port 1 active again
     END IF
     IF LEN(INKEY$) THEN EXIT DO             'continue until a keypress
LOOP

CALL CloseCom                                'close the first port
CALL SetActivePort(2)                        'make port 2 active
CALL CloseCom                                'close second port
END
```

SetActivePort will set the BASIC ERR function to 5, "Illegal function call" if you specify a port that has not been opened.

SetCom (Subroutine)

- **Purpose**

SetCom lets you modify the parameters for an already open communications port.

- **Syntax**

```
CALL SetCom(PortConfig)
```

- **Where**

PortConfig is a special ModemType TYPE variable that contains the new port values to use.

- **Comments**

SetCom is useful in many situations, but particularly if you connect to a remote system at a different baud than you had initially set. SetCom thus

lets you change the current parameters to match those of the remote system.

SetCom sets the BASIC ERR function to error 52, "Bad file number" if the communications port is not already open. It sets ERR to error 5, "Ill0egal function call" if you provide incorrect parameters.

The following example shows a typical usage for the SetCom routine.

```
DIM PortConfig as ModemType
PortConfig.Baud = 38400
PortConfig.Parity = "E"
PortConfig.DBits = 7
PortConfig.SBits = 1
CALL SetCom(PortConfig)
```

SetComPrintTO (Subroutine)

- **Purpose**

SetComPrintTO sets the time-out length for ComPrint.

- **Syntax**

```
CALL SetComPrintTO(Seconds%)
```

- **Where**

Seconds% is the new approximate time-out value in seconds, or zero to disable the time-out feature.

- **Comments**

If you are using handshaking and call ComPrint to transmit data, some method is needed to let your program regain control if the receiver asks you to stop sending but never allows you to resume. By default, ComPrint uses a time-out period of five seconds. SetComPrintTO allows you to change that period, or disable the time-out altogether using a value of zero. Be aware that disabling the time-out feature can cause your program to hang if the receiver never allows you to resume transmitting.

ComPrint uses the BIOS system timer count stored in low memory to keep track of how much time has elapsed. Although the BIOS used 18.2 timer ticks for each second, ComPrint uses 16 per second to simplify the math and add less code.

Therefore, the exact time-out period will be *xx* slightly shorter than what you specify.

SetDelimitChar (Subroutine)

- **Purpose**

SetDelimitChar lets you change the default delimiter recognized by the PDQParse function.

- **Syntax**

```
CALL SetDelimitChar(NewChar%)
```

- **Where**

NewChar% is the ASCII value of the new delimiting character.

- **Comments**

See the PDQParse routine elsewhere in this section.

SetFIFO (Subroutine)

- **Purpose**

SetFIFO is used to enable or disable the NS16550A FIFO mode.

- **Syntax**

```
CALL SetFIFO(TriggerLevel%)
```

- **Where**

TriggerLevel% is the point in the FIFO where an interrupt will occur.

- **Comments**

This command allows you to take advantage of the advanced NS16550A UART.

This UART contains a 16-character transmit and receive FIFO (First In, First Out) buffer. This buffer allows the CPU to service the port less frequently, allowing higher speeds and also the ability to run under multi-taskers such as Windows and DesqView without losing characters.

For more information on this UART, please see the tutorial section in this manual.

TriggerLevel% specifies the number of characters contained in the receive FIFO buffer before an interrupt is generated. Valid levels are 1, 4, 8, and 14. If you use a value other than these, SetFIFO will select the next lower legal value. A TriggerLevel% of 0 disables the FIFO operation.

SetHandshaking (Subroutine)

- **Purpose**

SetHandshaking is used to change the handshaking method that will be used by an already-open communications port.

- **Syntax**

```
CALL SetHandshaking("NON")
```

- **Where**

Any handshaking that had been in effect is disabled. (Of course, if "XON" were used in this example, then that handshaking method would be activated instead.)

- **Comments**

SetHandshaking accepts a string argument in the form of "RTS", "XON", or "NON", and uses that to establish the current handshaking method.

SetHandshaking sets the BASIC ERR function to error 52, "Bad file number" if the communications port is not open.

SetMCRExit (Subroutine)

- **Purpose**

SetMCRExit allows you to set the state of the DTR and RTS lines when you close the communications port.

- **Syntax**

```
CALL SetMCRExit(DTRState%, RTSState%)
```

- **Where**

DTRState% and RTSState% are either 0 or -1 to disable or maintain the line states respectively.

- **Comments**

PDQComm is well behaved in that it leaves the DTR and RTS lines in the state they were when the port was opened, unless you have changed them using SetMCRExit. However, this routine is useful when transferring control to another program, so your modem does not hang up because the DTR line drops. The brief example below shows SetMCRExit in context.

```
DTRState% = -1
RTSState% = 0
CALL SetMCRExit(DTRState%, RTSState%)
CALL CloseCom
```

Here, the DTR line is maintained active when the communications port is closed, thereby preventing the modem from hanging up. However, the RTS line is set to inactive.

SetMCRExit sets the BASIC ERR function to error 52, "Bad file number" if the communications port is not open.

SetxxxxWindow (Subroutines)

- **Purpose**

These routines select which window is to be addressed by subsequent calls to the initializing and printing routines.

- **Syntax**

```
CALL SetxxxxWindow(WinNum%)
```

- **Where**

Window number WinNum% is established as being the currently active window.

- **Comments**

There are currently five sets of emulation routines provided with PDQComm. Therefore, the actual name of the window selection routine you call will be either SetANSIWindow, SetTTYWindow, SetD215Window, SetVT52Window, or SetVT100Window.

The code that actually implements the various window setting commands has been commented-out in the source code files, to reduce the size of your programs. Since most applications do not need multiple windows active at one time, the code to implement this feature is usually unnecessary.

Please see the comments in the appropriate *xxxx*DISP.BAS file for instructions on activating these routines. Also see the section entitled "Using PDQComm" for more information on using windowed terminal emulations.

XModemReceive (Function)

- **Purpose**

XModemReceive receives a file over an open serial port using either an X-Modem CRC or X-Modem Checksum protocol.

- **Syntax**

