Async Professional ActiveX™

TurboPower Software Company
P.O. Box 49009
Colorado Springs, CO 80949-9009

Order line (U.S. and Canada): 800/333.4160
Elsewhere: 719/260.9136
Fax: 719/260.7151

www.turbopower.com

Table of Contents

Chapter 1: Introduction

Whether you are new to TurboPower or one of our many long-time customers, we would like to express our sincere appreciation for the confidence you have placed in us by choosing Async Professional ActiveX (APAX) for your serial communications needs.

The APAXPort ActiveX control contains all of the necessary functionality required to build any application, from simple direct serial communications to large scale applications requiring complex data communications facilities. All of the tools you need to perform direct serial, network, and Internet communications, file transfers, terminal emulations, automated data recognition, TAPI control, and session logging are packaged into the APAXPort control.

This manual attempts to simplify the broad functionality of the APAXPort control by addressing its capabilities in logical groupings as follows:

- Serial ports (RS-232 and RS-485) and logging facilities
- Winsock operation
- Terminal emulations
- TAPI devices
- File transfer protocols
- Data trigger management
- Visual features including status bar and tool bar

The first few chapters of this manual introduce the APAXPort ActiveX control and cover (at a fairly detailed level) the basics of serial communications. In using the APAXPort control, this is probably more information than you'll ever be required to know. The intent of including it in the manual was to lay a firm foundation of fundamental serial communications theory.

Chapter 4 provides some general guidelines for testing and troubleshooting serial connections from both a hardware and software vantage point. This list has evolved from years of experience addressing difficult serial communications problems. Should you find yourself experiencing difficulty, this chapter serves as an excellent starting point.

The remainder of this manual is a dedicated reference section to the APAXPort control. Here you'll find the details you need to get up and running quickly with APAX.

As many long-time customers can tell you, TurboPower Software Company is genuinely committed to your success using our products. We welcome all your comments, suggestions, and even constructive criticisms. We continue to strive to fully meet your expectations. Please let us know how we're doing. And thank you again for choosing APAX.

Technical Support

The best way to get an answer to your technical support questions is to post them in the APAX newsgroup on our news server (news.turbopower.com). Many of our customers find the newsgroups a valuable resource where they can learn from others' experiences and share ideas in addition to getting answers to questions.

To get the most from the newsgroups, we recommend that you use dedicated newsreader software. You'll find a link to download a free newsreader program on our web site at www.turbopower.com/tpslive.

Newsgroups are public, so please do not post your product serial number, 16-character product unlocking code or any other private numbers (such as credit card numbers) in your messages.

The TurboPower KnowledgeBase is another excellent support option. It has hundreds of articles about TurboPower products accessible through an easy-to-use search engine (www.turbopower.com/search). The KnowledgeBase is open 24 hours a day, 7 days a week. This provides another way to find answers to your questions even when we're not available.

Additionally, you can read about support options at www.turbopower.com/support.

System Requirements

To use the APAXPort ActiveX control you must have the following hardware and software:

1. A computer capable of running Microsoft Windows 95/98/2000/ME or Windows NT. A minimum of 16MB of RAM is recommended.
2. Any development platform that serves as an ActiveX container. The most common examples are Visual Basic, Visual C++, and Borland's Delphi.
3. A hard disk with at least 50MB of free space is strongly recommended. To install the APAXPort control and all of the example programs requires about 5MB of free disk space.

Installation

APAX can be installed directly from the TurboPower Product Suite CD-ROM. Before installing the product, please scan through the README.TXT file to find any late news that may affect installation.

To install from the CD, insert the TurboPower Product Suite CD and follow the instructions presented by the SETUP program. SETUP installs APAX in the C:\APAX directory by default. You can specify a different directory, if desired.

The setup program registers the APAXPort control with Windows. You will then need to install the control into your development environment. For development environments other than Visual Basic, consult the documentation pertaining to importing and using ActiveX controls. For Visual Basic developers, follow the instructions below:

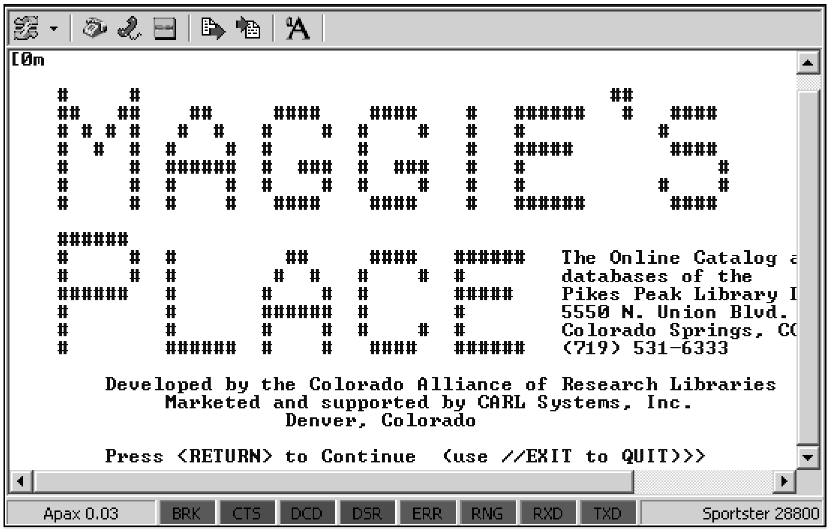
1. Start Visual Basic.
2. Click on the Tools menu or right-click on the Visual Basic toolbox.
3. Select Custom Controls from the menu. A dialog displaying the entire list of custom controls will be displayed.
4. Ensure that the check box labeled Controls is checked. The list should now display all of the ActiveX controls registered in your system.
5. Locate the TurboPower APAXPort control in this list and ensure that a check mark appears in the box adjacent to this entry.
6. Click OK and the APAXPort control will appear in the Visual Basic toolbox.

You are now ready to start using the APAXPort control in your applications.

An Overview of the APAXPort ActiveX Control

This section is intended to give you a very brief introduction to the APAXPort control. For functionality beyond this basic introduction, refer to the APAXPort reference chapters.

Once running in an application, the APAXPort control will (by default) appear in your form as the following figure illustrates.

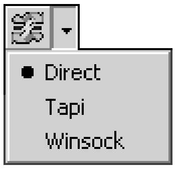


```
[0m

    #     #                                       ##
    ##   ##    ##      ####    ####    #   ######   #   ####
    # # # #   #  #   #    #  #    #   #   #           #
    #  #  #  #    #  #       #        #   #####        ####
    #     #  ######  #  ###  #  ###   #   #                #
    #     #  #    #  #    #  #    #   #   #           #    #
    #     #  #    #   ####    ####    #   ######       ####

    ######
    #     #  #         ##     ####   ######   The Online Catalog a
    #     #  #        #  #   #    #  #        databases of the
    ######   #       #    #  #       #####    Pikes Peak Library I
    #        #       ######  #       #        5550 N. Union Blvd.
    #        #       #    #  #    #  #        Colorado Springs, C
    #        ######  #    #   ####   ######   (719) 531-6333

        Developed by the Colorado Alliance of Research Libraries
             Marketed and supported by CARL Systems, Inc.
                       Denver, Colorado

        Press <RETURN> to Continue  (use //EXIT to QUIT)>>
```

A surprising amount of functionality is already built into the control—without writing a single line of code! Simply choose a device type, select an appropriate device, click the Connect button, and you're ready to communicate! The paragraphs below introduces you to the visual features of the APAXPort control.

Choosing a device type and establishing a connection

The APAXPort control allows for three different device connection types: Direct, TAPI, or Winsock. To select a device type, just click the Device Type button in the upper left of the control and make your selection.



Direct devices

To communicate directly with a remote device such as a weight scale or other data-collection device, select the Direct device type. APAX will then provide you with a list of all the available COM ports on your system. All you have to do is select the desired COM port.

TAPI devices

Selecting TAPI as a device type automatically presents you with a list of all the TAPI devices registered on your system. Here, all that is required is to select a device from this list and specify the phone number to dial.

Winsock devices

APAX provides for network and Internet communications as well. Select Winsock as your device type to specify a connection of this type. To complete the connection details for a Winsock device, specify a computer address, enter a service (such as ftp) or port number (e.g. 21), and instruct APAX to act as either a client or a server.

Note: APAX is limited to a single client connection if acting as a server.

Establishing a connection

If your device selection is anything but Winsock server mode, all you need to do to complete the connection is to click the Connect button.

If you have selected a device type of Winsock server mode, you'll need to listen for connections rather than connecting directly. To do this, simply click the Listen for Connection button.

Initiating a file transfer

To initiate a file transfer (using Zmodem as the default) or receive a file transfer from a remote machine, just click the corresponding button on the tool bar. If you are initiating the file transfer, the standard Windows File Open dialog will be displayed. Navigate to and select the file you want to send.

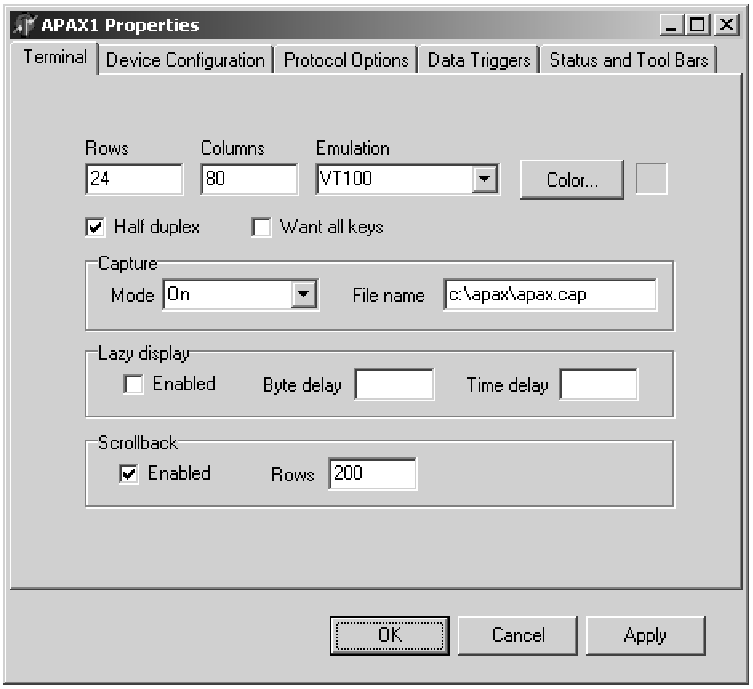
Setting the terminal font

The font that the terminal displays can be set by clicking the Font button and selecting the desired font from the standard Windows dialog.

Using the APAXPort's Property Pages

APAX provides several property sheets that assist you in setting the most commonly used design time properties. To access the property pages, right-click on the component at design time and select Properties from the context menu. APAX's property sheets simplify the configuration of the control as it relates to the terminal, basic device configuration, file transfer protocols, data triggers, and status light property settings. Following is a brief introduction to each of APAX's property sheets.

Terminal property page



Rows: 24
Columns: 80
Emulation: VT100
[✓] Half duplex
[ ] Want all keys

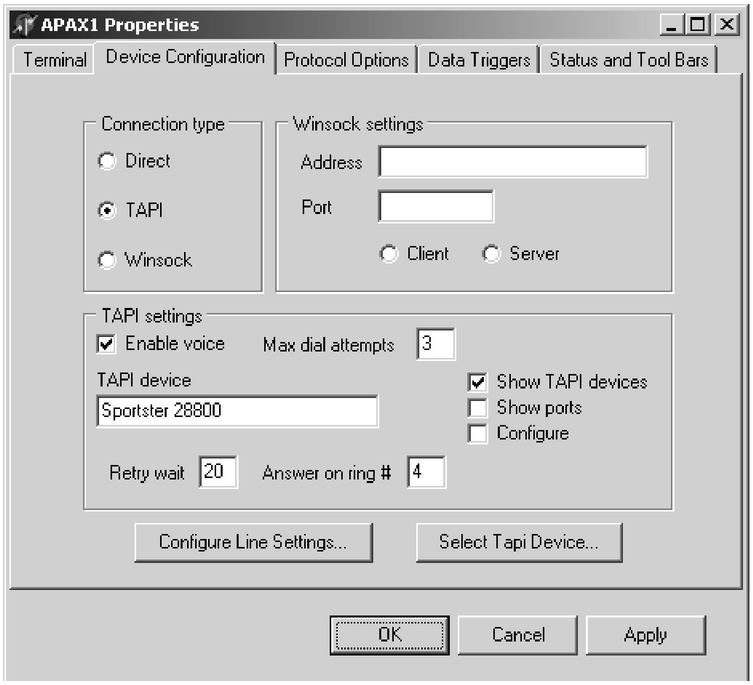
Capture
Mode: On
File name: c:\apax\apax.cap

Lazy display
[ ] Enabled
Byte delay:
Time delay:

Scrollback
[✓] Enabled
Rows: 200

The Terminal property page allows you to specify the number of rows and columns displayed by the terminal window, the emulation mode (TTY, VT100, or VT52), the terminal window's background color (in TTY emulation mode only), and various general properties that define the terminal window's behavior. A capture file can also be specified and capture mode can be enabled to record a terminal session to a disk file. The terminal property page also allows you to specify the size (in lines) of the scrollback buffer.

Device configuration property page



Connection type: ( ) Direct [✓ TAPI] ( ) Winsock

Winsock settings: Address: ___ Port: ___ ( ) Client ( ) Server

TAPI settings:
[✓] Enable voice
Max dial attempts: 3
TAPI device: Sportster 28800
[✓] Show TAPI devices
[ ] Show ports
[ ] Configure
Retry wait: 20
Answer on ring #: 4

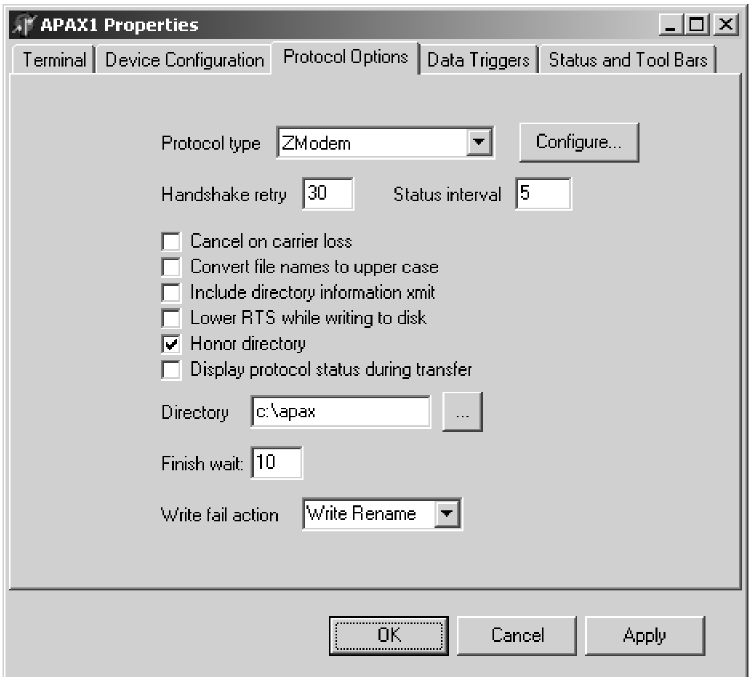
The Device Configuration property page allows you to specify the device type (Direct, TAPI, or Winsock).

If a Direct device type is selected, click the Configure Line Settings button to invoke an editor that allows you to specify the port number, baud rate, parity, data bits, and stop bits.

If a TAPI device type is selected, then this property page allows you to specify the TAPI device to use, whether or not voice is enabled, dial attempts, dial retry wait, and answer on ring settings. You can also specify what devices are displayed to your end user and filter serial ports (COMx) from the displayed list.

If a Winsock device type is selected, the device configuration property page allows you to specify the remote computer's address, and the port. You can specify this value as a service name such as ftp, or you can enter the port number directly. You can also specify whether the Winsock device should act as a client that initiates a connection or as a server that simply listens for incoming connections.

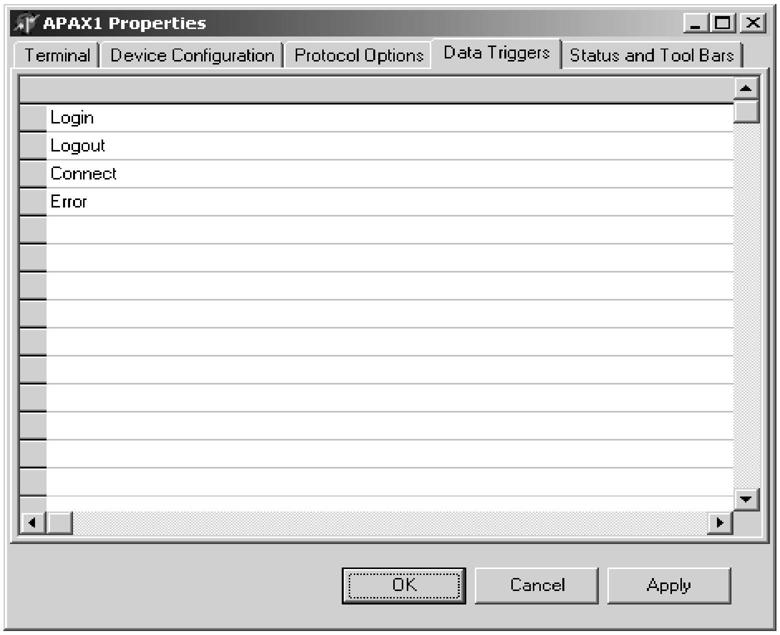
Protocol options property page



Protocol type: ZModem
Handshake retry: 30
Status interval: 5
[ ] Cancel on carrier loss
[ ] Convert file names to upper case
[ ] Include directory information xmit
[ ] Lower RTS while writing to disk
[✓] Honor directory
[ ] Display protocol status during transfer
Directory: c:\apax
Finish wait: 10
Write fail action: Write Rename

The Protocol Options property page allows you to select the type of file transfer protocol used to send and receive files. After selecting a protocol type, click the Configure button to specify additional parameters that are specific to the type of protocol selected. Several other properties that apply to all protocols are available as well on this property page.