```
RetCode = XModemReceive%(FileName$)
```

- **Where**

FileName$ is the name of the file you want to receive, and RetCode is assigned a return code as follows:

XModemReceive Return Codes

0 - File sent successfully
1 - DOS file error
2 - Timeout error
3 - Too many errors
4 - Receiver cancelled
5 - Transmitter cancelled

- **Comments**

Because XModemReceive has been designed as a function, it must be declared before it may be used.

XModemReceive will use the CRC method by default unless the sender doesn't support it, in which case it will switch automatically to use Checksum.

XModemReceive is contained in the XMODEM.BAS file. Please see the section entitled "Using PDQComm" for more information on file transfers.

XModemSend (Function)

- **Purpose**

XModemSend sends a file over an open serial port using either an X-Modem CRC or X-Modem Checksum protocol.

- **Syntax**

```
RetCode = XModemSend%(FileName$)
```

- **Where**

FileName$ is the name of the file you want to send, and RetCode is assigned a return code as follows:

XModemSend Return Codes

0 - File sent successfully
1 - DOS file error
2 - Timeout error
3 - Too many errors

4 - Receiver cancelled
5 - Transmitter cancelled

- **Comments**

Because XModemSend has been designed as a function, it must be declared before it may be used.

XModemSend will use the CRC method by default unless the receiver doesn't support it, in which case it will switch automatically to use Checksum.

XModemSend is contained in the XMODEM.BAS file. Please see the section entitled "Using PDQComm" for more information on file transfers.

UARTType% (Function)

- **Purpose**

Returns the type of UART at a specified address.

- **Syntax**

```
Port1UART = UARTType%(Address%)
```

- **Where**

Address% is the base address of the port. For example, Address% would be &H3F8 for COM1.

- **Comments**

Because UARTType% is designed as a function, it must be declared before you may use it.

UARTType% is declared in COMMDECL.BAS so if you include that file, you do not need to declare it yourself. This routine returns the following values:

-1 - Unknown
0 - 8250 / 8250B
1 - 8250A / 16450
2 - 16550
3 - 16550A

For more information on these UARTs, please see the tutorial section in this manual. SetFIFO calls this routine to see if you have a 16550A, so you do not need to call it before setting a 16550A in FIFO mode.

XOff (Function)

- **Purpose**

The XOff function returns -1 (True) if the remote system has sent an XOFF and XON/XOFF handshaking is currently enabled.

- **Syntax**

```
IF NOT XOff% THEN CALL ComPrint("Testing")
```

- **Where**

The message is sent using ComPrint, as long as the remote terminal has not sent an XOFF signal.

- **Comments**

Because XOff has been designed as a function, it must be declared before it may be used.

XOff sets the BASIC ERR function to error 52, "Bad file number" if the communications port is not open.

The XOff function allows you to manually check to see if the remote system is ready to receive characters when XON/XOFF handshaking is being used. If you had an earlier version of PDQComm, you do not need to call this routine before calling ComPrint anymore, because this is now handled automatically.

xxxxInit (Subroutines)

- **Purpose**

A call to the appropriate *xxxx*Init routine is required prior to accessing an emulation window.

- **Syntax**

```
CALL xxxxInit
```

- **Where**

All of the parameters that control the specified emulation window are initialized to their default values.

- **Comments**

There are currently five sets of emulation routines provided with PDQComm. Therefore, the actual name of the initializing routine you call will be either ANSIInit, TTYInit, D215Init, VT52Init, or VT100Init.

You must call the appropriate window initializing routine before using any of the supplied emulations. This ensures that the necessary parameters within the window handling code will have been set to the default initial values. Once the window has been properly initialized, you may then further customize the window as necessary.

These routines are contained in the file *xxxx*DISP.BAS, where *xxxx* is the name of the emulation being used. Please see the section entitled "Using PDQComm" for more information on using windowed terminal emulations.

xxxxPrint (Subroutines)

- **Purpose**

The various terminal printing routines are used to display text within a previously defined emulation window.

- **Syntax**