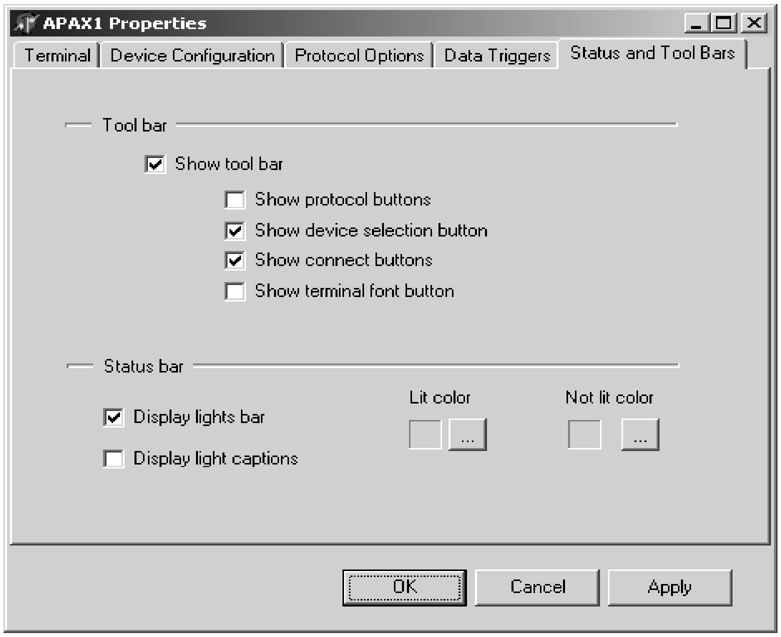
Data triggers property page



Login
Logout
Connect
Error

The Data Triggers property page provides a simple grid that allows you to specify multiple data triggers. Each individual data trigger should be entered on a single line. APAX concatenates these individual strings and separates them with the pipe (“|”) character and assigns this value to the DataTriggerString property. The DOS wildcard symbols (“?” and “*”) can be used within each string entered.

Status lights property page


APAX1 Properties

Terminal | Device Configuration | Protocol Options | Data Triggers | Status and Tool Bars

Tool bar

- [✓] Show tool bar
  - [ ] Show protocol buttons
  - [✓] Show device selection button
  - [✓] Show connect buttons
  - [ ] Show terminal font button

Status bar

- [✓] Display lights bar
- [ ] Display light captions

Lit color ... Not lit color ...

OK | Cancel | Apply

The Status Lights property page allows you to specify which (if any) of the status line lights appear on the APAXPort control. This property page also allows you to specify the colors (lit and not lit) of the lights and whether or not their corresponding caption should be superimposed. In addition, you can configure the toolbar from this property page. You can hide or show the toolbar altogether, or you can set the visible state of individual toolbar buttons.

Chapter 2: Communications Basics

Communications is a very difficult field of programming. Several factors contribute to this perception. First, many areas of communications programming lack sufficiently thorough (and understandable) documentation. Second, the typical communications program is expected to interface with a variety of hardware, software and drivers (all which may have slightly different behaviors). Additionally, the many issues that are dealt with during the course of programming a communications application are often very difficult to troubleshoot.

One of the biggest goals of APAX is to insulate you from these difficulties. It’s possible for an average programmer to work some communications functionality into an application with a minimum of communications knowledge by using an abstract control like APAX. Obviously, there is a certain amount of required knowledge to tackle something in a difficult field such as communications—but we hope to give you a good head start in any case.

This chapter introduces some of the communications fundamentals in a fairly basic way and describes how the functionality of APAX fits into the picture. If you have been programming in the field of communications for some time, you may be tempted to skip this chapter. However, there’s always the possibility that you may pick up a few tips here and there regarding how the different areas of APAX’s functionality fit into the overall equation. If you have a need for more detailed information, refer to “Chapter 3: Advanced Communication Principles” on page 19.

Communications Overview

As mentioned in the previous section, communications is a tough field. Many people consider it a black art, and that designation is very well deserved in many circumstances. Programming for environments such as Windows simplifies things in some ways, and makes things much more frustrating in other ways.

Some of the advantages of programming in Windows include the fact that you don't need to know things such as the intricacies of UART operation or the way bytes are framed by start/stop bits. It's now possible to save study of those topics for a rainy day. Not only is knowledge of what's going on at that level unnecessary, but it's often the case that the operating system actually prevents you from accessing the hardware directly.

The next few sections present communications basics in what is hoped to be a fairly simple manner. This chapter is by no means intended to be a comprehensive guide to data communications; it's merely intended to give you a base knowledge so you can start to understand the functionality available in APAX. Be sure to refer to the applicable sections of the reference chapters to get more information about the various areas of APAX functionality.

Data In/Out

Every communications program has at least one thing in common: the fact that the program communicates with other programs or devices. This means that data flows into or out of the application in some way. In most applications, data flows in both directions, but it's certainly possible for an application to only receive or send data. The data can be binary or text, but it's always treated as a stream of bytes by the low-level hardware and drivers. The methods of communication may differ slightly—they could be though an RS-232 serial port, an RS-485 port, a parallel port, or network communications card. The actual implementation of these methods differ, but the net result that bytes flow in and/or out of the program remains the same.

APAX provides all of the functionality you need to send and receive data over a serial or network connection. When communicating directly over a serial connection, APAX handles all of the necessary communications to the Windows communications drivers. APAX provides support for both the RS-232 protocol as well as the RS-485 protocol. Of these two protocols, RS-232 is by far the most common. Additionally, the Winsock services provided by APAX allow you to establish network and Internet connections to send and receive data.

The low-level functionality of APAX provides an interface to the rest of the world for your program. No longer do you need to concern yourself with the intricate details of serial communications. Using APAX, you can even define multiple data triggers that essentially monitor the input stream and notify you automatically when crucial data arrives.

Terminals and Emulators

Terminals display data and emulators decide how the data should displayed. They are always used together in APAX. Without an emulator, a terminal would have nothing to display. Without a terminal, an emulator would have nowhere to display the data it had interpreted.

An emulator has two jobs. It must, first and foremost, interpret the data coming through a serial communications device from a remote host computer. This incoming data will contain text to be displayed on the terminal and it will also contain terminal control sequences. These are sets of characters which, taken as a whole, denote commands for the terminal. Examples of such commands are to scroll the text on the terminal display, to switch from one character set to another, to move or position the cursor, and so on. The emulator thus has to separate out the incoming data stream into terminal control sequences (e.g., “move the cursor to row 1 column 1”) and the text that is to be displayed at the cursor (e.g., “Hello, world”).

The second job for the emulator is to convert keystrokes into their terminal equivalents. With many terminals, pressing a function key on the keyboard results in a sequence of characters being sent to the host computer. The host computer can identify this sequence and know which key was pressed. Obviously, the alphanumeric keys would generate the corresponding character, and there would be no conversion required.

Hence, it is the emulator that provides the characteristics of a given terminal. APAX supports two emulation protocols: TTY, which provides teletype emulation, and VT100 which provides true VT100 emulation. Either of these emulations can be selected via a single property setting aptly named Emulation. The terminal, on the other hand, has it easy—it merely displays the text that the emulator provides.

Modems and TAPI

Modems are hardware devices that make it possible to connect to a phone line via a standard serial port. Modem stands for MOdulator/DEModulator. They modulate the outgoing serial data so it can be properly transmitted along the phone line, and demodulate the incoming data, so it can be received by the serial port. Variations on this theme include ISDN modems, cable modems, and so on. Modems are somewhat difficult to work with in a general manner since there are several different standards used by modem manufacturers. These standards define the communication between an application and the modem. Your application must be able to adapt to those differences.

APAX has several internal tools that are designed to simplify the process of adapting to, and communicating with, different modems. All of the functionality you'll need to communicate with modems is built into the APAXPort control.

Telephony Application Programming Interface (TAPI) is an attempt by Microsoft to simplify the process of communication with the modem. It is a standardized API with which the application can interface, making it simple (in theory) to communicate with many different modems in a standardized way. Under TAPI, a great deal of the burden (and also control) is removed from the application developer in favor of placing the burden on the designers of the operating system and the manufacturers of the modems. This point has good and bad sides—it's great if everything works as planned, but can be very frustrating if things don't work so well. Again, TAPI support is already integrated in the APAXPort

Protocols

Protocols come in many flavors. The word “protocol” simply refers to a standard way of doing something. A log on/log off process on a BBS can actually be considered a protocol. A simple send string/receive string protocol, such as the one used to check Internet mail (POP3) is best handled using the data trigger mechanism built into APAX. Other protocols, such as the ones used to transfer binary data, are a bit more complicated and require the use of complex state machines to track how the process is progressing.

APAX provides a built in file transfer protocol engine that handles all of the most common protocols automatically. The protocol engine greatly simplifies the process of sending and receiving binary and ASCII text files. In fact, all you have to do is set a few properties and call a method to get things going in many cases.

Chapter 3: Advanced Communication Principles

This chapter provides a more advanced discussion of some the principles of serial communications and the general issues of serial communication under Windows. An APAX user is typically insulated from many of these details; both by APAX's encapsulation of the various communication APIs and by the layers of drivers that exist in Windows. With this in mind, it is possible to write a successful APAX program without reading (or fully understanding) this chapter.

This chapter is mainly intended for users who want (or need) a more detailed understanding of some of the low-level processes that may be affecting their application. This chapter is not intended to be a comprehensive guide to all aspects of serial communication.

Basics of Asynchronous Communication

After presenting some basic concepts, this discussion continues into some of the lower-level details of serial communications (starting with a detailed discussion of the UART chip). Generally, you do not need such detailed knowledge to use APAX effectively.

The broadest definition of serial communications includes anything that transmits or receives data in a serial fashion, where one bit follows another in a single stream over one wire. In parallel communications, by contrast, many bits are sent in parallel over many wires. Asynchronous serial communications simply means that the data stream includes start and stop bits, bits that mark the beginning and end of each character in the data stream. This is in contrast to synchronous communications where no start or stop bits are provided and the two ends of the link rely on synchronized clocks to know where each character starts and stops.

In the PC world, when people speak of serial communications they are invariably talking about the communications facility provided by the serial ports (or COM ports) at the back of a PC. To these ports you can attach a wide variety of input and output peripheral devices. In fact, you can attach and communicate with anything that adheres to the same serial communications standard as the serial port.

Given the wide variety of serial peripherals available and the corresponding variety of applications, it's sometimes hard to find a general purpose term for what you have attached to your serial port. When the term is too general it can be hard to understand. When the term is too specific it might not be clear how the point relates to other cases. In this manual the appropriate terms are used as they are called for. The manual uses the term "device" or "remote device" or just "remote" when the kind of device is not really important. When the point relates to a specific device such as a modem, the more specific term is used.

Information presented throughout this manual refers to something called a Universal Asynchronous Receiver/Transmitter (UART). This is the chip within your PC that handles the low-level details of receiving and transmitting data. You don't really need to know any more beyond that. If you are interested in learning more about the UART, read "Universal Asynchronous Receiver/Transmitter (UART)" on page 24.

Line parameters

Because serial communications are somewhat standardized, you don't need to know the lowest level details such as line voltages, pin names, and so forth. However, you do need to know about line parameters—baud rate, parity, data bits and stop bits—which specify the transfer rate and format of the data on the serial line. Usually, the line parameters are expressed like this:

```
9600,N,8,1
```

This describes a communications link operating at 9600 baud, no parity bit, eight data bits, and one stop bit. Both ends of the link must be using the same line parameters before any communication can take place.

You specify line parameters in your APAX application whenever you open a serial port. In some cases, you might not have any leeway at all—the device you want to communicate with might force particular line parameters upon you. More likely, though, you'll need to choose appropriate line parameters for your PC and the device attached to your serial port. Here are brief definitions of these line parameters and some guidelines for choosing appropriate values.

Baud rate

Baud rate is commonly used to mean bit rate—the number of bits transmitted per second. This is technically incorrect. Baud rate actually means the number of events per second in a communications line. Since an event can contain information about more than one bit, as is the case with high-speed modems, baud rate could be quite different than bit rate. At the serial port itself (where APAX usually concerns itself) each event is a single bit, so equating baud rate and bit rate is accurate.

When given a choice, you should generally select a baud rate as high as possible to give you the highest possible throughput. Understand, however, that there are very likely factors in your application or environment that might limit your throughput. The speed of your PC, the type of UART, the quality of the Windows communications driver, and the behavior of concurrently running tasks all affect the highest achievable communications speed.

Generally, any '386 machine should be able to achieve 9600 baud. Faster '486 and Pentium machines can achieve higher speeds, in some cases up to the limit of the Window communications driver, 115.2K baud. Due to the architecture of Windows, even the fastest machines may sometimes lose data. See "Performance issues" on page 40 for more information on getting the best performance out of your Windows machine.

Other factors affect your choice of baud rate as well. For example, if you're using a 2400 baud modem there is little reason for selecting 38400 baud transfer between your PC and the modem. Or, if you are collecting data from a device that sends only a few hundred bytes of data per minute, there's no sense in selecting a high baud rate. You would do better to select a lower baud rate, which would minimize transmission errors.

Data bits

A data byte can contain 5, 6, 7, or 8 bits. The vast majority of applications use either 7 or 8 bits since binary data is expressed in 8-bit bytes and text data can often be expressed in only 7 bits.

Many time-sharing systems, such as CompuServe, work with only 7 data bits because that's all they need to display text data. However, when binary data is transferred with a file transfer protocol, the system almost always switches to 8 data bits. In fact, Kermit is the only file transfer protocol in APAX that works with 7 data bits.

Stop bits

Stop bits follow the data bits in the serial data stream to mark the end of each data byte. The value for stop bits is always either 1 or 2. Generally, you should use 1 stop bit.

Parity

Parity describes a bit checking scheme. When used, all of the bits in each data byte are added together. A final bit, called the parity bit, is added so that the sum of all bits is either odd or even, whichever you specify. The transmitter calculates and transmits a parity bit. The receiver also calculates a parity bit and compares it to the parity bit it received. If the bits are equal, it is assumed that the character was received without error. Otherwise, it is assumed that there was an error during transmission.

The possible values for parity are:

Value	Result
pNone	No parity bit is added
pEven	A parity bit is added such that the bit sum is always even
pOdd	A parity bit is added such that the bit sum is always odd
pMark	A parity bit of value one is always added
pSpace	A parity bit of value zero is always added

Whether or not you should use parity depends on your application. Generally, you don't need to use parity bits if your application relies on some other means of checking data integrity such as block check characters in a file transfer.

Line errors and breaks

Serial I/O, like all forms of I/O, is subject to errors. A line error has occurred when the characters received by the receiver are different from those sent by the transmitter. This usually happens when the data line is disturbed by electrical interference. Your programs must be prepared to deal with line errors. Typical actions are to discard and ignore errant data or to ask the transmitter to send the data again. The action you take depends on the requirements of your application.

Line errors can also occur if the receiver and transmitter are using different line parameters or if you've selected a baud rate that is too high for the environment in which your application is running.

The following sections describe the types of line errors that can occur, what they mean, and how you can deal with them.

UART overrun error

This error means that a second character arrived at the serial port before the first one was processed. Generally this means that you are running at a baud rate that is too high and your machine is not fast enough to handle the characters as fast as they are arriving. The usual solution is to lower the baud rate until the UART overrun errors go away.

In some cases the problem is not that the baud rate is too high, but that another process is leaving interrupts disabled for too long or another virtual machine is hogging the CPU. See "Performance issues" on page 40 for more information about dealing with UART overruns.

Parity errors

Parity errors occur when the parity bit received differs from the parity bit calculated. If the receiver specifies odd parity and receives a character with even parity, a parity error results. The most common cause of parity errors is a mismatch in the parity line parameter between the transmitter and the receiver. Always suspect this if you get a lot of parity errors when you first connect to a new device. Another clue is when certain characters always return parity errors and other characters never return parity errors.

Parity errors can also be caused by interference on the data line (i.e., transmission errors). When this is the case, errors occur randomly with groups of errors interspersed with long periods of error-free transmission. In this case the recommended solution is to reroute the serial cable away from any sources of electrical interference. Shortening the cable could also help.