```
CALL xxxxPrint(Text$)
```

- **Where**

Text$ will be displayed within the currently defined emulation window, and any embedded control codes recognized by the selected emulation to set colors, position the cursor, and so forth will be acted upon.

- **Comments**

There are currently five sets of emulation routines provided with PDQComm. Therefore, the actual name of the windowed text printing routine you call will be either ANSIPrint, TTYPrint, D215Print, VT52Print, or VT100Print.

These routines are contained in the file *xxxx*DISP.BAS, where *xxxx* is the name of the emulation being used. Please see the section entitled "Using PDQComm" for more information on using windowed terminal emulations.

Chapter 4: Communications Tutorial

An Introduction To Serial Communications

All IBM PC and compatible computers come with two types of I/O (input/output) ports to allow them to communicate with external devices. These are the serial port and the parallel port. Although these two types of ports are used for similar purposes, they work in different ways.

A parallel port is called a *byte* device, because it sends and receives data eight bits at a time all at once over separate wires. This allows data to be transferred very quickly; however, the cable required can be expensive because of the number of individual wires it must contain. A serial port is a called a *bit* device, because it sends and receives data one bit at a time over one wire. While it takes eight times as long to transfer each byte of information this way, only a few wires are required. In fact, two-way (full duplex) communications is possible with only three separate wires—one to send, one to receive, and a common, or ground wire.

Synchronous And Asynchronous Communications

There are two basic types of serial communications, synchronous and asynchronous. With synchronous communications, the two devices initially synchronize themselves to each other, and then continually send characters to stay in sync. Even when data is not really being sent, the constant flow of bits allows each device to know where the other is at any given time. That is, each character that is sent is either actual data or an *idle* character. Synchronous communications allows faster data transfer rates than asynchronous methods, because additional characters to indicate where each group of bits that comprise one byte begins and ends is not required. Unfortunately, the serial ports on IBM-style personal computers are not synchronous devices, and PDQComm does not support synchronous communications.

Asynchronous communications is the method used on IBM PC and compatible computers. Asynchronous means "no synchronization", and it does not require sending and receiving idle characters. However, the beginning and end of each block of data must be identified by *start* and *stop* bits that indicate when the data transmission is about to begin and when it ends. The requirement to send these additional two bits cause asynchronous communications to be slower than synchronous.

An asynchronous line that is idle is identified with a value of 1, and this is also called a *mark state*. By using this value to indicate that no data is currently being sent, the devices are able to distinguish between an idle state and a disconnected line. When a character is about to be transmitted,

a start bit is sent. A start bit has a value of 0, which is also called a *space state*. Thus, when the line switches from a value of 1 to a value of 0, the receiver is alerted that a data character is about to come down the line.

Communicating By Bits

Once the start bit has been sent, the transmitter sends the actual data bits. There may either be 5, 6, 7, or 8 data bits, depending on the configuration you have selected. Both the receiver and the transmitter must agree on the number of data bits to be used, as well as the timing of the start and stop bits.

Notice that when only 7 data bits are employed, you cannot send ASCII values greater than 127. Likewise, using 5 bits limits the highest possible value to 31. After the data has been transmitted, a stop bit is sent. A stop bit has a value of 1—or a mark state—and it can be detected correctly even if the previous data also had a value of 1. This is accomplished by the stop bit's duration. Stop bits can be 1, 1.5, or 2 bit periods in length.

The Parity Bit

Besides the synchronization provided by the use of start and stop bits, an additional bit called a *parity bit* may optionally be transmitted along with the data. A parity bit affords a small amount of error checking, to help detect data corruption that occurs during transmission. You can choose either even or odd parity, or none at all. When parity is being used, the number of marks (logical 1 bits) are counted, and a single bit is transmitted following the data bits to indicate whether the number of 1 bits just sent is even or odd.

For example, when even parity is chosen, the parity bit is transmitted with a value of 0 if the number of preceding marks is an even number. For the binary value of 0110 0011 the parity bit would be 0. If even parity were in effect and the binary number 1101 0110 were sent, then the parity bit would be 1. Odd parity is just the opposite, and the parity bit is 0 when the number of mark bits in the preceding word is an odd number.

Please understand that parity error checking is very rudimentary. While it will tell you if there is a single bit error in the character, it doesn't show in which bit the error occurred. Worse, if two bits are in error—for example, if two 1 bits are received incorrectly as 0—then the parity would not reflect any error at all. Figure 4-1 shows the letter A (ASCII value 65) as it appears when being transmitted with 7 data bits, 1 stop bit and even parity.

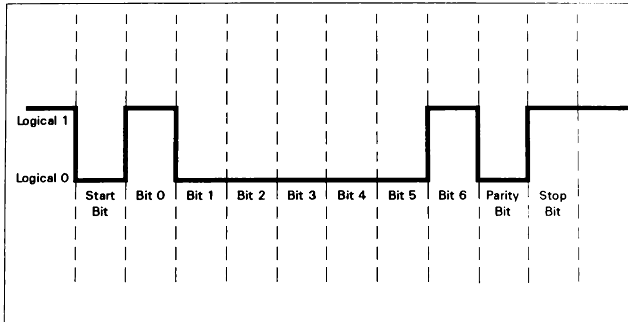


Figure 4-1. Letter "A" (ASCII 65) transmitted with 7 data bits, 1 stop bit, and even parity.

Bidirectional Communications

Now lets continue to some other important terms. The first is *simplex communications*, and when this method is used data can travel in one direction only. This is the simplest type of communications, in which one PC or terminal has only a receiver and the other has only a transmitter. A parallel printer port, for the most part, is a simplex connection because data flows only to the printer. *Half-duplex* is used to describe a system whereby both devices can send and receive, but not at the same time. Some of the modem protocols that will be discussed later in this section are half-duplex.

Finally, *full-duplex* means that both devices can send and receive data at the same time. The serial port on your computer is a full-duplex device, and it uses separate lines for transmitting and receiving data. Other modem protocols that will be discussed later support full-duplex communications over one line.

Baud Versus Bits Per Second

One of the most misused terms in serial communications is *baud*, which many people erroneously believe corresponds to bits per second (BPS).

The baud unit is named after Jean Maurice Emile Baudot, who was an officer in the French Telegraph Service. He is credited with devising the first uniform-length 5-bit code for characters of the alphabet in the late 19th century. What baud really refers to is *modulation rate*. This is the

number of times per second that a line changes state. Isn't this the same as BPS? Well, not exactly. If you connect two serial ports together using direct cables, baud and BPS are in fact the same. That is, if you are running at 38400 BPS, then the line is also changing states 38400 times per second. But when considering modems, this isn't the case.

Because modems need to transfer signals using a telephone line, the maximum baud rate is severely limited. Depending on the type of modulation being employed, modems are limited to either 1200 or 2400 baud. This is a physical restriction of the lines provided by the phone company. You may think you have a 2400 baud modem, but if you do, then your modem that can actually transfer data at 9600 BPS or greater. In fact, 2400 BPS modems are actually 600 baud devices, and the increased data throughput available with 9600 BPS modems is accomplished by the use of sophisticated phase modulation techniques.

RS-232C

The *RS* in RS-232C stands for Recommend Standard, *232* is the identification number for this standard, and *C* is the latest version of the standard. The serial port on most computers is a subset of the RS-232C standard. The full RS-232C standard specifies a 25-pin "D" connector of which only 22 pins are used. Most of these pins are not needed for PC communications, and indeed, many newer computers are equipped with connectors having only 9 pins.

DCE And DTE Devices

Two terms you should be familiar with are DTE and DCE. DTE stands for Data Terminal Equipment, and DCE stands for Data Communications Equipment. RS-232 lines are not bidirectional, and these terms indicate in which direction data is flowing. Your computer is usually a DTE device, while modems are usually a DCE device. Most devices let you choose how they are to be configured.

The RS-232 standard states that DTE devices use a 25-pin male connector, and DCE devices use a 25-pin female connector. You can therefore connect a DTE device to a DCE using a straight pin-for-pin connection. However, to connect two like devices, you must instead use a *null modem* cable. Null modem cables reverse the sense of the transmit and receive lines, and are discussed later in this chapter. Figures 4-2 and 4-3 below illustrate the connections and signal directions for both 9- and 25-pin connectors. Note that in the following paragraphs, DTE is used to mean your computer.

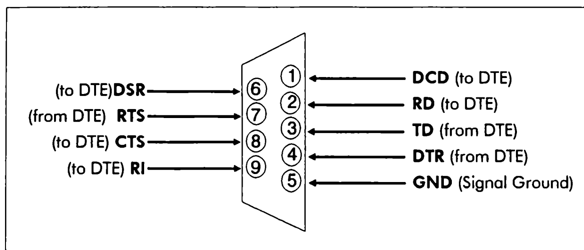


Figure 4-2. 9 Pin RS-232 Connector

Now lets take a closer look at the individual wires within a serial cable. The TD (transmit data) wire is the one through which data from a DTE device is transmitted. This name can be deceiving, because this same wire is used by a DCE device to *receive* its data. The TD line is kept in a mark condition by the DTE device when it is idle. The RD (receive data) wire is the one on which data is received by a DTE device, and the DCE device keeps this line in a mark condition when idle.

RTS stands for Request To Send. The DTE device puts this line in a space condition to tell the remote device that it is ready to send data. The complement of the RTS wire is CTS, which stands for Clear To Send. The DCE device puts this line into a space condition to tell the DTE device that it is ready to receive the data. Together, these two lines make up what is called CTS/RTS handshaking. PDQComm supports this handshaking, as well as software handshaking using special control characters.

DTR stands for Data Terminal Ready. Its function is very similar to the RTS line, and it controls the output from a DTE device. DSR (Data Set Ready) is the companion to DTR in the same way that CTS is to RTS. DSR/DTR handshaking works in the same way as CTS/RTS, but PDQComm does not currently support that because most modems use

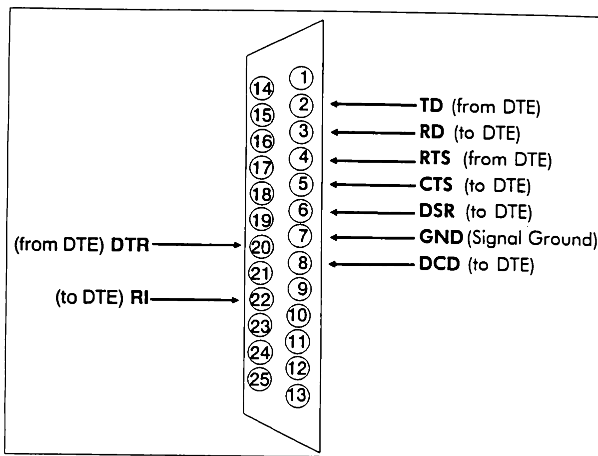


Figure 4-3. 25 Pin RS-232 Connector

DTR to disconnect a call. When the computer drops the DTR line to False, it causes the modem to hang up. This helps to explain QuickBASIC's infamous DTR bug—when QuickBASIC closes a communications port, it always drops the DTR line. PDQComm leaves DTR the way it was when the port was opened, and also allows you to set the state of the line when the port is closed. Note that DTR and DSR are simply an alternate method of hardware handshaking, and it would be pointless to use both CTS/RTS and DTR/DSR at the same time.

CD stands for Carrier Detect. Carrier Detect is used by a modem to signal that it has a connection with another modem, or has detected a carrier. The CD line is useful in determining when the remote device hangs up. A carrier will be explained later in the section on modems.

The final line that we will discuss is RI. RI stands for Ring Indicator, and the modem toggles the state of this line when an incoming call rings your phone. Understand that not all serial ports and modems, or even cables, support the RI line. A much better way to determine when the phone is ringing is to read the string that the modem returns, as shown in the section "Writing a BBS program with PDQComm".

Finally, the RS-232C standard imposes a cable length limit of 50 feet. You can probably ignore this "standard" in most cases, since a cable can be as long as 300 feet at baud rates up to 19200 using modern serial boards. Of course, you must use high quality, well shielded cable when running very long RS-232 lines.

Cables, Null Modems, And Gender Changers

In a perfect world, all serial ports on every computer would be DTE devices with 25-pin male "D" connectors. All modems, printers, and anything else you want to connect to would be DCE devices with 25-pin female connectors.

This would allow you to use a cable in which each pin on one end of the cable is connected to the same pin on the other end. Unfortunately, we don't live in a perfect world. Serial ports use both 9 and 25 pins, most devices can be configured as either DTE or DCE, and—as in the case of serial printers—very well may be configured as a DCE or DTE device. Because of this lack of standardization, special cables called null modem cables and gender changers are often required.

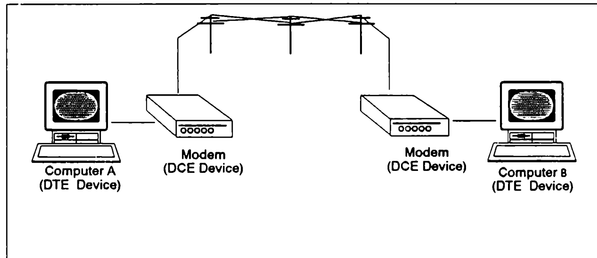


Figure 4-4. Two Computers Connected Via Modems.

Let's look first at null modem cables. Figure 4-4 shows how two computers can communicate with each other using modems. This is the only way to connect them when the computers are physically separated by a long distance. Now consider the situation where the computers are close to one another. In that case you add the expense of the modems, plus you are limited in the maximum baud rate you can use. A null modem cable eliminates the need for a modem entirely, by allowing a direct connection between two DTE devices.

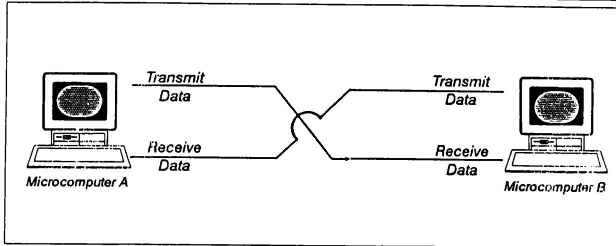


Figure 4-5. Principle of a Null Modem Cable.

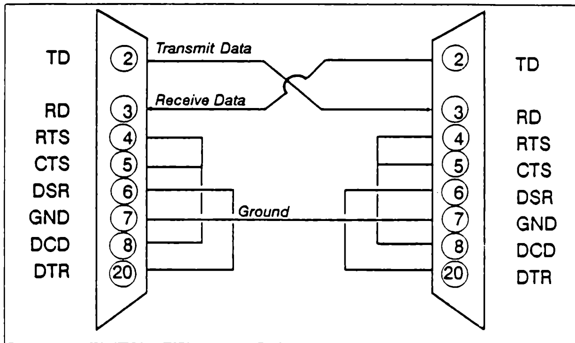


Figure 4-6. A 3-Line Null Modem Cable.

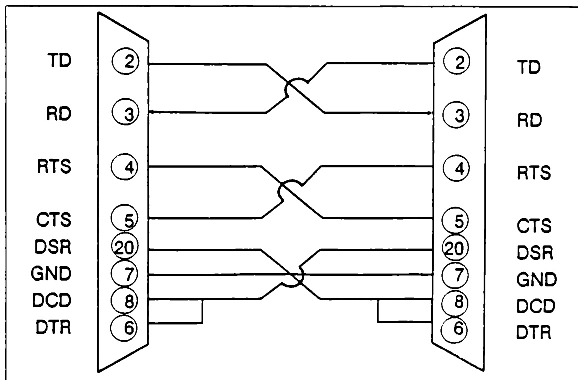


Figure 4-7. A 7-Line Null Modem Cable.

Figure 4-5 illustrates the principle of a null modem cable. By fooling each device into thinking the other is a DCE device, you can communicate at much higher baud rates, albeit at much shorter distances. Figure 4-6 shows a null modem cable that uses only three lines. However, with only three lines you cannot use hardware handshaking such as CTS/RTS or DSR/DTR. Figure 4-7 shows a null modem cable that requires 7 lines and also supports hardware handshaking.

These null modem cables will work in most cases as shown; however, some serial printers may require minor changes.

Table 4-1
A 9 Pin To 25 Pin Adapter

9-Pin Connector	25 Pin Connector
DCD 1	8
RD 2	3
TD 3	2
DTR 4	20
GND 5	7
DSR 6	6
RTS 7	4
CTS 8	5
RI 9	22

Now consider the problem of connecting to computers when one has a 25-pin connector and the other has a 9-pin. In this case you must make or purchase an adapter. Most serial ports that use 9-pin connectors also come with an adapter to convert it to a 25-pin port. If you are handy with a soldering iron you can easily create your own 9-to-25 pin adapter, as shown in Table 4-1. A standard adapter will contain a 9-pin female and a 25-pin male. Of course, you can use whatever gender suits your purposes.