Framing errors

A framing error occurs when the data bits in the serial stream are not followed by a valid stop bit, which must always have the value '1'. As with parity errors, the most likely cause of framing errors is a mismatch in line parameters between the receiver and transmitter. Always verify the line parameters at both ends of the communication link whenever you get framing errors.

Framing errors can also be caused by electrical interference. Again, the recommended solution for such errors is to shorten or reroute the serial cable.

Breaks

Breaks are not really line errors, but they do represent a special line condition. A “line break” or “break” is a condition in which zero bits are transmitted for at least as long as it takes to send one character (one “character time”). The UART recognizes this special condition and can notify you that it has received a break. Breaks are used when a device needs to signal another device to handle a special condition.

Universal Asynchronous Receiver/Transmitter (UART)

This topic covers the detailed inner workings of the UART chip, which is at the heart of most serial ports. You don’t need to know these details. In fact, Windows insulates applications from these details to such a degree that you can’t apply them even if you know them.

In some circumstances, however, you might find this information helpful for thinking through a debugging problem, or you just might want a clearer picture of what’s really happening on the chip. If you’re just getting started with APAX, or you already know all you care to know about UARTs, you might want to skip this section. But be sure to pick up the discussion at “Flow control” on page 32.

The UART chip is the brain of the serial communications facilities on IBM PCs, PS/2s, and compatibles. In nearly all cases, this chip is from a family of National Semiconductor integrated circuits. Older PCs use UARTs with chip designations INS8250 and INS8250B. Newer machines use NS16450 and NS16550 chips. Although there are slight differences between the chips in speed, internal behavior, and features, their basic properties are the same.

The UART is responsible for all of the grunt work of serial communications. It transmits data by taking a byte and serializing the bits onto the output line. It receives data by reading a stream of bits from the input line and de-serializing them into a data byte. The UART also controls the line parameters discussed earlier, and is responsible for setting and reacting to various line and modem control and status signals.

The UART does all of these things in response to requests from a program. The program exchanges data with the UART through the UART’s registers. To the program, these registers are nothing more than addresses somewhere in the PC’s I/O address space.

The IBM PC architecture also associates a hardware interrupt with each UART. You can use the serial port without using the interrupt, but it’s generally not practical to do so. The names that are typically used to refer to serial ports (Com1, Com2, etc.) tell you the address of the UARTs and what hardware interrupt they use.

These addresses and interrupts are governed by two standards: the IBM PC standard for Com1 through Com4; and the IBM PS/2 standard for Com1 through Com8. The following tables show the addresses and interrupts for each standard:

IBM PC Standard (Com1, 2) and de facto Standard (Com3, 4)

ComName	Base Address	IRQ	Vector
Com1	03F8h	4	0Ch
Com2	02F8h	3	0Bh
Com3	03E8h	4	0Ch
Com4	02E8h	3	0Bh

IBM PS/2 Standard

ComName	Base Address	IRQ	Vector
Com1	03F8h	4	0Ch
Com2	02F8h	3	0Bh
Com3	3220h	3	0Bh
Com4	3228h	3	0Bh
Com5	4220h	3	0Bh
Com6	4228h	3	0Bh
Com7	5220h	3	0Bh
Com8	5228h	3	0Bh

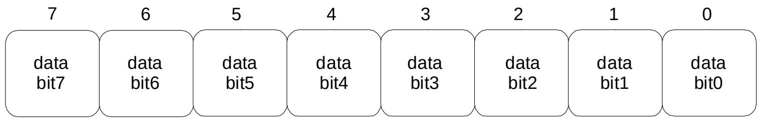
Even though the standards only define up to eight serial ports, many serial port boards support additional serial ports at other base addresses and IRQs. APAX can open any serial port that is defined in Windows by simply using the COM number. Windows defaults to the values shown in the tables above. To inform Windows of a non-standard address or IRQ, you must run Control Panel/Ports or Control Panel/Add New Hardware.

Registers

The Windows communications driver communicates with a UART via the UART registers. It controls the UART by writing information into its registers and it retrieves data and status information by reading the registers. A UART contains eight registers, each with a specific purpose, accessed on the PC through I/O ports starting at the base address and continuing for the next eight port addresses. The register at the base address is called register 0, the next register is register 1, and so on. Since the COM1 UART's base address is 03F8h, register 0 is at 03F8h, register 1 is at 03F9h, and so on up to register 8 at 03FFh.

Each of these registers also has one or more names, as shown in the following descriptions. Registers that provide status information when read are designated (read). Registers that are used to program the UART are designated (write). Those that are used both ways are designated (read/write).

Register 0: receiver buffer register (read)
transmit holding register (write)
divisor latch low (read/write)



Register 0 Bit Definitions

Register 0 has three names and three purposes. When you read from register 0, you are reading the latest received character (if there is one). When you write to register 0, you are passing the next character to be transmitted (if the UART is ready).

The third purpose of register 0 comes into play when setting the baud rate. When the divisor latch access bit is set, register 0 specifies the low byte of the baud rate divisor. The baud rate divisor is a value which, when divided into a preset constant, yields the desired baud rate. This preset constant is determined by an internal clock rate that is the same for all PCs.

The baud rate divisor is determined by this equation:

```
divisor = 115200 / baud rate
```

Hence, the process for setting the baud rate on a UART is to calculate the baud rate divisor, set the divisor latch access bit, write the low byte of the divisor to register 0, write the high byte of the divisor to register 1, and finally clear the divisor latch access bit.

Register 1: interrupt enable register (write) divisor latch high (read/write)

7	6	5	4	3	2	1	0
N/A	N/A	N/A	N/A	modem status	line error/ break	xmit ready	recv char

Interrupt Enable Bit Definitions

UARTs can generate an interrupt in response to four different conditions. Programs specify which interrupt conditions they want to enable by writing into this register. Here are the four interrupt conditions:

- A character was received
- The transmitter just finished transmitting a character
- An error or break signal occurred
- A modem status signal changed

To enable a particular interrupt, set the proper bit in a byte mask and write the byte mask to the interrupt enable register. To disable the condition, reset the bit to 0 and write the byte mask to the interrupt enable register.

Register 1 also has a second name (divisor latch high) and a second purpose. When the divisor latch access bit is set, register 1 becomes the high byte of the baud rate divisor (used for setting the baud rate). See the previous discussion of the divisor latch low register for more information on setting the baud rate.

Register 2: interrupt identification register (read) FIFO control register (write)

7	6	5	4	3	2	1	0
FIFO enabled	FIFO enabled	N/A	N/A		see below		int pend

Interrupt Identification Bits

This is the counterpart to the interrupt enable register. Once you've enabled the desired interrupt conditions and received an interrupt, this register indicates which condition caused the interrupt. Only the three least significant bits of the register are actually used, as described in the following paragraph.

Since it is possible, even likely, that more than one condition can occur at the same time, bit 0 is used to determine whether all conditions that currently exist have been handled. When bit 0 has a value of 0, there are conditions waiting to be handled. When bit 0 has a value of 1, all outstanding conditions have been handled. Bits 3, 2, and 1 taken together identify the cause of the interrupt.

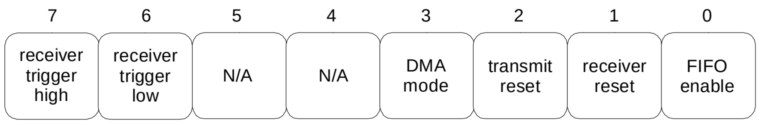
Because multiple conditions can occur at the same time, the UART presents the conditions in a prioritized order. The following table shows the priority used by the UART and the corresponding bit masks:

Bits 3-0	**Priority**	**Interrupt type**
0 0 0 1	None	None
0 1 1 0	Highest	Line error or line break
0 1 0 0	Second	Received data available
1 1 0 0	Second	Received data available (FIFO time-out)
0 0 1 0	Third	Transmitter holding register empty
0 0 0 0	Lowest	Modem status change

The FIFO time-out condition obviously occurs only on UARTs operating with a FIFO (first in, first out) buffer. The FIFO buffer is typically a 16-byte buffer on the UART chip that holds received characters until the application can retrieve them. When the FIFO buffer is filled to a preset level, a received data available interrupt is generated. The FIFO time-out interrupt occurs only when the data does not reach this preset level. The interrupt is generated when characters are waiting in the receive FIFO buffer and four character-times elapse without receiving any new characters.

In addition to providing information about pending interrupt conditions, this register also provides two FIFO status bits. These bits are always 0 for UARTs that don't possess FIFO buffers. They are both set for UARTs that possess the FIFO buffers if FIFO mode is currently activated.

Register 2 doubles as a writable register for enabling and disabling FIFO buffers. In its role as the FIFO control register, the 8 bits have the following meanings:



FIFO Control Bit Definitions

The first step in specifying FIFO control information is always to set bit 0. This enables writing to the other FIFO control bits. When FIFO mode is enabled or disabled, the FIFO data buffers are cleared of all data.

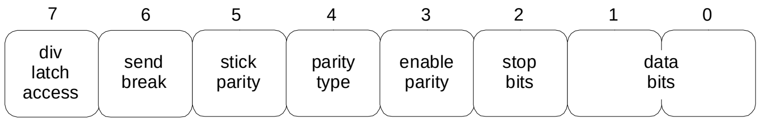
Writing a 1 to bit 1 clears just the receive FIFO buffer. Writing a 1 to bit 2 clears just the transmit FIFO buffer.

Bit 3 is used to control DMA access.

Bits 6 and 7 are used to specify the receive FIFO trigger level, the number of bytes stored in the FIFO before a receive interrupt is generated. The following table shows the possible trigger levels and the corresponding bit values:

`Bit7`	`Bit6`	`Trigger level`
0	0	1
0	1	4
1	0	8
1	1	14

Register 3: line control register (write)



Line Control Bit Definitions

The line control register is used to set the line parameters baud rate, data bits, stop bits, and parity.

Bits 0 and 1 specify the number of data bits to use. The following table shows how these bits are interpreted:

`Bit1`	`Bit0`	`Data bits`
0	0	5
0	1	6
1	0	7
1	1	8

Bit 2 specifies the number of stop bits to use. When bit 2 is set, two stop bits are generated and checked. When bit 2 is clear, one stop bit is generated and checked. (Note that when data bits is 5, setting bit 2 actually specifies 1.5 stop bits.)

Bits 3, 4, and 5 control parity as shown in the following table:

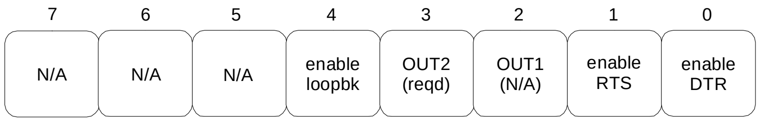
Bit5	Bit4	Bit3	Parity type
0	0	0	None
0	0	1	Odd
0	1	1	Even
1	0	1	Space
1	1	1	Mark

"Space" parity means that a 0 is transmitted after each character regardless of its value. "Mark" parity means that a 1 is transmitted. The UART automatically computes the parity bit and transmits it when appropriate.

Bit 6 is used to generate a line break. While this bit is set, the UART continuously sends zeros on the output line.

Bit 7 is the divisor latch access bit described earlier. When this bit is set, registers 0 and 1 become the divisor latch registers used to set the desired baud rate.

Register 4: modem control register (write)



Modem Control Register Bit Definitions

The primary purpose of the modem control register is to manage the DTR (Data Terminal Ready) and RTS (Request To Send) signals of the serial port. These two signals are also called the handshaking or hardware flow control signals.

Bit 0 controls the state of the DTR signal. Writing a 1 into bit 0 raises the DTR signal and writing a 0 lowers it. Bit 1 controls the state of the RTS signal. Writing a 1 into bit 1 raises the RTS signal and writing a 0 lowers it.

Bit 2, OUT1, is a general purpose output signal, but it's not used in the PC architecture. Bit 3, OUT2, is a general purpose output signal that must always be enabled before interrupts can occur.

Bit 4 enables an internal loopback mode that can be used to test some facets of proper UART operation.

Register 5: line status register (read)

7	6	5	4	3	2	1	0
FIFO error	shift register empty	hold register empty	break received	frame error	parity error	data overrun	char received

Line Status Bit Definitions

This register provides information about line error and line break conditions. It also provides the status of receive operations (whether a character was received) and transmit operations (whether the UART is ready to transmit a character).

Bit 7, the FIFO error status bit, is set if a character in the FIFO has a line error. Generally, you don't need to worry about this because the error is revealed in the normal fashion when the character is extracted from the FIFO buffer.

Bit 6, when set, indicates that the transmitter shift register is empty. The shift register, used internally by the UART, holds the character currently being transmitted while the individual bits are being shifted onto the output data line.

Bit 5, when set, indicates that the transmitter holding register, register 0, is empty. You should never write a character to register 0 unless this bit is set. After a character is placed in the holding register and any character already in the shift register has been transmitted, the UART moves the new character into the shift register and clears bit 5.

Bit 4 is set whenever a line break is received. This also causes a line error interrupt.

Bits 3 through 1, when set, indicate a line error has occurred. The nature of these line errors is discussed earlier in this section. All of these bits also generate interrupts.

Bits 4 through 1 are automatically cleared when this register is read.

Bit 0, when set, indicates that characters are waiting in the receive buffer register (register 0) or the receive FIFO. It remains set until all received data is read.

Register 6: modem status register (read)

7	6	5	4	3	2	1	0
DCD carrier detect	RI ring indicator	DSR data set rd	CTS clear to send	delta DCD	delta RI	delta DSR	delta CTS

Modem Control Register Bit Definitions

Just as the UART can control the modem control signals, it can also read and report on the status of similar signals that are controlled by the attached device.

The modem status register actually provides two types of information. The most significant 4 bits show the current state of the four modem signals. The least significant 4 bits indicate which of the signals have changed state since the last time the register was read.

All these signals assume that a modem is connected to the serial port by a cable that contains all of the proper connections. Some non-modem devices also use these signals to provide hardware flow control or other device specific control functions.

Bit 7, data carrier detect (DCD), means that the local modem has established a connection to a remote modem. This bit remains set for as long as this connection is valid.

Bit 6, ring indicator (RI), is set whenever the phone is ringing (i.e., a call is coming in and needs to be answered).

Bit 5, data set ready (DSR), is generally set whenever a modem is attached and turned on. This assumes that the modem is configured to provide the DSR signal.

Bit 4, clear to send (CTS), is generally set whenever an attached modem is ready to receive data. This assumes that the modem is configured to provide the CTS signal.

Bits 3 through 0 are set whenever the corresponding modem signal changes. These bits are automatically cleared when this register is read.

For consistency, bit 2 is called the delta RI bit, but its proper name is the trailing edge ring indicator. It is set, and generates an interrupt, on the first ring from an incoming call. This is the most reliable way of checking for an incoming call.

Flow control

Flow control refers to the ability of either end of a communications link to control the rate of data it is receiving. Flow control is required when different parts of a communication link have different maximum speeds for handling data.

Consider the example of a PC that is receiving data, at a very high speed, from an attached device (e.g., data collection equipment). Assume this data is arriving so fast that the PC doesn't have time to process and store it all. This situation, continuing unchecked, would eventually overflow the PC's input buffer and data would be lost.

The solution is for the PC to tell the other end of the link to temporarily stop sending data. Once the PC is caught up, it tells the other end of the link to resume sending data again. In lengthy transactions, this stop/start process might be repeated many times. This process is called flow control.

You can look at flow control from two perspectives: receive flow control and transmit flow control. Receive flow control is the ability to tell the other side of a communication link to stop sending data to you. Transmit flow control is the ability to honor a request from the other end of a communication link to stop sending data to it. In order for your program to fully implement flow control, it must be able to do both.

Flow control comes in two varieties: hardware flow control and software flow control. Hardware flow control relies on signal lines within the serial cable to stop and start the flow. Software flow control relies on special characters in the data stream.

It is the Windows communications driver that imposes or honors flow control requests. APAX routines request that the driver enable, disable, or modify flow control. After the communication driver's flow control feature is enabled, it starts and stops the data flow automatically, as needed.

The following two subsections discuss flow control between a PC and whatever is directly attached to the PC's serial port. This is rather straightforward when the port is attached to an instrument or another computer. However, when it is connected to a modem, which is then connected via a phone link to a remote modem and PC, other issues arise. One such issue is the difference between the local PC to local modem flow control, and the local PC to remote PC flow control. These issues are covered in "Modem flow control" on page 36.

Hardware flow control

Hardware flow control (sometimes called hardware handshaking) is implemented using control signal lines in the serial cable. The name and meaning of these signals comes from the RS-232 specification. Since the RS-232 specification describes a connection between a terminal and a modem, these hardware flow control signals are properly called modem control signals and modem status signals.

Many other serial devices—printers, plotters, lab instruments, and so on—also support these modem signals for hardware flow control. Unfortunately, some manufacturers that claim to support the RS-232 standard actually treat these signals somewhat differently.