The final problem you may encounter is having two connectors of the same gender that must be connected. Again, you can either purchase these or make your own. (Usually these adapters are grossly overpriced, and besides, you're never to old to learn how to solder.) We recommend using short (1 to 4 inch) pieces of ribbon cable in conjunction with inexpensive snap-on connectors of the same gender. In fact, snap-on connectors eliminate the need for soldering entirely. Using two 25-pin male connectors allows you to connect two 25-pin female connectors, and vice versa.

MODEMS

Modems provide a way to connect computers over long distances through public telephone lines. Back in the days when 64K was considered all the memory a PC would ever need, a 300 BPS modem was state of the art. If you have ever used a 300 BPS modem, you know how frustrating and slow communications can be at that speed. However, modems are now available that can operate at 32 times this speed.

Originally, the main problem facing modem manufacturers who attempted to achieve higher speeds was the inability of phone lines to handle the high frequencies necessary to accommodate higher baud rates. The bandwidth of the phone system limits the modulation to either 1200 or 2400 baud, depending on the modulation method employed. Then Hayes developed a modulation technique that allowed their modems to transmit two bits of information for each change in the line state. This modulation technique is called *quadrature phase shift keying*, and a description of how it works is beyond the scope of this manual. But by using this technique, Hayes was able to create a 1200 BPS modem that operated at 600 baud. Because of this breakthrough Hayes is now a standard, and most modems you see these days claim to be Hayes compatible. Other recent modulation techniques allow 4 or more bits to be encoded into each baud.

As of this writing, 2400 BPS modems can be purchased for about $100. 9600 baud modems are also quickly gaining in popularity, due to the institution of a recent standard. Originally, there was no standard for 9600 BPS modems, so if you bought one you could communicate only with modems made by the same manufacturer. Modems are now available that have a throughput rate of up to 38400 BPS using a variety of schemes including data compression. Table 4-23 following contains a listing of some common modem standards.

Table 4-2
Common Modem Standards

STANDARD	DESCRIPTION	BAUD	BPS
Bell 103	Used by most 300 BPS modems	300	300
V.22	Hayes smartmodem 1200mb	600	1200
V.22bis	2400 BPS extension to V.22	600	2400
V.32	9600 BPS full duplex	2400	9600
V.32bis	14400 BPS extension to V.32	2400	14400
V.42	Error correcting protocol for above	N/A	N/A
V.42bis	Lempel-Ziv compression	N/A	38400

If you plan to purchase a state of the art modem, you should look for a V.32 modem that employs V.42 error correction and V.42b data compression. This guarantees that your modem will be able to connect with most other high speed modems in the future. Currently, these modems cost around $600; however, the price continues to drop as more and more manufacturers enter the market.

All modems are controlled by sending special commands. These commands are commonly referred to as Hayes "AT" commands, because Hayes defined them using an "AT" (attention) prefix string to tell the modem that a subsequent command is coming. Appendix A contains a description of the most common AT commands.

UARTs

The word UART stands for Universal Asynchronous Receiver Transmitter, and it is at the heart of every serial communications port. A UART can be thought of as a type of parallel to serial converter. It accepts parallel data one byte at a time from the CPU, and converts it to the equivalent stream of serial bits that are actually transmitted. The UART also receives serial bits, and converts them into bytes that the CPU can digest.

The IBM PC uses the National Semiconductor 8250 family of UARTs for its serial port. Currently, there are three generations in this family, with each offering improved performance over its predecessor. We will briefly discuss each member of this family.

The INS8250-B UART

The INS8260-B was used in the original IBM PC, and continues to be used in most serial ports today. The basic features of this UART include:

- Utilizes interrupt driven transmit, receive, line status, and data set functions.
- Includes Modem control functions (CTS, RTS, DSR, DTR, RI, and DCD).
- Line break generation and detection.

Table 4-3 following provides a summary of the registers available for controlling this UART.

To access a particular register, start with the base address of the serial port, and then add the register offset found in the table. Note that some of these registers share the same address, specifically offsets 0 and 1.

well as in high speed modems. Notice that the original part contained a bug which rendered the FIFO Buffers useless. The "A" version of this UART fixes the bug and allows correct FIFO operation.

SUMMARY

This section provided a brief overview of serial communications in general, and also discussed some of the specific hardware details within a serial port. In particular, you learned how characters are translated into the bits that are actually transmitted, and how special control wires are used to implement hardware handshaking. Finally, a comparison of various types of UARTS found in IBM PC compatible computers was given. For a more comprehensive description of UART's, we highly recommend the Nation Semiconductor data book entitled "Advanced Peripherals - Data Communications, Local Area Networks, UARTs".

APPENDIX - A

STANDARD HAYES "AT" COMMANDS

All Hayes-compatible modems recognize a standard set of commands, to control various aspects of the modem's operation. Most of these commands begin with the letters "AT", which stands for Attention. For example, the ATDT command tells the modem to access the phone line and dial a number.

The following list of commands is not complete, but it does contain those that are most commonly used. You should consult your modem manual for information on the more advanced commands. Note that all commands must be followed by a CHR$(13) carriage return, except "+++" and "A/". Also note that many modems require these commands to be sent in upper case.

+++ ***On line escape sequence***. Using this sequence returns the modem to command mode from on-line mode. On-line mode is when you are connected to a remote system. In order for your modem to recognize the escape sequence, you must not send any characters to the modem for at least 1 second before and after the escape sequence. This is generally used to get the modem's attention so you can send an "ATH" command to force it to hang up the line. The modem responds with OK when it goes into command mode. At that point you may issue any of the AT commands.

ATO This returns the modem to on-line mode after the "+++" escape sequence has put it into command mode. Note that the "O" is an upper case letter, and not a zero.

A/ ***Execute last command sent.*** A/ does not need a terminating carriage return.

ATA ***Answer phone.*** This command tells the modem to pick up the phone and attempt to connect with a remote modem.

ATDx ***Dial a telephone number.*** The x is replaced with P for pulse dialing or T for touch tone dialing. For example, to call the Crescent Software Support BBS, you would issue the following string to the modem:

```
"ATDT1-203-426-5958" + CHR$(13)
```

If you have rotary (pulse) dialing, you would replace the T with a P.

If your phone is on a PBX system and requires you to dial 9 and wait for a dial tone, you can add a comma between the 9 and the rest of the number as shown below:

"ATDT9,1-203-426-5958" + CHR$(13)

The comma specifies a 2 second pause. For longer pauses simply use multiple commas. You may also use the S-8 register to change the delay time for each comma received. (See the section that describes the S-registers below.)

ATEn ***Echo all commands to the local display.*** The n is replaced by either 0 or 1. ATE0 disables character echo in command mode, and you will not see subsequent AT commands on your PC as you type. ATE1 enables command echo to the screen, which is the default for most modems. Typing ATE alone is the same as ATE0.

ATHn ***Hook switch control.*** The n value is either 0 or 1. ATH0 causes the modem to hang up (go on-hook). ATH1 causes the modem to pick up the line (go off-hook). ATH1 is not the same as ATA because the modem will not try to connect when ATH1 is used. You could use ATH1 if you want your BBS to return a busy signal to callers. Using ATH alone is equivalent to ATH0.

ATMn ***Speaker Control.*** The n values range from 0 through 2. ATM0 keeps the speaker turned off at all times, ATM1 keeps the speaker on until a carrier is detected, and ATM2 keeps the speaker on at all times. The default for most modems is ATM1.

ATQn ***Result code control.*** The n value is either 0 or 1. ATQ0 tells the modem to echo the result of each AT command to the local display, and ATQ1 disables echo. In most cases you will not want to disable the result codes because they can be used by your programs to determine the success or failure of an operation. The default for most modems is ATQ0.

ATSr? ***Read S-Register.*** S-registers control many modem functions and are described below. The r is replaced with the register number you want to read. For example, to read S-register 0 you would send the command string "ATS0?". The modem then returns the current setting to the console for display.

ATSr=n ***Set S-Register.*** This command will set the specified S-register r to the value you assign as n. For example, sending "ATS8=1" tells the modem to pause only one second for each comma received instead of two.

ATVn ***Select result code format.*** The n value is either 0 or 1. ATV0 selects the short form of result codes, which display either one or two digits. ATV1 specifies the long form of result code. Long codes return a text string that describes the result in words. Table A-1 shows some common result codes and their meanings. The default for most modems is ATV1.

ATXn ***Extended result code selection.*** We have found this command to mean different things with different modems. The n value tells the modem how many different result codes you want. For example, ATX0 restricts the range to 300 BPS-compatible result codes, which is only CONNECT. Most modems use ATX4 to enable all result codes, but you should check your modem manual to be certain.

ATZ ***Modem reset command.*** This command restores the modem to the power-on default configuration set in ROM or non-volatile RAM. Depending on your modem brand and model, you may be able to store selected modem default settings that are retained even when the power is turned off.

TABLE A-1
MODEM RESULT CODES

SHORT FORM	LONG FORM	MEANING
0	OK	Command received okay
1	CONNECT	Connect at 300 BPS
2	RING	Ring detected
3	NO CARRIER	No carrier detected
4	ERROR	Error in command
5	CONNECT 1200	Connect at 1200 BPS
6	NO DIAL TONE	No dial tone found
7	BUSY	Busy signal
8	NO ANSWER	No answer
10	CONNECT 2400	Connect at 2400 BPS

S-REGISTERS

The S-Registers define various modem configuration settings, and are specified using the ATS command as described in the preceding section. For example, register S0 lets you specify how many times the phone must ring before the modem is to answer it. You may read and write these registers, using the ATS command described earlier.

S0 ***Ring to answer.*** Specifies the number of times the phone must ring before the modem will pick up. If set to 0, auto answer is disabled.

S1 ***Ring count.*** Counts the number of rings. Resets to 0 if a ring is not sensed within any 8-second interval. S1 operates only when S0 is set to a value greater than 0.

S2 ***Escape code character.*** Sets the ASCII value of the character used to send an escape sequence. The default is 43 (+).

S3 ***Carriage return character.*** Sets the ASCII value of the character recognized as a carriage return. The default is 13.

S4 ***Line feed character.*** Sets the ASCII value of the character recognized as a line feed. The default is 10.

S5 ***Backspace character.*** Sets the ASCII value of the character recognized as a backspace. The default is 8. This register is limited to values between 0 and 32.

S6 ***Wait for dial tone.*** Sets the number of seconds the modem should wait for a dial tone before returning a "NO DIAL TONE" message.

S7 ***Wait for carrier after dial.*** Sets the number of seconds the modem should wait for a carrier after dialing before issuing a "NO CARRIER" message.

S8 ***Pause time for comma.*** Sets the number of seconds to pause for each comma encountered within an ATD (dial) command. The default is 2 seconds.

S9 ***Carrier detect response time.*** Sets the number of seconds a received carrier must be present for before the modem will recognize it as a valid carrier signal.

S10 ***Lost carrier to hang up delay.*** Sets the number of seconds the modem should wait before hanging up the line when the carrier is lost.

S25 ***Delay to DTR.*** This is the number of seconds the DTR line must be false before the modem is to hang up the line.

APPENDIX - B

TERMINAL EMULATION CONTROL CODES

This section documents all of the control codes that are recognized by the various emulations provided with PDQComm. Note that character code values are enclosed in parenthesis, and except for the Data General D215 are given as decimal numbers. The symbols *row, col*, and # are used to indicate a numeric value you provide that specifies either a row, a column, or some other numeric parameter.

TTY CONTROL CODES RECOGNIZED BY **TTYDISP.BAS**

CODE (DECIMAL)	*FUNCTION*
(7)	Beep
(8)	Backspace
(9)	Tab
(10)	Line Feed
(12)	Form Feed (Clear Screen)
(13)	Carriage Return

ANSI CONTROL CODES RECOGNIZED BY **ANSIDISP.BAS**

CODE(DECIMAL)	*FUNCTION*
(7)	Beep
(8	Backspace
(9)	Tab
(10)	Line Feed
(12)	Form Feed (Clear Screen)
(13)	Carriage Return
(27)(91)row(59)col(72)	Set Absolute Cursor Position
(27)(91)row(59)col(70)	Set Absolute Cursor Position
(27)(91)#(65)	Move Cursor Up # Lines
(27)(91)#(66)	Move Cursor Down # Lines
(27)(91)#(68)	Move Cursor Forward # Columns
(27)(91)#(69)	Move Cursor Backward # Columns
(27)(91)(115)	Save Cursor Position
(27)(91)(117)	Restore Cursor Position
(27)(91)(50)(74)	Erase Display
(27)(91)(75)	Erase Line
(27)(91)#(59)#(59)(109)	Set Graphics Rendition

The sequence #(59) above may be repeated as often as needed, where the value given as # is as follows:

0 - All Attributes Off
1 - Bold On
4 - Underline On
5 - Blink On
7 - Reverse Video On
8 - Concealed On
30 - Black Foreground
31 - Red Foreground
32 - Green Foreground
33 - Yellow Foreground
35 - Magenta Forground
36 - Cyan Foreground
37 - White Foreground
40 - Black Background
41 - Red Background
42 - Green Background
43 - Yellow Background
44 - Blue Background
45 - Magenta Background
46 - Cyan Background
47 - White Background

DATA GENERAL **D215** CODES RECOGNIZED BY **D215DISP.BAS**

CODE (OCTAL)	*FUNCTION*
(007)	Beep
(004)	Blink Disable
(003)	Blink Enable
(017)	Blink Off
(016)	Blink On
(015)	Carriage Return
(032)	Cursor Down
(010)	Cursor Home
(031)	Cursor Left
(030)	Cursor Right
(027)	Cursor Up
(035)	Dim Off
(034)	Dim On
(013)	Erase To End Of Line
(036)(106)(106)	Erase To End Of Screen
(014)	Erase Screen
(012)	New Line

(002) or (036)(105)	Reverse Video Off
(026) or (036)(104)	Reverse Video On
(025)	Underscore Off
(024)	Underscore On
(020)(col)(row)	Set Absolute Cursor Position

VT52 CONTROL CODES RECOGNIZED BY **VT52DISP.BAS**

CODE (DECIMAL)	*FUNCTION*
(7)	Beep
(8)	Backspace
(9)	Tab
(10)	Newline
(12)	Form Feed (Clear Screen)
(13)	Carriage Return
(27)(65)	Cursor Up
(27)(66)	Cursor Down
(27)(67)	Cursor Right
(27)(68)	Cursor Left
(27)(72)	Cursor Home
(27)(73)	Reverse Line Feed
(27)(74)	Erase To End Of Screen
(27)(75)	Erase To End Of Line
(27)row col(89)	Set Absolute Cursor Position

VT100 CONTROL CODES RECOGNIZED BY **VT10DISP.BAS**

CODE (DECIMAL)	*FUNCTION*
(7)	Beep
(8)	Backspace
(9)	Tab
(10)	Line Feed
(12)	Form Feed (Clear Screen)
(13)	Carriage Return

IF **VT52** COMPATIBLE MODE

CODE (DECIMAL)	*FUNCTION*
(27)(65)	Cursor Up
(27)(66)	Cursor Down

(27)(67)	Cursor Right
(27)(68)	Cursor Left
(27)(72)	Cursor Home
(27)(73)	Reverse Line Feed
(27)(74)	Erase To End of Screen
(27)(75)	Erase To End of Line