Nevertheless, much common ground does exist, particularly among modems. The type of automatic hardware flow control used by Windows should work with any popular modem on the market. However, when connecting to instruments, lab equipment, printers, or other computers, you might find slight variations in their interpretation of hardware flow control. Flow control options are provided to help you cope with these situations.

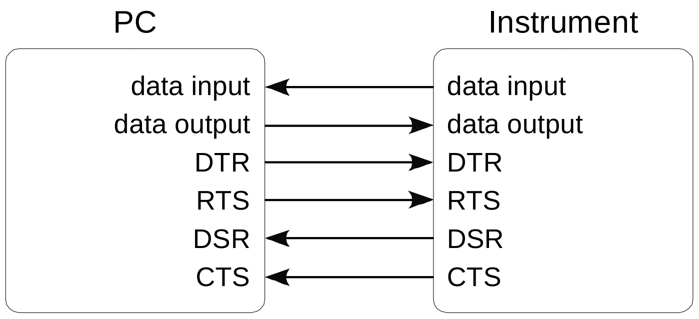
APAX hardware flow control is completely automatic. Once you turn it on, the Windows communication driver manages the modem control output signals for receive flow control and honors the modem status input signals for transmit flow control.

Standard hardware flow control requires that the modem should raise the CTS signal before the PC will transmit characters, and that the PC should lower the RTS signal when its input buffer fills while receiving. Once the PC has drained the input buffer it raises the RTS signal again. Variations on this flow control scheme exist (using different control lines, lowering signals instead of raising them) but they can all be handled using the same concepts.

Flow control happens automatically. The application can continue to send and receive characters without regard for flow control. The only issue you might need to be concerned about is how to determine if sufficient output buffer space is available before sending data. That is, you must handle the possibility that characters won't be transmitted immediately because they are blocked by flow control. You can easily check for available space by reading the OutBuffFree property of the APAXPort control.

This covers everything you need to know to use hardware flow control with APAX. The discussion continues with a more detailed look at hardware flow control. If you're curious, you might want to read on.

Let's look at the case of a PC that is sending commands to a laboratory instrument and receiving large amounts of data back from it.



The lines between the PC and the instrument represent some of the physical lines within the cable connecting these two devices. The meaning of the lines marked data input and data output should be obvious (that's where the data flows). The names of the other signals are straight from the RS-232 specification. The arrows in the diagram indicate whether a signal is an output from the PC or an input to the PC. A quick glance shows that DTR and RTS are outputs and DSR and CTS are inputs. The following is an explanation of the acronyms used:

- DTR - data terminal ready
- RTS - request to send
- DSR - data set ready ("data set" is another term for modem)
- CTS - clear to send

These signals have two states: on and off. (You might also see these two states referred to as high and low, raised and lowered, or asserted and de-asserted.)

DTR is commonly turned on by your application to indicate that your program is up and running but not necessarily ready to receive data. RTS is turned on by your program when it is ready to receive data.

DSR is an input signal from the attached device that, when on, means it is correctly attached and is turned on, but not necessarily ready to receive data. CTS is an input signal from the attached device that, when on, means it is ready to receive data.

DTR and DSR aren't usually used in flow control (although the standard Windows communication driver and APAX allow it). Instead, DTR is usually turned on when you open a port just to notify the attached device that the port is now open. Likewise, the attached device usually turns on DSR as soon as you power it on.

The RTS and CTS signals are the ones commonly used to provide hardware flow control. These signals are set and monitored automatically by the communication driver. The driver lowers RTS when its input buffer fills to the threshold you specified and it raises RTS again when your program has emptied the input buffer below the resume threshold. While transmitting data, the communication driver honors the setting of the CTS input signal and won't transmit unless the signal is high. Whenever the CTS signal switches from low to high, the communication driver resumes transmitting any data that is waiting in its output buffer.

Software flow control

Software flow control (sometimes called XOn/XOff flow control) is implemented by creating a "stop" character and a "start" character. The most commonly used characters are XOff (ASCII 19) and XOn (ASCII 17). When the communication driver receives an XOff character, it stops sending data to the remote. When it receives an XOn character, it resumes sending any data that is waiting in its output buffer.

Conversely, if the input buffer rises above the specified threshold, the communication driver sends an XOff to stop the remote from sending data. Once the input buffer is drained sufficiently, the driver sends XOn to request the remote to start transmitting again.

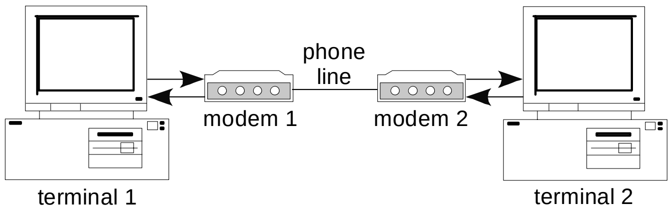
As with hardware flow control, all of this is handled automatically by the communication driver. You only need to enable it.

Buffer margins are built into APAX to account for propagation delays. This situation occurs because the remote device may continue to send characters for a brief period after the XOff is sent. The same issue applies to the resuming the data communications. To keep things going as fast as possible, we don't want to completely empty the input buffer before sending the XOn. Since the remote device might take a while to respond to the XOn, it should be sent before the input buffer is completely empty. Again, this functionality is automatic and built into the APAXPort control.

APAX also provides support for one way software flow control. The two types of one way flow control are transmit flow control and receive flow control. With transmit flow control, data transmission is halted when an XOff is received and resumed when an XOn is received. With receive flow control, an XOff is sent when the input buffer hits the upper internal buffer margin, and an XOn is sent when the input buffer drops below the lower internal buffer margin.

Modem flow control

This discussion focuses on how flow control relates to various modem configurations. The following diagram is used to discuss modem flow control.



Terminal1 and Terminal2 are PCs. Terminal1 is local; Terminal2 is at a distant location. Modem1 is the local modem sitting next to the local PC; Modem2 is the remote modem sitting next to the remote PC. In terms of the RS-232 naming conventions, the terminals are data terminal equipment (DTEs) and the modems are data communications equipment (DCEs).

Let's take the simplest case first. Assume that the two modems are both low-speed (2400 bps), non-MNP, non-V.42 modems. Such modems typically do not support any type of flow control between the terminal and the modem. The link between two such modems is not being managed for error control or data compression. When a serial stream of bits leaves Terminal1, it travels through Modem1, across the phone line into Modem2, and into Terminal2. So logically, this is a straight line connection and there aren't any stopping points where flow control might be needed.

This is not to say that you'd never have flow control in such situations. Certainly you can't have hardware flow control, since there's no direct connection between the modem control signals from Terminal1 and Terminal2. But you can, and sometimes must, have software flow control. The software flow control acts as though the modems aren't part of the connection at all and only controls flow between Terminal1 and Terminal2 (known as a pass-through connection).

Let's say that Terminal1 is sending lots of text to the person sitting in front of Terminal2. Once the Terminal2 screen fills, that person will want to pause the flow of data while reading the text. If both terminals are honoring software flow control, then pressing Ctrl-S (which is the XOff character) on the keyboard is sufficient. This XOff gets transmitted without interference all the way to Terminal1, which then stops transmitting. When the person in front of Terminal2 is ready for additional text, Ctrl-Q (which is the XOn character) is pressed. This XOn gets transmitted to Terminal1, which resumes sending text. This is software flow control.

Now consider the case of managed modem links. "Managed modem link" means that the link between Terminal1 and Terminal2 is no longer a pass-through connection. Instead, Modem1 collects a group of bytes from Terminal1, packages them into a block, and sends that block, complete with error control information, to Modem2. If the block is received without errors, Modem2 acknowledges receipt of the block (i.e., it tells Modem1 that it got the block OK) and passes the stream of bytes on to Terminal2. If there are errors, Modem2 doesn't pass the data on to Terminal2, but instead asks Terminal1 to retransmit the block.

Before this can work, both modems must agree to the same management scheme. Currently, the modem industry supports two standards for managing this link: V.42 LAPM (Link Access Procedure for Modems, a standard supported by CCITT) and MNP (for Microcom Network Protocol). Describing either of these standards is far beyond the scope of this manual. If your modem supports one of these standards, then your modem manual will provide the necessary information.

Each of these standards provides a managed link between the two modems. This provides opportunities for error detection and correction, data compression, and end-to-end hardware or software flow control.

The following example clarifies what is meant by end-to-end hardware flow control. Suppose that Terminal1 is transmitting data at a high rate over the link to Terminal2. Terminal2 can't process the data fast enough, so it drops its RTS line, forcing Modem2 to stop passing data. This also causes Modem2 to stop accepting data from Modem1 (which forces Modem1 to stop accepting data from Terminal1). Since Modem1 cannot accept data any more, it lowers its CTS signal, telling Terminal1 to stop transmitting its data.

With unmanaged links, however, the software flow control is between the terminal and modem instead of between the two terminals. But, as explained earlier, a managed link's flow control between the terminal and modem is just as good as terminal-to-terminal in an unmanaged link. Managed links generally require either software or hardware flow control. This is true even if the link is operating at low speeds (2400 bps) since you never know when modem-to-modem retransmissions might occur, potentially causing the transmitting terminal to overflow its modem's buffer.

Serial Communication Under Windows

Windows communications drivers handle the hardware details. Applications rely on a set of Windows functions to configure and manage the serial port. These functions serve as a bridge between the communications driver and the applications program.

First, Windows communications routines send serial port control and I/O requests to a standard communications driver named VCOMM. However, VCOMM does not directly control the serial port, but relies on port drivers to do that. Windows provides a built-in port driver for standard serial port hardware.

Second, Windows provides support for multiple concurrent threads. Because of this, a program can simply create a thread that sleeps while it waits for communications events.

APAX takes advantage of threads to minimize CPU usage, while at the same time minimizing response time to communications events. When an APAXPort control opens the associated port, it creates three threads: a communications thread; a timer/dispatcher thread to handle incoming data; and an output thread to send data in the background.

The communications thread uses the Windows communications functions SetCommMask and WaitCommEvent to sleep while waiting for communications events (e.g., incoming data or changes in line or modem status). When a communications event occurs, Windows wakes up the communications thread, which then notifies the timer/dispatcher thread that a communications event occurred and must be processed.

As its name suggests, the timer/dispatcher thread has two roles: timing and dispatching. It sleeps until its sleep time elapses or a communications event occurs. Then the dispatcher processes the event, checking for any events that are due. If an event is required, the dispatcher formats and generates the corresponding event.

The dispatcher is working within its own thread, not within your application’s thread. To avoid synchronization problems, it always generates events via the Windows message dispatcher (SendMessage). SendMessage assures that the thread that created the window is active before the message is delivered. Therefore all APAX events are processed on the expected thread (the thread that created the component in question).

If a needed application thread, say a thread that created a protocol, is blocked (waiting for a semaphore or other event) when an OnProtocolStatus event is generated, the OnProtocolStatus event must wait until the protocol thread becomes unblocked. Because the OnProtocolStatus event is generated from the dispatcher thread, the dispatcher itself becomes blocked until the OnProtocolStatus can be delivered. This presents the possibility of the deadlock where neither side can proceed and the application appears to be hung.

To avoid such deadlocks, trigger events are always generated conditionally. If the recipient thread doesn't respond to the SendMessage within a short period of time (usually 3 seconds), the SendMessage attempt is abandoned and the deadlock is prevented. The downside to this protection is that the event in question is never seen, but this is more desirable than deadlock.

The best way to avoid missed events is to assure that threads that are expecting events are never blocked for more than a fraction of a second. If you must block for longer periods, you should create alternate, unblocked threads for handling the expected serial port, protocol, fax or other communications events.

Performance issues

Communications applications gain a lot from the Windows architecture: a common API for communications tasks, multitasking support, and device independence. This ease of use is unfortunately offset by a loss in performance. The Windows architecture forces a huge amount of overhead onto communications applications, with the result that the highest achievable baud rate and throughput will always be reduced.

Because there are so many variables that come into play (machine speed, device settings, ill-behaved Windows applications, etc.) it is impossible to predict how well an application will perform.

There are several ways you can optimize performance:

- Use as low a baud rate as possible. For example, don't use a baud rate of 38.4K baud when the data rate is only a few hundred bytes per second.
- While communicating, reduce the number of active windows that are performing background processes. If you are redistributing your application to other users, warn them of lower data throughput when multiple background windows are active during communications.
- Use a 16550 UART, which has a 16 byte FIFO buffer.
- In those cases where incoming data is extremely critical, use an intelligent serial port board, which off-loads serial interrupt processing tasks from the CPU.

Event Management

Windows is a message-driven environment and APAX is designed to fit into the Windows message system. APAX provides a communication dispatcher that is activated regularly to process data received by the Windows communication driver. The dispatcher transfers the received data from the Windows input buffer to its own dispatch buffer and, when appropriate, generates trigger events that can be processed by your application.

The standard dispatcher is started whenever a COM port is opened. Each opened port starts three new threads that together provide the dispatching functions. When the port is closed the three threads are released. A Winsock specific dispatcher is used when you are operating in Winsock mode.

Once the dispatcher gains control, it copies received data from the Windows buffers to its buffer and updates internal fields with new line or modem status information. If any of these actions require trigger events, the dispatcher sends a message to the appropriate component and the component generates the corresponding event.

APAX uses the term “trigger” for any event that can cause the dispatcher to generate a message. Triggers are associated with a particular APAXPort control and their internal handling is done transparently to the APAX user.

Chapter 4: Overviews and Troubleshooting Sessions

Communications programming is such an intricate undertaking—and APAX so comprehensive a library—that many developers find gaining a broad view of the requirements and possible solutions for any task to be the largest obstacle they face.

This chapter provides overviews of two issues that have been raised frequently by APAX users: picking the right modem for your next project and issues regarding voice modems and TAPI voice support. A third overview offers a collection of tips and techniques for debugging Windows communications programs and diagnosing common hardware difficulties.

Two troubleshooting topics give some common questions and answers that appear regularly in the TurboPower newsgroups regarding communications sessions and file transfers.

Overview: Choosing a Modem

This topic covers recommendations for picking the right modem for your next project.

"What modem should I buy?" is a very common question. Those of you asking the question probably had a hard time getting a straight answer. Unfortunately, this particular question is difficult to answer with any reasonable degree of accuracy. This section explains why, along with some general recommendations for buying your next modem.

It's a jungle out there!

The modem market is extremely dynamic and competitive. Any modem may cease to exist tomorrow. (Indeed, the manufacturer of the modem may have disappeared.) Those of you keeping an eye on technology news probably noticed that some of the "big boys" in the modem business (such as Motorola and Hayes) closed up shop over the last year or two.

This level of competition has driven margins down to a bare minimum. Unfortunately, the competition seems to be primarily based on price and feature set. Competition is good, but not when it gets to the point that corners are cut. We'll mention some specifics about these "cut corners" a bit later.

You find yourself thinking, "This should be easier..."

The more you learn about modems, their quirks, and how to deal with them, the more you realize how tough the situation is. There are variations in the hardware and firmware used in the modems, even modems with the same exact make and model number. These variations may not be public knowledge: the details of such variations are often kept within the walls of the modem maker. With modems, like any other piece of hardware, there are variations in quality due to an imperfect manufacturing process. Granted, the variations are usually kept to a minimum when you deal with the more respected manufacturers, but they exist nonetheless. Even top ranked manufacturers can have a bad day (or batch).

Is the situation hopeless? Well, no. To be honest, most modems work reasonably well most of the time, so there's no need to panic. Arming yourself with a bit of knowledge before venturing out to the nearest computer store is worth the effort though; it'll increase your chances of getting something with a reasonable level of reliability.

Quick Lesson: Modems 101

What exactly is a modem, anyway? Well, even that is hard to answer these days as it's become a bit of a marketing term. As an example, Digital Subscriber Line (DSL) service in the United States labels the device used to connect the phone line to the computer a modem, but it's really more of a network router. Strictly speaking, a modem is a MOdulator/DEModulator. It converts digital signals from the computer's serial port to modulated analog signals for the phone line (and vice-versa).

The term "modulated" or "modulation" refers to the technique of combining information (the data from the serial port) with a carrier wave that travels well through the target medium (the phone line). The carrier wave for a standard phone line is usually restricted to audio frequencies, since that is what the phone line (and associated equipment) is designed to handle. Modulation also makes things like radio and television possible, the carrier wave there being very high and ultra high frequencies that can travel through air/space.

Besides the basic modulation/demodulation, a modem has a lot of other jobs to do. It needs to be able to properly connect with a modem on the other end of the line, negotiating things like the type of carrier to use as well as the type of error correction and data compression to use. Once a good connection is established and data starts flowing, the modem dynamically encodes, decodes, compresses, decompresses, modulates, and demodulates the data—all while checking the data for errors (requesting a resend of any data that has errors). That's for simple modem communications. Things like faxing and voice communications add more factors to the equation.

The good news is most of this work is done without your knowledge (or even APAX's knowledge). Just keep in mind that it's a complex process and things can easily go wrong.

So what should I buy?

Here are some general guidelines. Remember, these are only general guidelines. TurboPower has several modems that break one or more of these guidelines and still work fine. We also have a couple modems that follow all the rules and are problematic.

Avoid Winmodems and RPI modems, otherwise known as software modems

These modems offload some of the processing that the modem has to perform to the host computer. They use software drivers to handle things like compression and error correction that are normally handled by the hardware/firmware in the modem. To be fair, these modems have a couple of advantages: the drivers are easy to update, and the overall cost of the modem is lowered (the whole concept of a software modem probably came about as a result of the competition in the modem market).

However, software modems have several disadvantages. For example:

- The host computer is forced to donate resources in support of the communications session (not only the CPU, but also memory, data bus, power, and so on).
- Shifting these duties to software results in an overall loss of efficiency (custom hardware is better suited to handle this type of processing).

- Most software modems will replace the standard Windows serial port drivers. This could affect all serial communications on the system, since all access to the serial ports will go through the replacement drivers. For example, some replacement drivers only support 8 data bits, no parity, and 1 stop bit, which would cause garbled characters if a connection is made by any serial port on the system using any other serial port settings.
- The modems are tied to the operating system. They currently do not work in alternate operating systems such as Linux due to the lack of drivers.
- The software drivers that support these modems are proprietary, and cannot be used directly without license. For this reason, APAX does not have access to the error correction and compression features of these modems unless the modem is accessed through TAPI.

How can you tell if a modem is a software modem? Usually, you'll see "Winmodem" or "RPI" somewhere on the box. Another indicator is if Windows is the only supported operating system. A fairly comprehensive database of modems that identifies software modems can be found at http://www.o2.net/~gromitkc/winmodem.html. This Web site also has additional information about modems, as well as useful links to other modem Web sites.

Use external modems

This is simply the best way to ensure you get a modem with all its brains intact. It does not seem to be practical to produce an external modem that uses drivers to handle things like error correction and compression. An external modem is also easier to monitor and troubleshoot (most have status indicator lights on the front panel). External modems have their own power supply, so there is no additional load on your computer's power supply. External modems are often easier to install and set up, since you don't have to open the computer case and deal with system settings such as IRQs. Admittedly, this situation has improved somewhat with innovations like Plug and Play, but that's not available with all operating systems. USB modems are sometimes Winmodems also. We have found one USB external modem that supports RS232 and USB. The RS232 connection is not a Winmodem, but the USB connection is.

Don't "chase the latest technology"

Modem makers frequently race to hit the market first with a new feature in order to gain market share. It often takes a little while to make new features reliable though, so the first few batches of modems sporting a brand new feature often aren't as reliable as subsequent batches will be.

Get a modem with the features you need, and no more

In other words, if you need a modem strictly for faxing, why get a voice modem? This is a cost saving recommendation for the most part, but there's a certain "less can go wrong" issue also.

Get a modem that supports more than one fax class if you'll be faxing

Having options in this area is a good thing. If one of the available standards doesn't work for a given situation, another option often will.

Apply common sense

Use well-known brands. It's tough to know for sure if the maker of your modem will still be around in a year or two in the event you need a new driver or support for your modem. The odds seem to be a bit better if you stick to an established brand.

Buy from a store with a reasonable return policy

This should allow you to test the modem in the environment and with the application you'll be using. If you need to buy many modems for a project, buy one or two first and test them thoroughly with the code you'll be using before committing the money for all of them. If you're going to be buying several hundred modems, make sure you're getting modems that will work well in your situation.

Overview: TAPI Voice Support

This topic offers tips and techniques for using voice modems and TAPI voice support.

- TAPI Voice support is only available with the Unimodem/V TSP (TAPI Service Provider) and the Unimodem/5 TSP. Unimodem/V is installed by default in Windows 95OSR2, Windows 98 and Windows ME. Unimodem/5 is installed by default in Windows 2000. A voice modem that is TAPI compliant and accepts the AT+V or AT#V command set is also required. Unimodem/V is available for download from Microsoft's web site for Windows 95. A list of voice modems that Microsoft has tested is included in the Unimodem/V and Unimodem/5 installation's Readme. TAPI Voice support is not available in Windows NT 4.0 unless the modem manufacturer provides a voice-enabled TSP.
- Windows 95 OSR1 does not come with Unimodem/V, but OSR2 does. You can verify whether you have Unimodem/V by checking the version number of the Windows\System\Unimdm.tsp. It will be 4.1 or greater for Unimodem/V.
- Windows NT does not support the Unimodem/V Driver. However, some manufacturers may supply their own TAPI compliant driver for NT support. Windows 2000 includes Unimodem/5, which is fully voice capable. Unimodem/V and Unimodem/5 are not the same thing. The "V" in Unimodem/V stands for "Voice", the "5" in Unimodem/5 is for NT 5 (Windows 2000).
- Be aware that manufacturers (hardware and TSPs) may include or exclude TAPI functionality at their discretion. Some devices may or may not support certain functionality.
- Implement organized and optimized state machines in the OnTapiDTMF and/or OnTapiWaveNotify event handlers if you are implementing DTMF for automated Voice and wave recording prompting.
- Both the telephone line and the modem must support Caller ID for an application to support it. Caller ID formats vary around the world: make sure your modem supports the format used by your telephone company.
- Wave files used in TAPI applications must be of a specific format. Simply opening the Sound Recorder and recording will not provide compatible WAVE files. PCM 8,000 Hz, 16 Bit, Mono is usually a valid format, but you will have better sound quality if you use a format directly supported by your modem. (See your modem's documentation for the native formats it supports.) You can convert other WAVE files by running Sound Recorder, opening an existing WAVE file, selecting the Properties Dialog, choosing the Convert Now... button and changing the settings. It is a good idea to go ahead and create a new TAPI quality selection for the wave type list.

- Trim your wave prompts. In sound recorder (or any wave editor), trim silence from the beginning and the end of the wave file. Otherwise, users may get impatient with your application.
- In general, it is good practice to set InterruptWave to True. This allows callers to break the current wave prompt being played and proceed with the call. However, if there is a voice prompt that must be listened to completely before the caller should proceed, set InterruptWave to False for that single prompt. Always reset InterruptWave back to True when possible.
- The sound quality provided by a voice modem varies greatly between manufacturers, models, and sometimes the modem batch. The best sound quality and control will come with a dedicated voice board such as those provided by Dialogic, BrookTrout, MediaPhonics, etc. Regular, off the shelf voice modems are primarily data modems with the voice command set added on. Many voice modems do not have facilities for volume control or audio stream normalization. The dedicated voice boards cost more, but they are worth it for higher quality voice processing.
- Not all voice modems or TSPs provide accurate call progress detection. This means that your modem may not be able to tell when the remote party actually answers the phone when you call them. Unimodem/V and Unimodem/5 bypass call progress detection to a large extent when an outbound call is made, and they signal a connection shortly after dialing, regardless of whether or not the called party has answered the phone. Inbound calls are signaled more reliably, since the TSP knows when it answered the call. Likewise, Unimodem/V and Unimodem/5 usually cannot tell when the remote party hangs up the phone. Dedicated voice boards usually provide much more reliable call progress detection.
- APAX negotiates for TAPI version 1.4, which is backwards compatible in all later TAPI versions as of this writing. APAX implements a few TAPI 2.0 functions, such as retrieving the port number associated with the device. APAX also implements the TAPI Line device; TAPI Phone devices are not supported.
- Unimodem does not provide any voice modem capabilities. Unimodem/V, Unimodem/5, and some third party TAPI service providers support voice modem capabilities.

TSP	Win 95 OSR1	Win 95 OSR2	Win 98	Win ME	NT 4	W2K
Unimodem	X				X	
Unimodem/V	Download	X	X	X		
Unimodem/5						X

Overview: Debugging Windows Communications Programs and Communications Hardware

In this section, we provide a collection of tips and techniques for debugging Windows communications programs and diagnosing common hardware difficulties. Some of these suggestions are very simple and you may well already use them. Others, however, are specific to communications programs and might cover issues you haven't previously faced.

First, always make sure that your hardware is set up correctly (check connections, cabling, switches, etc.). The best way to verify this is to start with a known, reliable communications program. If your known, reliable communications program doesn't work, you know that there's something wrong with the serial port, the cable, the device you are connected to, or the line parameters. In this case, try the techniques listed below for diagnosing hardware problems.

Using the debugger

If you have used some older communications libraries, you may recall cautions about using debuggers with communications programs. For example, some DOS debuggers tend to interfere with communications interrupt service routines and cause loss of incoming data and prevent outgoing data from being transmitted.

Under Windows you can ignore those cautions. The communications interrupt service routine is in the Windows device driver and isn't blocked by Windows debuggers. While in a debugger, you can freely step into or over any communications routine without harming either the input or output data flow.

Be aware, however, that it is still possible for incoming data to "stack up" in the communications driver. While you are leisurely stepping through a routine in the debugger, your application won't be processing timer or communications notification messages. If these messages aren't processed, data cannot be removed from the communications driver. If data is arriving in an uninterrupted stream, the driver's input buffer will eventually fill to capacity. If flow control is in place, the driver will impose flow control, otherwise data will certainly be lost.

Using APAX dispatch logging

APAX provides an auditing tool called dispatch logging, which works at a very low level. Dispatch logging provides an exact chronology (with millisecond timestamps) of all events processed by the internal dispatcher, as well as state changes in the APAXPort control. It's handy for figuring out problems with hardware flow control and other control signal situations (e.g., "Why isn't my program answering a ringing phone?"). See "Dispatch logging" on page 67 for more information on this facility.

Getting technical support

TurboPower Software Company offers a variety of technical support options. For details, please see the “TurboPower Product Support” card enclosed in the original package or go to www.turbopower.com/support.

Technical support is always challenging and throwing communications problems into the equation makes the task even more difficult. For that reason, you should do several things before asking for support. These may seem like trivial things (and some of them are indeed trivial) but getting them out of the way ahead of time could save you some effort.

- First and foremost, if you’re writing an application and “not getting anything” please try the supplied, unmodified, demonstration programs. This is a polite way of saying “make sure it’s plugged in” before deciding your application doesn’t work. Whether you’re connecting to a piece of data collection equipment, plugging in a new plotter, or just trying to send commands to a modem, start from a known, reliable program to prove to yourself that the device is hooked up, properly configured, and connected with a working cable.
- If you’ve proven that all is well with your hardware but your program still isn’t behaving properly, be sure to use APAX’s built-in dispatch logging to try to find the problem.
- Finally, any APAX routine that can fail generates an exception or returns an error code if an error occurs. A fair percentage of technical support requests are the result of an application program continuing to use an object after an error has been reported. To avoid this problem in your programs, be sure to follow up on exceptions and check all error codes.
- If you tried a “known good program” and applied the built-in logging tool and you’re still having a problem figuring out what’s going on, then contact us through one of our support options and we’ll do our best to help you find a solution. Depending on the problem you’re having, we may ask such questions as “What did the example programs do in that situation?” or “Did you try TermDemo?” or “What error code was returned?”. If you have answers to such questions handy, we’ll probably be able to zero in on the problem much faster. We might also need to discuss your log file. Please be sure to have such files available when the problem warrants it.

Common problems

Here's a brief discussion of some of the common problems that popped up during development and testing of APAX. They are organized in a question and answer format.

Nothing works, not even the supplied test programs. What's wrong?

It may be a hardware or cabling problem that you'll need to figure out before you can go any further. Common problems include disconnected, improperly configured, or outright bad modems. Other causes include two or more serial ports using the same system resource(s), or another device (e.g.: a mouse or network card) using a system resource usually reserved for a serial port.

Despite the increasing power and sophistication of desktop computer systems, serial communications remains a remarkably primitive and awkward set of standards and practices that leaves a lot of room for problems to arise.

The modem isn't working. What do I do?

Modems are peculiarly delicate devices; they can be easily damaged by physical events, static discharge, or "spike" currents over the phone line. They even sometimes fail right out of the box.

In general:

- Make sure that the phone line is attached and is live (check with an actual phone device to make sure you get a dial tone).
- Make sure the phone cord is going into the correct modem jack. Most modems have two: one for the "line" into which you should plug the line that is going to the wall jack, and one for the "phone" which allows you to attach a phone or other device beyond the modem.

If you're using an external modem:

- Make sure the modem is plugged in and turned on (you should see lights on the front panel).
- Make sure that the cable between the computer and the modem is attached to the correct port on the computer. Some SCSI interface cards have a port that looks exactly like a 25-Pin serial port. Also make sure you get a "straight through" cable for this purpose. A "null modem" cable may sound like what you need, but is actually used for a different purpose.

If you're using an internal modem:

- Make sure the modem is seated properly in the card slot and that you have the latest drivers for the modem installed.

- Check for resource conflicts. See the sections below on serial port setup for information on possible resource conflicts.

If none of these seem to help:

- Make sure the phone cord is good (test with a phone using that cord).
- If possible, try a different modem device in the same situation to eliminate a bad modem as the problem.
- Try the serial port checks listed next.

The serial ports aren't responding. What do I do?

Serial communications on a PC operates through serial "ports". These originally were physical wires that had a particular organization and operation. Now your setup may include "virtual" serial ports that exist only in software; these allow communicating with many kinds of devices as if they were serial devices (such as USB modems).

For communication through a serial port to occur, the port must be configured correctly. Serial ports require certain resources from the computer in which they are installed. Generally this will consist of a interrupt request (IRQ) number, and a base address.

On IBM PC Compatibles, the traditional resource assignments for the first four serial ports (COM1-COM4) are:

Port	IRQ	Base Address (in hexadecimal)
COM1	4	03F8
COM2	3	03E8
COM3	4	02F8
COM4	3	02E8

Note that traditionally COM1 and COM3 share an IRQ, as do COM2 and COM4. This is a hold over from the early days of the IBM PC when there were only 8 IRQs available. The nature of software at the time made it unlikely that more than one or two ports would be accessed simultaneously.

On modern systems it is generally desirable and often necessary to set COM3 and COM4 to different IRQs than those listed in order to prevent conflicts (it's generally best to leave COM1 and COM2 where they are). IRQ 5 (traditional for LPT2) is often available for one of them.

Internal modems generally present themselves as COM ports to the computer and similarly require their own unique settings.

Some specialized serial port hardware (multi-port boards) permit IRQ sharing among a number of ports. These will typically have specialized driver software to manage the multiple ports.

My modem/serial port card says it's "Plug and Play". What does that mean?

Plug and Play is a set of standards that allows computer systems to query devices installed in the system and determine what they are and their capabilities. Plug and Play devices may include items built onto the system's main circuit board, or may include add-on cards of various kinds.

Many new computer systems include built-in support for Plug and Play hardware. The Windows 95 and 98 operating systems include extensive system level support for identifying hardware. Plug and Play under WindowsNT is somewhat sketchy, but Windows 2000 is on par with Windows 9x/ME.

Some add-on cards for serial ports are Plug and Play, as are many internal modems. Also most modern computer main boards have two serial ports on-board which are often handled by Plug and Play.

For one or two ports these should generally work as-is, but a common requirement to get multiple ports operating correctly is to disable Plug and Play for these ports and set their resources manually.

How do I set up those "on-board" serial ports?

If your system's main circuit board (motherboard) features on-board serial ports, there are generally some settings for these available in the BIOS Setup program.

The BIOS Setup program is usually accessible via a special keystroke at system start-up (often pressing by Delete or a function key; look for a message indicating how and when your system boots).

Accessing the Serial Port settings varies widely among BIOS models, so check your main board or computer manual for where these might be located.

Often, you can set the on-board ports for some kind of automatic mode, which means the IRQ and address range are set dynamically by Plug and Play when the system starts.

Ports set up in this way will generally end up with standard IRQ and base address assignments. This is not guaranteed, however, and some software has problems dealing with ports with peculiar settings.

So what do I do if something isn't working?

If you're having problems with serial ports with hardware, manually set IRQ and base address values (often set by jumpers on an expansion card). Make sure that each port's settings are unique.

If you're having problems with serial ports with Plug and Play settings, try setting the on-board (or any other Plug and Play) ports to specific IRQs and base addresses rather than allowing them to be determined dynamically. Using the traditional resource assignments mentioned above is usually the best approach.

If the ports are on an expansion card, the same caveat applies as for internal modems: make sure that the card is properly seated in the slot. If the ports are on the main board, there are typically small cables that run from the system board to the physical port outlets on the back of the computer. The connectors used to attach the cables to the main board are often small and may come loose. Make sure they are oriented correctly and well seated on the correct pins. Some port cards also use similar short cables and the same applies to them.

So how do I fix the Windows settings?

In 32-bit Windows, all hardware is managed through the Device Manager. This is accessible in the System applet in Control Panel or by right-clicking on My Computer and selecting Properties, then clicking on the Device Manager tab.

Look for entries under Ports (COM & LPT). Clicking on one of the ports listed there and selecting Properties will show an informational dialog. The Resources tab has the settings for the IRQ and Address Range. You can change the IRQ and base address settings here. Setting them explicitly can sometimes help with Plug and Play conflicts.

What about TAPI?

TAPI (Telephony Application Programming Interface) is a formalized set of routines to allow programs to make use of various telephony hardware.

Windows 95 introduced a generic implementation of TAPI that all programs could access, enhanced versions were included in NT 4.0 and later operating systems.