(27)row col(89)	Set Absolute Cursor Position

IF **ANSI** COMPATIBLE MODE

CODE (DECIMAL)	*FUNCTION*
(27)(91)row(59)col(72)	Set Absolute Cursor Position
(27)(91)row(59)col(70)	Set Absolute Cursor Position
(27)(91)#(65)	Move Cursor Up # Lines
(27)(91)#(66)	Move Cursor Down # Lines
(27)(91)#(68)	Move Cursor Forward # Columns
(27)(91)#(69)	Move Cursor Backward # Columns
(27)(91)(115)	Save Cursor Position
(27)(91)(117)	Restore Cursor Position
(27)(91)(74) or	
(27)(91)(48)(74)	Erase From Cursor To End Of Screen
(27)(91)(49)(74)	Erase From Beginning Of Screen To Cursor
(27)(91)(50)(74)	Erase Entire Screen
(27)(91)(75) or	
(27)(91)(48)(75)	Erase From Cursor To End Of Line
(27)(91)(49)(75)	Erase From Beginning Of Line To Cursor
(27)(91)(50)(75)	Erase Entire Line
(27)(91)#(59)#(59)(109)	Character Attributes

The sequence #(59) above may be repeated as often as needed, where the value given as # is as follows:

0 - All Attributes Off
1 - Bold On
4 - Underline On
5 - Blink On
7 - Reverse Video On

Index

INDEX

A

AT Command Set . . . . . 4-11, Appendix A
AdjustRecBuffer 3-1
ANSI Emulation . . . . . 2-12 to 2-17, 3-33
  ANSIInit 3-34
  ANSIPrint 3-34
ANSI.SYS . . . . . 2-11 to 2-17
ASC 2-3
ASCII File Transfer . . . . . 2-17 to 2-18
ASCIIReceive% 3-2
ASCIISend% 3-3
Asynchronous Communications 4-1

B

Baud 4-3
BIN 2-3
BIOSInkey 3-3
BIOSPrint 3-4
Bits Per Second 4-3
Bulletin Board System (BBS) . . . . . 1-1, 2-8 to 2-10

C

Carrier Detect 4-6
Carrier% 3-4
CD[n] 2-4
Checksum$ 3-5
CloseCom 3-6
CLS 2-15
COLOR 2-11, 2-13, 2-15
ComEof% 2-5, 3-6
ComInput$ 2-5, 2-10, 3-7
ComLineInput 2-10, 3-8
ComLoc% 2-5, 3-9
ComPrint 3-9
CRC16$ 3-10
CS[n] 2-4
CTS/RTS . . . . . 2-6 to 2-7, 4-6

D

D215 Emulation . . . . . 2-12 to 2-17
  D215Init 3-33
  D215Print 3-34
Data Set Ready 4-5

Data Terminal Ready 4-5
DCE 4-4
DS[n] 2-4
DTE 4-5
DTR 3-10

E

EOF() 2-5

F

Far Strings 2-1
File Transfers . . . . 2-17 to 2-18
  ASCIIReceive% 3-2
  ASCIISend% 3-3
  Checksum$ 3-5
  CRC16$ 3-10
  XModemReceive% 3-30
  XModemSend% 3-31
  XStat 2-17
FlushBuffer 3-11
Full Duplex 4-1, 4-3

G

GetComPorts 3-11
GetLineStatus 3-12
GetPortConfig 2-8, 3-13

H

Half Duplex 4-3
Handshaking . . . . 2-6 to 2-7, 4-7
  CTS/RTS 4-5
  RTS 3-23
  SetHandshaking 3-29
  XOff% 3-33
  XON/XOFF . . . . 1-5, 2-5 - 2-7
Hayes . . . . 4-10, Appendix A

I

INPUT$ 2-5
IRQ Restrictions 1-5
  OpenComX 3-16

L

LF 2-3
LINE INPUT 2-10

LOC() 2-5
LOCATE 2-11, 2-13, 2-15

M

Mark State 4-1
ModemType 3-13, 3-18
Modems 4-9
  AT Command Set . . . . 4-11, Appendix A
  Hayes . . . . 4-10, Appendix A
  Null Modem . . . . 4-4, 4-7 to 4-8
Multiple Emulation Windows 2-16

N

NS16550 . . . . 1-5, 4-14
Null Modem . . . . 4-4, 4-7 to 4-8

O

OneColor% 3-14
OPEN "COM" 2-3
  ASC 2-3
  BIN 2-3
  CD[n] 2-4
  CS[n] 2-4
  DS[n] 2-4
  LF 2-3
  OP[n] 2-4
  RS 2-4
  TB[n] 2-4
OpenCom 2-3, 2-5, 3-14
  GetLineStatus 3-12
  SetActivePort 3-25
OpenComX 3-1, 3-16
  AdjustRecBuffer 3-1
  SetCom 2-8, 3-26
OP[n] 2-4
OverRun% 3-17

P

Parity 4-2
ParseComParam 3-18
Pause 3-19
PDQExist% 3-19
PDQParse$ 3-20
PDQPrint 2-10, 3-21
PDQRestore 3-22
PDQTimer& 3-22

PDQValI 3-23
PDQValL 3-23
PRINT 2-10

Q

Quick Libraries 2-2

R

Receive Buffer 2-6
  AdjustRecBuffer 3-1
  FlushBuffer 3-11
  OverRun% 3-17
RS 2-4
RS-232C 4-4
  Carrier Detect 4-6
  Carrier% 3-4
  Data Set Ready 4-5
  Data Terminal Ready 4-5
  DCE 4-4
  DTE 4-5
  Full Duplex 4-1, 4-3
  Half Duplex 4-3
  Parity 4-2
  Simplex 4-3
RTS/CTS . . . . 2-6 to 2-7
RTS 3-23

S

ScanCodes% 3-24
SendBreak 3-24
SetActivePort 3-25
SetANSIWindow 3-30
SetCom 2-8, 3-26
SetComPrintTO 3-27
SetD215Window 3-30
SetDelimitChar 3-28
SetFIFO 3-28
SetHandshaking 3-29
SetMCRExit 3-29
SetTTYWindow 3-29
SetVT100Window 3-29
SetVT52Window 3-29
Simplex 4-3
Space State 4-2
Synchronous Communications 4-1

T

TB[n] 2-4
Terminal Emulations . . . . 2-11 to 2-17, Appendix B
  Multiple Emulation Windows 2-16
  Set*xxxx*Window 3-29
  *xxxx*Init 3-33
  *xxxx*Print 3-34
TermType . . . . 2-12 to 2-17
TSR 2-11
TTY Emulation . . . . 2-12 to 2-17
  TTYInit 3-34
  TTYPrint 3-34

U

UARTS . . . . 4-11 to 4-14
  NS16550 . . . . 1-5, 4-15
  SetFIFO 3-28
UARTType% 3-32

V

VT100 Emulation . . . . 2-12 to 2-17
  VT100Init 3-34
  VT100Print 3-34
VT52 Emulation . . . . 2-12 to 2-17
  VT52Init 3-34
  VT52Print 3-34

X

XMODEM . . . . 2-17 to 2-18
XModemReceive% 3-30
XModemSend% 3-31
XOff% 3-33
XON/XOFF . . . . 1-5, 2-5 to 2-7
XStat 2-17

11 BAILEY AVENUE, RIDGEFIELD, CONNECTICUT 06877
PHONE: (203) 438-5300 - SALES: 1-800-35-BASIC

CRESCENT
SOFTWARE, INC.