If you're having trouble using TAPI to access or operate a particular device:

- Make sure the device actually appears in the list of TAPI devices (in the Modems applet in Control Panel). If it doesn't, it probably needs to have drivers installed. If you know you installed the drivers already and the device has previously worked, it may not be "visible" to the operating system for some reason, which is usually a hardware issue. Check the sections "The modem isn't working. What do I do?" on page 52 and "The serial ports aren't responding. What do I do?" on page 53 for diagnostics.
- If you're using Windows 95 you may need to obtain the updated TAPI (UNIMODEM/V) software from Microsoft. Later Windows versions should already have installed newer drivers (though it's still sometimes prudent to check, some device may have installed older drivers over the new ones, and Microsoft may have come out with something new after this manual went to print).

- If you have device names that are not unique within the first 20 characters, early versions of TAPI sometimes gets confused in device selection. Unfortunately, the only way to change the TAPI assigned names for these devices is to edit the registry or the INF file that is used for installing the modem. The best solution is to install the updated TAPI drivers that don't have this problem.
- Make sure you have installed the latest drivers (INF files) for your modem. Check the modem manufacturer's Web site for updated drivers.

What about Winmodems?

A current trend in modem technology is to simplify the physical hardware of the modem device and supply some portion of its functionality in the form of software drivers for the modem. Such devices are generically referred to as "Software Modems". Because the vast majority of them are designed to work with some version of the Microsoft Windows Operating System, are also frequently called "WinModems".

This approach has made some sophisticated modem technology much cheaper to implement, but has also created a number of headaches.

First, these software drivers generally expose a TAPI interface, and so these modems often must be initialized via TAPI in order to work correctly, which can cause problems with older or otherwise TAPI naive software.

Second, the drivers for these modems are generally operating system specific; a driver for Win95 may not work on Win98, and almost certainly won't on NT (much less OS/2 or Linux). The skills necessary to write good device drivers are deep and hard won, and many drivers don't behave entirely as advertised. Also, even if a particular Win95 driver is good, it doesn't mean that the NT driver for the same modem is as good (or that the manufacturer even has one).

Often the installation software for Winmodem drivers will replace the default Windows serial drivers with ones of their own. These drivers sometimes behave unpredictably when accessing other serial hardware in the system.

If you're having trouble getting a software modem (WinModem) to work:

- Make sure that the drivers are installed correctly.
- Make sure you have the latest drivers from the manufacturer.
- Make sure to open the modem using TAPI in your program.

Why am I getting overrun errors?

A UART overrun occurs when a character is received at the serial port before the Windows communications driver has a chance to process the previous character. That is, characters are coming too fast for the driver to handle them.

There is a finite limit to the speed at which a given machine can receive data. Because of the extra layers of overhead in Windows, this limit is substantially lower than under DOS. A baud rate that worked under DOS simply may not be achievable under Windows.

A more likely cause, however, is that another Windows task is leaving interrupts off for too long. While interrupts are off, the communications driver isn't notified of incoming characters. If interrupts are left off for more than one character-time, it's very likely that you will lose characters due to UART overruns.

One known cause of long interrupts-off time is virtual machine creation and destruction. The only solution is to avoid opening or closing DOS boxes during critical communication processes.

Interrupts could also be left off by other Windows device drivers or virtual device drivers.

Why do my protocol transfers seem slow?

This usually means that your status routine is taking too much time. You shouldn't try to do any lengthy calculations, disk I/O, or any other time consuming activities in your status procedure. You can test this hypothesis quickly by trying a test run without your status procedure or with a very simple status procedure instead.

Why am I getting parity and framing errors?

Either you're operating with a different set of line parameters than the remote device, or your cable is picking up interference. Generally, the higher the baud rate you select, the more likely you are to suffer from electrical interference. If you suspect that your cable is picking up interference from other electrical sources, consider rerouting the cable run away from such sources.

My protocol transfer never gets started. What's wrong?

This could be due to any of several problems including mismatched line parameters, wrong protocol selected, or the file to transmit could not be found. Your best bet is to generate a dispatch log and see just how far the protocol was able to progress. Also, try one of the demonstration programs in the same situation to see if it works. Generally, this should provide enough information to find and correct the problem.

My Zmodem file transfer program generates lots of psBlockCheckError errors and psLongPacket errors, but other protocols work fine. What's going on?

The answer in this case is almost always lack of hardware flow control. The problem shows up in Zmodem but not other protocols because Zmodem is a streaming protocol. Data is sent in a continuous stream without pauses for acknowledgments. Flow control is required to prevent the sender from overflowing the modem or the receiver. Remember, flow control must be enabled at four places: your software, your modem, the remote software, and the remote modem. See “Flow control” on page 32 for information on flow control. Consult your modem manual for the hardware flow control enable command for your modem.

Troubleshooting a Connection Session

This topic addresses some common problems for troubleshooting a communications session.

Every communications session relies on a stable connection to perform at its best. Modern phone lines and data cables are relatively reliable and connection parameters are somewhat standardized, but there will come a time when nothing you do seems to work the way you hope.

Problems with a communications session can come in many forms and at different times in the session. The first step in troubleshooting a connection session is to make sure the application is set up correctly for the system on which it is being run. After that, try one of the example projects that illustrates what you are trying to do (or comes close). These programs are used as benchmarks for further troubleshooting. Also, use one of the communications applications installed with the operating system. HyperTerminal or Terminal can help in identifying system setup issues.

Here are some common problems and how to resolve them:

Why do I get an exception when I try to open the serial port?

To open a port, the APAXPort control tries to activate the serial port of the computer identified in the ComNumber property. If the port is not present on the system, in use by another application, not correctly configured at the system level, or the system resources are too low, an error is generated. You can trap that error and bring up the default Comport Selection Dialog by setting the ComNumber property to 0 and then opening the port again.

Why won’t my device won’t respond to commands?

If you send configuration and initialization commands to the device and it does not respond, make sure the device is turned on, the necessary device drivers are loaded, any serial cables are functional, and that you have the ComNumber property properly set. You can test the serial cable by using a different cable. Also, send the commands in upper case and make sure the device is set up to respond with verbose results instead of numerical codes.

Why does my mouse stop working when I open the port?

If other serial devices stop working when the port is opened, the most likely cause is an interrupt conflict, which you will need to resolve by reviewing the IRQs used by all devices on the system.

Why won't my device won't dial?

If you get a "No Dialtone" message, make sure the phone cord is inserted into the correct jack on the modem and the wall. Also make sure the cord is functional by using it with a phone. In some cases, services supplied by local telephone companies use an interrupted dialtone as an alert. This can cause a "No Dialtone" message as well.

Why will my device dial but not connect?

If the modems start handshaking but do not complete the connection, try placing the call again. The telephone company routes each call differently each time and you might have had a bad connection. Call a different number or modem to verify that the local setup is correct. Turn off Error Correction on your device.

Why do I get random or garbage characters after I have connected?

Make sure the Parity, StopBits, DataBits, and Baud properties match the system to which you are connecting. Verify that you are using the same type of flow control on both ends of the connection.

Why are the characters in the terminal window doubled when I enter them?

Turn off the Echo mode of the modem, or set the TerminalHalfDuplex property to False.

What do I do when I have tried everything but still nothing works?

There are a few times when changing the component properties do not seem to work. If the phone lines are verified as being good, reset the modem before using it by sending it the "Reset to factory defaults" command. For most modems, it is "AT&F"<cr>(refer to your modem manual for the specific command for your modem). After sending this command, you must wait until the command has been executed by the device before sending additional commands. If this doesn't work, look at the system environment. Remove any non-standard device drivers one-by-one until the problem is eliminated. Then add the device drivers one-by-one again until the specific driver that is causing the problem is identified. Once it is identified, contact the device manufacturer for updated drivers. The video drivers are a good place to start looking, so change the video mode to a standard Windows-supplied mode. Believe it or not, this simple change has solved everything from strange displays to eliminating errors while sending files.

Also, many Winmodem, RPI, HSP, or other software modems replace the standard serial port drivers when they are installed. Some of these replacement drivers do not support nonstandard port setting (anything other than 8 data bits, no parity, and 1 stop bit). To compound the problem, there are several replacement drivers that simply ignore attempts to change port parameters, which will result in garbled text or connection failures when non-standard setting are used.

Troubleshooting a File Transfer

This topic addresses common problems for troubleshooting a file transfer.

You have a stable connection, everything seems to be in order, but the file transfers aren't working as you'd expect.

Where do I begin?

The first step is finding out what caused the failure. To do that, look at the ErrorCode parameter of the OnProtocolFinish event. "Undefined" error codes are usually Windows API errors. Look these up in the Windows API help files installed with your compiler. Once you know what caused the failure, you can usually spot the problem easily.

Why does nothing happen when I call StartTransmit or StartReceive?

Set the ProtocolStatusDisplay property to True. This provides visual feedback on the status of the transfer. If the local transfer is truly not doing anything, then make sure the other end is set to send or receive the transfer. For example, if you are sending a file with Zmodem, the sending machine will send 'rz' followed by ASCII 13 to let the receiver know something is coming. If the receiver is not watching for this character sequence, your transfer will eventually time-out.

Why does only one OnProtocolFinish event fire when I send or receive a batch transfer?

The OnProtocolFinish event fires when the entire protocol session is complete. In a batch transfer, the protocol session ends with the transfer of the last file in the batch or upon a terminal error. The ErrorCode parameter of the OnProtocolFinish event tells you what caused the termination of the entire session. To get the information for the individual files, use the OnProtocolLog event.

Why are my transfers are slow?

You first need to determine if this is a valid problem. Each character that gets transferred takes 10 bits, so a 28,800 bps connection will result in 2,880 cps. If you think the transfers are still slow, flow control is most likely at the root of the problem. Both sides of the transfer need to implement the same form of flow control. Hardware flow control is preferable over software flow control but some systems do not support it. If you are using an external modem, make sure the cable supports hardware flow control signals. A slow transfer can also be a sign that the connection is not stable. Hang up and try the call again to see if you can get a better connection. Other conditions that would cause slow transfers are running other CPU intensive applications during the transfer, an overloaded protocol status event, and frequent disk access.

Chapter 5: Serial Ports and Logging

An application uses the APAXPort control to control serial port hardware. All serial port I/O is performed by calling methods of APAXPort control and by writing event handlers that respond to serial events.

Sending and receiving data through the serial port is obviously part of the process, but most communications applications also need to identify and handle data according to a specific need. The APAXPort control provides high level functionality that simplifies tasks common to any serial communications application.

The more common RS-232 communications protocol was covered in detail in Chapter 2: Communications Basics and will not be covered again here.

RS-485 Support Overview

RS-485 serial networks usually consist of two or more serial devices all connected to the same 2-wire serial cable. The transmitted data is represented by voltage differences between the two lines instead of a voltage difference between a single line and a common ground as in RS-232. This difference allows RS-485 networks to operate over much greater distances than RS-232.

RS-485 requires specific serial port hardware that supports RS-485 voltages and conventions. Most standard serial ports provided on a motherboard and even most add-in serial ports do not support RS-485 mode.

Since both RS-485 wires are required to transmit data, an RS-485 device can either receive data or transmit data (but not both) at any given moment. RS-485 devices usually spend most of their time in receive mode, monitoring the line for incoming data. When one device starts transmitting all other devices in the network receive that data, so messages usually include an address byte to allow devices to ignore messages not addressed to them.

With such a network, the PC normally acts as a master, addressing and sending data to each remote slave device and processing its response before moving on to the next device. Before the PC can transmit, it must take control of the data line. While transmitting, it cannot receive any data so it must release control of the line after transmitting so it can receive the response. This switch from transmit to receive mode can be either automatic (controlled by the RS-485 board or converter), or it can be manual (controlled by the PC software or driver).

The mechanism provided by RS-485 boards for switching the data line from receive to transmit mode falls into three categories:

- RTS Control
- Automatic
- Other

Most currently available RS-485 boards use the RTS line to control the state of the data line. Before transmitting data, the application raises the RTS line of the port, which tells the RS-485 board to switch to transmit mode. After transmitting the data the application lowers RTS to switch the line back to receive mode. These boards are supported by the APAXPort control by using the RS485Mode property (with some exceptions noted in the following paragraphs).

Some boards and converters handle the RS-485 data line switch automatically, with no assistance from the software. These boards are supported by APAX but do not require use of the RS485Mode property.

The few remaining boards use proprietary techniques for providing RS-485 support instead of the RTS or automatic switching described above. These boards are not specifically supported by APAX, but can probably be used anyway if your code performs the actions required by the board's documentation.

RTS control

Since the APAXPort control provides an RTS property, your application could manually raise RTS before transmitting data and lower RTS after transmitting. This would work in theory, but is somewhat problematic in that when a PutXxx method has returned, the data may not have been completely transmitted. Lowering RTS at that time would result in some data being truncated. Even lowering RTS after a calculated delay would be error prone since the calculation would have to account for delays in UART (the serial port chip) buffering and would be susceptible to unpredictable delays due to multi-tasking.

A better approach is to use the APAXPort control's built-in RTS line control, which is available through the RS485Mode property. When RS485Mode is set to True, all PutXxx methods raise RTS before transmitting the first byte, wait for the data to be completely transmitted, then lower RTS. The wait accurately accounts for data in the UART, assuring that RTS is lowered at the proper time.

The RTS line control follows the output buffer. If the output buffer empties while your code is formatting and transmitting a command, the RTS line could be lowered and raised again. This might cause some RS-485 devices to misinterpret the message.

It is better to pre-format a command in a buffer and use a single PutData call to transmit it than to format and transmit at the same time using multiple PutXxx commands.

For example, you should use:

```
Message = "!" + Address + MsgLength + Message + "$"
PutString(Message)
```

rather than:

```
PutString("!")
PutString(Address)
PutString(MsgLength)
PutString(Message)
PutString(MsgLength)
PutString("$")
```

Under Windows 95/98 and Windows 2000, the waiting is handled within APAX since the communications API doesn't provide the necessary accuracy. Under Windows NT, the waiting is handled by the serial port driver, since it does provide the necessary accuracy.

The automatic handling of the RTS line is possible for standard ports in all environments and for all non-standard serial ports that provide the necessary driver support. Part of this support (in Windows 95/98 and Windows 2000) includes the detection of the serial port hardware's base address. If this address cannot be detected, attempts to transmit in RS-485 mode will raise an error.

If the serial port doesn't use standard serial port hardware, then RS485Mode cannot provide automatic RTS line control for that port. In such cases, however, the board likely provides some other mechanism for handling RS-485 support (assuming it's RS-485 capable) and you will need to consult the board's documentation for details.

Under Windows NT, the waiting is handled within the serial port driver and replacement drivers must also provide this support in order for the APAXPort control's RS485Mode property to work. If they do not, it is again likely that the board provides some other mechanism for supporting RS-485.

Debugging Facilities

In a perfect world, all programs would work flawlessly as they were typed in. Since things rarely work out this nicely, it is often necessary to break out the debugging tools and apply some hard-won debugging knowledge to get programs to behave themselves. Communications programs introduce some new debugging issues, and your existing tools and knowledge may no longer be adequate.

For example, suppose you're writing a data collection program that regularly receives data from an instrument and writes the data to a database. While testing, you notice that a small percentage of the data in the database is wrong. Broadly speaking, there are two explanations for such a problem: 1) the instrument sent bad data; or 2) your program somehow corrupted the good data before writing it to disk. Given that the errors occur infrequently, you'd probably have to add specific debugging code to your program to create an audit report of all received data. Later you'd compare this audit report to the data in the database. If the data matched, you would know that the instrument sent bad data; otherwise you could conclude that your program corrupted the data. Either way, you would know what debugging steps to take next. To gain this insight you would use Dispatch logging.

Dispatch logging

It is often beneficial to know exactly when data arrived at the port. The standard Windows communications driver doesn't provide enough information to determine exactly when data arrived. The next best thing is knowing when the APAX internal dispatcher got the data, and that's how dispatch logging works.

Dispatch logging creates an audit trail of each action taken by the APAXPort control. These entries are stored in a circular queue of a specified size. Since the queue is circular, it contains information about the most recent transmitted or received characters. Entries in this queue are of variable length, and the queue can be as large as 16 million bytes.

The queue can be dumped to a text file at any time. This text file is a report of all dispatcher events in the following format:

```
APAX v1.00
Operating System : Windows NT 4.0 Service Pack 4

Time      Type      SubType     Data      OtherData
--------  --------  ----------  --------  ---------
00000010  TrDatChg  Avail       00000001
00000010  TrgHdAlc  Window      7DDE03CE
00000010  TrgHdAlc  Window      870302A2
00000010  TrDatChg  Avail       00000001
00000010  TrgHdAlc  Procedure   00000000
00000010  TrDatChg  Avail       00000001
00000010  TrigAllc  Data        00000008   rz[0D]
00000010  TrigAllc  Data        00000010   [05]
00000010  TrigAllc  Data        00000018   [10]
00000010  TrigAllc  Data        00000020   [1B]I
00000010  TrigAllc  Status      00000029   (Modem status)
```

The first three lines are the header of the text file a provide the installed version of APAX, and the current operating system.

The first column of the report is a timestamp. It represents the time elapsed from the time dispatch logging was turned on to when the entry was made, measured in milliseconds. The multimedia API TimeGetTime is used to calculate this timestamp, so it should be accurate to the nearest millisecond. The second column is the major category of log entry, the third column identifies the log entry subtype, the forth column provides additional information related to the event (often a handle or data count), and the remaining column adds any additional information that could be useful for the event being logged.

The following tables identify possible entries in a dispatch log.

Log entry type: Dispatch

An entry of this type means that a communications event is being processed.

Subtype	Data	Other Data
ReadCom - The Windows communications driver has notified APAX that incoming data is available and APAX's dispatcher has, in turn, read the available data.	The number of bytes read	The actual data
WriteCom - APAX has sent data to the Windows comm driver.	The number of bytes sent	The actual data
Line status - A line status event has been received from the Windows comm driver.	None	None
Modem status - A modem status event has been received from the Windows comm driver.	The numeric value of the event received	A translation of the numeric value. (DCTS, DDSR, TERI, DDCD, CTS, DSR, RI, DCD)

Log entry type: Trigger

An entry of this type means that a trigger is being dispatched by the dispatcher.

Subtype	Data	Other Data
Avail - A data avail event is being dispatched.	The number of bytes ready to be read	None
Timer - A timer event is being dispatched.	The handle of the trigger	None
Data - A data trigger match event is being dispatched.	The handle of the trigger	None
Status - A status trigger is being dispatched.	The handle of the trigger	None

Log entry type: TrigAllc

An entry of this type means that a trigger is being allocated.

Subtype	Data	Other Data
Data - A data trigger is being allocated.	The handle of the trigger	What the trigger is being set to trigger on
Timer - A timer trigger is being allocated.	The handle of the trigger	None
Status - A status trigger is being allocated.	The handle of the trigger	The type of the status trigger

Log entry type: TrigDisp

An entry of this type means that a trigger is being disposed.

Subtype	Data	Other Data
Data - A data trigger is being deleted.	The handle of the trigger	None
Timer - A timer trigger is being deleted.	The handle of the trigger	None
Status - A status trigger is being deleted.	The handle of the trigger	None

Log entry type: TrgHdAlc

An entry of this type means that a trigger handler has been allocated.

Subtype	Data	Other Data
Window - A window handle based trigger handler is being registered with the comport.	The window handle	None
Procedure - A procedure pointer based trigger handler is being registered with the comport.	None	None
Method - A method pointer based trigger handler is being registered with the comport.	None	None

Log entry type: TrgHdDsp

An entry of this type means that a trigger handler has been disposed.

Subtype	Data	Other Data
Window - A window handle based trigger handler is being deregistered from the comport.	The window handle	None
Procedure - A procedure pointer based trigger handler is being deregistered from the comport.	None	None
Method - A method pointer based trigger handler is being deregistered from the comport.	None	None

Log entry type: TrDatChg

An entry of this type means that data associated with the trigger has been changed.

Subtype	Data	Other Data
Avail - The data trigger length value is being changed.	The new length	None
Timer - The time-out value for a particular timer trigger is being changed.	The handle of the timer trigger	The new time
Status - SetStatusTrigger is being called for a particular status trigger.	The handle of the trigger being changed	The new value mask for the trigger

Log entry type: Telnet

An entry of this type logs the telnet negotiation conversation during a Winsock telnet session. Each SubType shows the type of negotiation being logged.

Subtype	Data	Other Data
Sent WILL - APAX is acknowledging that it will support a requested mode.	The numeric value of the command being negotiated	A translation of the numeric command
Sent WON'T - APAX is refusing to support a requested mode.	The numeric value of the command being negotiated	A translation of the numeric command
Sent DO - APAX is requesting the support of a telnet mode.	The numeric value of the command being negotiated	A translation of the numeric command
Sent DON'T - APAX is requesting that a telnet mode not be supported.	The numeric value of the command being negotiated	A translation of the numeric command
Recv WILL - The telnet host is acknowledging that it will support a requested mode.	The numeric value of the command being negotiated	A translation of the numeric command

Subtype	Data	Other Data
Recv WON’T - The telnet host is refusing to support a requested mode.	The numeric value of the command being negotiated	A translation of the numeric command
Recv DO - The telnet host is requesting the support of a telnet mode.	The numeric value of the command being negotiated	A translation of the numeric command
Recv DON’T - The telnet host is requesting that a telnet mode not be supported.	The numeric value of the command being negotiated	A translation of the numeric command
Command - A subnegotiation command has been received. APAX currently doesn’t support any of these commands, but they are logged nonetheless.	The numeric value of the command being negotiated	The collected command
Sent Term - A string identifying the terminal emulation type has been sent to the host.	The numeric value of the command being negotiated	The string sent

Log entry type: Packet

An entry of this type is an event that indicates state changes in packets.

Subtype	Data	Other Data
Enable - The packet is being enabled. If the packet is using start- and/or end-strings, the next log entries are StartStr and EndStr events.	None	The name of the packet component
Disable - The packet is being enabled.	None	The name of the packet component
StringPacket - A string packet event is being dispatched. If the end condition was a string, the next log entry is another StringPacket event with the value of the end string in the 'other data' column. If the end condition was a size event, the next log entry is a SizePacket event.	None	The name of the packet component
SizePacket - Describes the end value for a StringPacket.	None	The value of the end condition of the previously listed StringPacket
PcktTimeout - A packet time-out event is being generated.	None	None
StartStr - Describes the start string for a particular packet. The event is always generated as a part of the enable sequence for the packet, if it has a start string. See the Enable event above.	None	The value of the start string of an enable sequence

Subtype	Data	Other Data
EndStr - Describes the current end string for a particular packet. The event is always generated as a part of the enable sequence for the packet, if it has an end string. See the Enable event above.	None	The end string of an enable sequence
Idle - The packet is not currently collecting data and is not waiting for a string.	None	None
Waiting - The packet is waiting for its start string to come in.	None	None
Collecting - The packet's start condition has been met and thus the packet currently has ownership to the incoming data.	None	None

Log entry type: Error

The dispatcher was called recursively. This is an error and can cause events to be missed. The cause is usually that event handlers take too long to process their data. No SubTypes exist for this type of entry.

Log entry type: XModem

Entries for this Type track APAX's progress through the send or receive XModem protocol state machine. The SubType indicates the current state of the state machine.

Log entry type: YModem

Entries for this Type track APAX's progress through the send or receive YModem protocol state machine. The SubType indicates the current state of the state machine.

Log entry type: ZModem

Entries for this Type track APAX's progress through the send or receive ZModem protocol state machine. The SubType indicates the current state of the state machine.

Log entry type: Kermit

Entries for this Type track APAX's progress through the send or receive Kermit protocol state machine. The SubType indicates the current state of the state machine.

Log entry type: Ascii

Entries for this Type track APAX's progress through the send or receive ASCII protocol state machine. The SubType indicates the current state of the state machine.

Log entry type: User

A user defined event type. You can add custom strings to a comports logfile using the comport's AddStringToLog method. These strings have the type User in the log file.

Logging facility

The state of the logging facility is controlled by setting the Logging property to one of the following values:

Value	**Explanation**
`tlOff`	`Turns off logging without saving the log data`
`tlOn`	`Turns logging on or resumes logging after a pause`
`tlDump`	`Writes the log data to a new file, turns off logging`
`tlAppend`	`Appends the log to an existing file, turns off logging`
`tlClear`	`Clears the log buffer but leaves logging on`
`tlPause`	`Pauses logging`

Port and Logging References

Following is a list of the APAXPort control's properties, methods, and events that pertain to basic serial communications (RS-232 and RS-485), and logging facilities. This is only a subset of the functionality of the APAXPort functionality. Additional properties, methods, and events are introduced in other chapters.

Properties

Baud	HWFlowUseDTR	OutBuffUsed
ComNumber	HWFlowUseRTS	Parity
CTS	InBuffFree	PromptForPort
DataBits	InBuffUsed	RI
DCD	LineError	RS485Mode
DeviceType	LogAllHex	RTS
DSR	Logging	StopBits
DTR	LogHex	SWFlowOptions
FlowState	LogName	TAPIMode
HWFlowRequireCTS	LogSize	XOffChar
HWFlowRequireDSR	OutBuffFree	XOnChar

Methods

AddStringToLog	FlushOutBuffer	PutString
Close	PortOpen	PutStringCRLF
FlushInBuffer	PutData	SendBreak

Events

OnCTSChanged	OnLineBreak	OnPortOpen
OnDCDChanged	OnLineError	OnRing
OnDSRChanged	OnPortClose	OnRXD

Reference Section

AddStringToLog method

Description

Adds the specified string to the log file specified by the LogName property.

Syntax

expression.**AddStringToLog(*S*)**

Part	**Description**	**Data Type**
expression	An expression that returns an APAXPort object	**APAXPort**
S	Specifies the string to be added	**String**

Remarks

This procedure is useful if you need to add more human-readable content to the log file. The parameter passed in appears on a new line of the log file followed by a carriage return/line feed pair. This procedure has no effect if the Logging property is set to tlOff or tlPause, or if the LogName property is blank.

Example

The following example adds the line handshaking to the log file:

```
APAXPort1.Logging = tlOn
APAXPort1.AddStringToLog("Handshaking")
```

See also

Logging, LogName, LogSize, LogHex, LogAllHex

Baud property

Description

Determines the baud rate used by the port. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`Baud`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Default: 19200

Generally acceptable values for Baud include 300, 1200, 2400, 4800, 9600, 19200, 38400, 57600, and 115200.

If the port is open when Baud is changed, the line parameters are updated as soon as any data existing in the output buffer has drained. Baud does not validate the assigned value before passing it on to the communications driver. The driver may reject the value, leading to an exception.

See also

ComNumber, DataBits, Parity, StopBits

Close method

Description

Closes the port.

Syntax

expression.**(Close)**

expression must reference an **APAXPort.**

Remarks

Calling Close terminates an active connection and closes the port. This method applies to all device types. After the physical port has been closed and input and output buffers have been deallocated, the OnPortClose event is fired.

See also

OnPortClose, PortOpen

ComNumber property

Description

Determines the serial port number (Com1, Com2, etc.) used by the APAXPort component. Read/write.

Data type

Integer

Syntax

expression.**ComNumber**[= *value*]

expression must reference an **APAXPort.**

Remarks

Default: 0

ComNumber does not validate the port number. When the port is opened, the Windows communications driver will determine whether the port number is valid and generate an error if it is not.

If the port is open when ComNumber is changed, the existing port is closed and then reopened using the new number. Triggers are maintained during this operation.

Example

The following example creates, configures, and opens a comport component at run time:

```
Dim MyPort as APAXPort
...
Set MyPort = new APAXPort
MyPort.ComNumber = 1 'use Com1
MyPort.Baud = 9600
MyPort.Parity = pNone
MyPort.DataBits = 8
MyPort.StopBits = 1
ApdComPort.Connected = True
MyPort.PortOpen
```

See also

Baud, Parity, DataBits, StopBits, PortOpen, DeviceType

CTS property

Description

Returns True if the port's clear to send line (CTS) is set. Read-only, run-time.

Data type

`Boolean`

Syntax

`expression``.`**`CTS`**

`expression` `must reference an` **`APAXPort.`**

Remarks

The following example transmits a large block of data after assuring that the remote has raised the CTS signal:

```
if APAXPort1.CTS = True Then
  APAXPort1.PutData(BigBlock, 1024)
End If
```

See also

DSR

DataBits property

Description

Determines the number of data bits of the port. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`DataBits`**`[= `*`value`*`]`

`expression`` must reference an `**`APAXPort.`**

Remarks

Default: 8

Acceptable values are 5, 6, 7, and 8.

If the port is open when DataBits is changed, the line parameters are updated immediately. DataBits does not validate the assigned value before passing it on to the communications driver. The driver may reject the value, leading to an exception.

See also

Baud, ComNumber, Parity, StopBits

DCD property

Description

Returns True if the port's data carrier detect line (DCD) is set. Read-only, run-time.

Data type

`Boolean`

Syntax

`expression``.`**`DCD`**

`expression`` must reference an `**`APAXPort.`**

Remarks

DCD is usually set only for serial connections made through a modem. Your modem sets DCD to indicate that it has a connection with another modem. If either modem hangs up or the connection is lost for another reason, your modem clears DCD (assuming it is configured to do so.) Hence, if your application uses a modem connection, you might want to check DCD periodically to assure that the connection is still valid or, better yet, use a modem status trigger for the same purpose.

The following example detects carrier loss and handles the error:

```
if APAXPort1.DCD = False then
  'handle unexpected disconnect
End If
```

See also

DSR

DeviceType property

Description

Defines the mode in which the port operates.

Syntax

`expression.`**`DeviceType`**`[= value]`

expression must reference an **APAXPort.**

Settings

Valid settings for DeviceType are:

Constant	**Description**
xdtDirect	Direct mode is generally used to communicate with a serial device over a direct connection.
xdtTAPI	TAPI mode is generally used to establish or answer calls using TAPI features.
xdtWinSock	Winsock mode is used to establish communications over network and Internet connections.

Remarks

Default: xdtDirect

See also

PortOpen, WinsockConnect, WinsockListen, TAPIAnswer, TAPIDial

DSR property

Description

Returns True if the port’s data set ready line (DSR) is set. Read-only, run-time.

Data type

`Boolean`

Syntax

*`expression.`***`DSR`**

`expression`` must reference an `**`APAXPort.`**

Remarks

DSR is a signal that the remote device sets to indicate that it is attached and active. You might want to check this signal before transmitting and periodically thereafter.

See also

DCD

DTR property

Description

Determines the current state of the data terminal ready signal (DTR). Read/write.

Data type

`Boolean`

Syntax

*`expression.`***`DTR`**`[= `*`value`*`]`

`expression`` must reference an `**`APAXPort.`**

Remarks

Default: True

Some types of remote devices require that this signal be raised before they transmit. For example, the default configuration of many modems is not to transmit data unless the PC has raised the DTR signal. Use this property to check the status of the DTR line or to assert the DTR line to inform the remote that you are ready to receive.

Example

The following example lowers the DTR signal after opening the port and later raises it again:

```
Dim MyPort as APAXPort
Set MyPort = New APAXPort
MyPort.PortOpen
MyPort.DTR = False
...
MyPort.DTR = True
```

See also

RTS

FlowState property

Description

Returns the state of hardware or software flow control. Read-only, run-time.

Data type

`TFlowControlState`

Syntax

`expression``.`**`FlowState`**

`expression` `must reference an` **`APAXPort.`**

Settings

Valid settings for FlowState are:

Constant	Description
fcOff	Flow control is not in use.
fcOn	Flow control is enabled, but blocking is not imposed in either direction.
fcDSRHold	The application cannot transmit because the other side has lowered the DSR line.
fcCTSHold	The application cannot transmit because the other side has lowered the CTS line.
fcDCDHold	The application cannot transmit because the other side has lowered the DCD line. Note: The APAXPort does not currently provide DCD flow control.

Constant	Description
fcXOutHold	The application cannot transmit because it has received an XOff character from the remote.
fcXInHold	The application has sent an XOff character to the remote to prevent it from transmitting data.
fcXBothHold	The application has both sent and received an XOff character.

5

Remarks

Windows doesn't provide information on the state of receive hardware flow control, so fcOn is returned even if the local device is blocking received data by using a hardware flow control line.

In the rare case where both hardware and software flow control are enabled for a port, FlowState can return ambiguous results. In particular, if flow is blocked by both hardware and software flow control, FlowState can return only the fact that one type is causing the block.

See also

HWFlowUseDTR, HWFlowUseRTS, HWFlowRequireDSR, HWFlowRequireCTS, SWFlowOptions

FlushInBuffer method

Description

Clears the input buffers used by both the Windows device driver and the APAX internal dispatcher.

Syntax

`expression.`**`FlushInBuffer`**

`expression` `must reference an` **`APAXPort.`**

Remarks

FlushInBuffer also resets all data triggers to disregard any cleared data.

Example

The following example flushes all data currently in the input buffer if a line error is detected:

```
If APAXPort1.LineError <> leNoError Then
  '...error handling
  APAXPort1.FlushInBuffer
End If
```

You probably shouldn't do this routinely after each line error. Logic like this is usually appropriate only before trying to synchronize with the transmitter in a file transfer protocol.

See also

FlushOutBuffer

FlushOutBuffer method

Description

Clears the output buffers used by both the Windows device driver and the APAX internal dispatcher.

Syntax

`*expression*.**FlushOutBuffer**`

expression must reference an **APAXPort.**

Remarks

Any data pending in the output buffer is not transmitted.

Example

The following example discards any data in the output buffer after a remote device reports an error:

```
If ErrorDetected = True Then
  APAXPort1.FlushOutBuffer
  '...resync with remote
End If
```

See also

FlushInBuffer

HWFlowRequireCTS property

Description

Determines the CTS hardware flow control options for the port. Read/write.

Data type

`Boolean`

Syntax

`expression``.`**`HWFlowRequireCTS`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Hardware flow options can be combined to enable hardware flow control.

Receive flow control stops a remote device from transmitting while the local input buffer is too full. Transmit flow control stops the local device from transmitting while the remote input buffer is too full.

Setting the HWFlowUseRTS and/or HWFlowUseDTR properties to True enables receive flow control. When receive flow control is enabled, the corresponding modem control signals (RTS and/or DTR) are lowered when the input buffer reaches the 90% level. The remote must recognize these signals and stop sending data while they are held low.

As the application processes received characters, buffer usage eventually drops below the 10% level. At that point, the corresponding modem control signals are raised again. The remote must recognize these signals and start sending data again.

Transmit flow control is enabled by setting the HWFlowRequireCTS and/or HWFlowRequireDSR properties to True. With one or both of these options enabled, the Windows communications driver doesn't transmit data unless the remote device is providing the corresponding modem status signal (CTS and/or DSR). The remote must raise and lower these signals when needed to control the flow of transmitted characters.

Note that flow control using RTS and CTS is much more common than flow control using DTR and DSR.

Example

The following example enables bi-directional hardware flow control:

```
MyPort.HWFlowUseRTS = True
MyPort.HWFlowRequireCTS = True
... 'use port
MyPort.HWFlowUseRTS = False
MyPort.HWFlowRequireCTS = False
```

RTS is lowered for receive flow control and CTS is checked for transmit flow control. Later in the application, hardware flow control is disabled.

See also

HWFlowUseDTR, HWFlowUseRTS, HWFlowRequireDSR, HWFlowRequireCTS, SWFlowOptions

HWFlowRequireDSR property

Description

Determines the DSR hardware flow control options for the port. Read/write.

Data type

`Boolean`

Syntax

`expression``.`**`HWFlowRequireDSR`**`[= `*`value`*`]`

`expression`` must reference an `**`APAXPort.`**

Remarks

Hardware flow options can be combined to enable hardware flow control.

Receive flow control stops a remote device from transmitting while the local input buffer is too full. Transmit flow control stops the local device from transmitting while the remote input buffer is too full.

Setting the HWFlowUseRTS and/or HWFlowUseDTR properties to True enables receive flow control. When receive flow control is enabled, the corresponding modem control signals (RTS and/or DTR) are lowered when the input buffer reaches the 90% level. The remote must recognize these signals and stop sending data while they are held low.

As the application processes received characters, buffer usage eventually drops below the 10% level. At that point, the corresponding modem control signals are raised again. The remote must recognize these signals and start sending data again.

Transmit flow control is enabled by setting the HWFlowRequireCTS and/or HWFlowRequireDSR properties to True. With one or both of these options enabled, the Windows communications driver doesn't transmit data unless the remote device is providing the corresponding modem status signal (CTS and/or DSR). The remote must raise and lower these signals when needed to control the flow of transmitted characters.

Note that flow control using RTS and CTS is much more common than flow control using DTR and DSR.

Example

The following example enables bi-directional hardware flow control:

```
MyPort.HWFlowUseRTS = True
MyPort.HWFlowRequireCTS = True
... 'use port
MyPort.HWFlowUseRTS = False
MyPort.HWFlowRequireCTS = False
```

RTS is lowered for receive flow control and CTS is checked for transmit flow control. Later in the application, hardware flow control is disabled.

See also

HWFlowUseDTR, HWFlowUseRTS, HWFlowRequireDSR, HWFlowRequireCTS, SWFlowOptions

HWFlowUseDTR property

Description

Determines the DTR hardware flow control options for the port. Read/write.

Data type

`Boolean`

Syntax

`expression``.`**`HWFlowUseDTR`**`[= `*`value`*`]`

`expression`` must reference an `**`APAXPort.`**

Remarks

Hardware flow options can be combined to enable hardware flow control.

Receive flow control stops a remote device from transmitting when the local input buffer is too full. Transmit flow control stops the local device from transmitting when the remote input buffer is too full.

Setting the HWFlowUseRTS and/or HWFlowUseDTR properties to True enables receive flow control. When receive flow control is enabled, the corresponding modem control signals (RTS and/or DTR) are lowered when the input buffer reaches the 90% level. The remote must recognize these signals and stop sending data while they are held low.

As the application processes received characters, buffer usage eventually drops below the 10% level. At that point, the corresponding modem control signals are raised again. The remote must recognize these signals and start sending data again.

Transmit flow control is enabled by setting the HWFlowRequireCTS and/or HWFlowRequireDSR properties to True. With one or both of these options enabled, the Windows communications driver doesn't transmit data unless the remote device is providing the corresponding modem status signal (CTS and/or DSR). The remote must raise and lower these signals when needed to control the flow of transmitted characters.

Note that flow control using RTS and CTS is much more common than flow control using DTR and DSR.

Example

The following example enables bi-directional hardware flow control:

```
MyPort.HWFlowUseRTS = True
MyPort.HWFlowRequireCTS = True
... 'use port
MyPort.HWFlowUseRTS = False
MyPort.HWFlowRequireCTS = False
```

RTS is lowered for receive flow control and CTS is checked for transmit flow control. Later in the application, hardware flow control is disabled.

See also

HWFlowUseRTS, HWFlowRequireDSR, HWFlowRequireCTS, SWFlowOptions

HWFlowUseRTS property

Description

Determines the RTS hardware flow control options for the port. Read/write.

Data type

`Boolean`

Syntax

`expression``.`**`HWFlowUseRTS`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Hardware flow options can be combined to enable hardware flow control.

Receive flow control stops a remote device from transmitting while the local input buffer is too full. Transmit flow control stops the local device from transmitting while the remote input buffer is too full.

Setting the HWFlowUseRTS and/or HWFlowUseDTR properties to True enables receive flow control. When receive flow control is enabled, the corresponding modem control signals (RTS and/or DTR) are lowered when the input buffer reaches the 90% level. The remote must recognize these signals and stop sending data while they are held low.

As the application processes received characters, buffer usage eventually drops below the 10% level. At that point, the corresponding modem control signals are raised again. The remote must recognize these signals and start sending data again.

Transmit flow control is enabled by setting the HWFlowRequireCTS and/or HWFlowRequireDSR properties to True. With one or both of these options enabled, the Windows communications driver doesn't transmit data unless the remote device is providing the corresponding modem status signal (CTS and/or DSR). The remote must raise and lower these signals when needed to control the flow of transmitted characters.

Note that flow control using RTS and CTS is much more common than flow control using DTR and DSR.

Example

The following example enables bi-directional hardware flow control with limits at the 10% and 90% levels of the buffer:

```
MyPort.HWFlowUseRTS = True
MyPort.HWFlowRequireCTS = True
... 'use port
MyPort.HWFlowUseRTS = False
MyPort.HWFlowRequireCTS = False
```

RTS is lowered for receive flow control and CTS is checked for transmit flow control. Later in the application, hardware flow control is disabled.

See also

HWFlowUseDTR, HWFlowUseRTS, HWFlowRequireDSR, HWFlowRequireCTS, SWFlowOptions

InBuffFree property

Description

Returns the number of bytes free in the dispatcher buffer. Read-only, run-time.

Data type

`Integer`

Syntax

`expression``.`**`InBuffFree`**

`expression`` must reference an `**`APAXPort.`**

Remarks

This routine returns the number of bytes of free space in the APAXPort dispatcher buffer. It does not tell you the free space in the Windows communications driver input buffer.

Because the dispatcher automatically drains the Windows buffer, its status is rarely relevant to the program.

Example

The following example checks to see that there's significant free space in the dispatcher buffer before performing a time-consuming operation that doesn't drain the buffer:

```
if MyPort.InBuffFree > 128 Then
  '...perform a time-consuming operation
End If
```

See also

InBuffUsed

InBuffUsed property

Description

Returns the number of bytes currently available for reading from the dispatcher buffer. Read-only, run-time.

Data type

`Integer`

Syntax

`expression``.`**`InBuffUsed`**

`expression` `must reference an` **`APAXPort.`**

Remarks

This routine returns the number of bytes currently loaded in the APAX dispatcher buffer. It does not include bytes in the Windows communications driver input buffer that haven't yet been moved to the dispatcher buffer.

Because the dispatcher automatically drains the Windows buffer, this buffer's status is rarely relevant to the program.

The following example checks InBuffUsed to see if received data is available for processing:

```
if MyPort.InBuffUsed <> 0 Then
  '...process data
End If
```

See also

InBuffFree

LineError property

Description

Returns a non-zero value if line errors have occurred since the last call to LineError. Read-only/run-time.

Data type

`Integer`

Syntax

`*expression*.**LineError**`

`*expression* must reference an **APAXPort.**`

Remarks

Line errors can occur during calls to the PutData or PutString methods. If your application must detect line errors, it should check LineError after each such call or group of calls.

The LineError property returns 0 if no errors were detected or the port is not yet open. Otherwise it returns a numeric value from the following list that indicates the most severe pending error:

Constant	Value	Description
leBuffer	1	Buffer overrun in COMM.DRV
leOverrun	2	UART receiver overrun
leParity	3	UART receiver parity error
leFraming	4	UART receiver framing error
leCTSTO	5	Transmit time-out waiting for CTS
leDSRTO	6	Transmit time-out waiting for DSR
leDCDTO	7	Transmit time-out waiting for DCD (RLSD)
leTxFull	8	Transmit queue is full

Example

The following example checks for line errors after receiving data with GetBlock:

```
Ch = MyPort.PutString("Testing")
if MyPort.LineError <> 0 Then
  '...error handling
End If
```

See also

PutData, PutString

LogAllHex property

Description

Determines whether or not all port data is written to the log file in hex format. Read/write.

Data type

`Boolean`

Syntax

`expression``.`**`LogAllHex`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

By default, only certain characters are logged in hexadecimal format. Set this property to True to log all characters in hexadecimal format.

See also

LogHex, LogName, LogSize, Logging

Logging property

Description

Determines the current logging state. Read/write.

Data type

`TTraceLogState`

Syntax

`expression``.`**`Logging`**`[= `*`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Valid settings for Logging are:

Constant	Value	Description
tlOff	0	No logging is performed
tlOn	1	Enables logging
tlDump	2	Writes the contents of the logging buffer to disk and sets logging to tlOff
tlAppend	3	Appends the contents of the logging buffer to disk and sets logging to tlOff
tlClear	4	Clears the contents of the logging buffer and continues logging
tlPause	5	Temporarily pauses logging. To resume logging, set this property to tlOn

Remarks

Default: tlOff

Setting this property to tlOn allocates an internal buffer of LogSize bytes and informs the dispatcher to start using this buffer. To disable logging without writing the contents of the log buffer to a disk file, set Logging to tlOff. This also frees the internal buffer.

Setting Logging to tlDump overwrites any existing file named LogName, or creates a new file if a file of this name does not exist.

Example

The following example turns on logging and later dumps the logging buffer to APAX.LOG:

```
MyPort.Logging = tlOn
'...
MyPort.LogName = "APAX.LOG"
MyPort.Logging = tlDump
```

See also

LogHex, LogAllHex, LogName, LogSize, Tracing

LogHex property

Description

Determines whether non-printable characters stored in a dispatch logging file are written using hexadecimal or decimal notation. Read/write.

Data type

`Boolean`

Syntax

`expression``.`**`LogHex`**`[= `*`value`*`]`

`expression`` must reference an `**`APAXPort.`**

Remarks

Default: True

See also

Logging, LogName, LogSize, LogAllHex

LogName property

Description

Determines the name of the file used to store a dispatch log. Read/write.

Data type

`String`

Syntax

`expression.``**LogName**[= `*`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Default: LogName is APAX.LOG

See also

Logging, LogSize, LogAllHex

LogSize property

Description

Determines the number of bytes allocated for the dispatch logging buffer. Read/write.

Data type

`Integer`

Syntax

`expression.``**LogSize**[= `*`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Default: 10000

Each dispatch entry consumes at least 10 bytes. Many entries use additional buffer space to store a sequence of received or transmitted characters.

This property should normally be set before a logging session begins. If a changed value is assigned to LogSize while a logging session is active, the current session is aborted (which clears all information from the logging buffer), the new buffer is allocated, and a new logging session is started.

See also

Logging, LogName, LogAllHex

OnCTSChanged event

Description

Defines an event that is fired any time the CTS modem line changes state.

Syntax

`Private Sub` *`expression`* **`OnCTSChanged(ByVal`** *`NewValue`* **`as Boolean)`**

Part	**Description**	**Data Type**
expression	References the **APAXPort** object that fired the event	**APAXPort**
NewValue	Represents the current state of the CTS line	**Boolean**

Remarks

This event is triggered automatically whenever the CTS (clear to send) modem status line changes state. The current state of the CTS line is determined by reading the value of NewValue. NewValue is True when the CTS line transitioned to a high (set) state. Otherwise, NewValue is False.

See also

OnDCDChanged, OnDSRChanged, OnLineBreak, OnLineError, OnRing

OnDCDChanged event

Description

Defines an event that is fired any time the DCD modem line changes state.

Syntax

Private Sub *expression*_**OnDCDChanged(ByVal** *NewValue* **as Boolean)**

Part	Description	Data Type
expression	References the **APAXPort** object that fired the event	**APAXPort**
NewValue	Represents the current state of the DCD line	**Boolean**

Remarks

This event is triggered automatically whenever the DCD (data carrier detect) modem status line changes state. The current state of the DCD line is determined by reading the value of NewValue. NewValue will be True if the DCD line transitioned to a high (set) state. Otherwise, NewValue will be False.

See also

OnCTSChanged, OnDSRChanged, OnLineBreak, OnLineError, OnRing

OnDSRChanged event

Description

Defines an event that is fired any time the DSR modem line changes state.

Syntax

Private Sub *expression*_**OnDSRChanged(ByVal** *NewValue* **as Boolean)**

Part	Description	Data Type
expression	References the **APAXPort** object that fired the event	**APAXPort**
NewValue	Represents the current state of the DSR line	**Boolean**

Remarks

This event is triggered automatically whenever the DSR (data set ready) modem status line changes state. The current state of the DSR line is determined by reading the value of NewValue. NewValue will be True if the DSR line transitioned to a high (set) state. Otherwise, NewValue will be False.

See also

OnCTSChanged, OnDCDChanged, OnLineBreak, OnLineError, OnRing

OnLineBreak event

Description

Defines an event that is fired any time a line break is received.

Syntax

`Private Sub` *`expression_`***`OnLineBreak()`**

`expression` `references the` **`APAXPort`** `that fired the event.`

Remarks

This event is triggered automatically whenever the line break condition was received.

See also

OnCTSChanged, OnDCDChanged, OnDSRChanged, OnLineError, OnRing

OnLineError event

Description

Defines an event that is fired any time a line error occurs.

Syntax

`Private Sub` *`expression_`***`OnLineError(ByVal`** *`Error`* **`as TAPXLineError)`**

Part	Description	Data Type
expression	References the **APAXPort** object that fired the event	**APAXPort**
Error	Indicates the line error that occurred	**TAPXLineError**

Settings

Possible values for Error are:

`Constant`	`Description`
`xlsParity`	`A parity error occurred`
`xlsFraming`	`A framing error occurred`
`xlsOverrun`	`An overrun error occurred`

Remarks

Line errors can occur during calls to PutData, PutString, or PutStringCRLF. Line errors can also occur after data has been received (following the OnRXD event). If your application must detect line errors, you should write a handler for this event.

See also

OnCTSChanged, OnDCDChanged, OnDSRChanged, OnLineBreak, OnRing, PutData, PutString, PutStringCRLF

OnPortClose event

Description

Defines an event that is fired just before the port is closed.

Syntax

`Private Sub` *`expression`*`_`**`OnPortClose()`**

`expression` `refers to the` **`APAXPort`** `that is about to be closed.`

Remarks

Use an event handler here to perform any necessary cleanup prior to closing the port and freeing the port's resources.

See also

OnPortOpen

5

OnPortOpen event

Description

Defines an event that is fired just before the port is opened.

Syntax

`Private Sub` *`expression_`***`OnPortOpen()`**

`expression`` refers to the `**`APAXPort`**` that is about to be opened.`

Remarks

Use an event handler here to perform any necessary setup prior to opening the port.

See also

OnPortOpen

OnRing event

Description

Defines an event that is fired any time a ring indication is received.

Syntax

`Private Sub` *`expression_`***`OnRing()`**

`expression`` refers to the `**`APAXPort`**` that fired the event.`

Remarks

This event is triggered automatically whenever the leading edge of a ring signal is detected.

See also

OnCTSChanged, OnDCDChanged, OnDSRChanged, OnLineError, OnLineBreak

OnRXD event

Description

Defines an event that is fired when data is received.

Syntax

```
Private Sub expression_OnRXD(
  Data as OleVariant, ByVal Size as Integer)
```

Part	Description	Data Type
expression	References the **APAXPort** object that fired the event	**APAXPort**
Data	Contains the received data	**OleVariant**
Size	Contains the number of characters received	**Integer**

Remarks

This event is generated whenever data is received in the port. This event should be used to read data from the input stream when you don't know the format of the incoming data. If you know the format of the incoming data, you should configure one or more data packets to capture the data and notify you of its arrival rather than use this event.

Most often, the Data parameter will contain only a single character and the Size parameter will be one. It is possible however, that multiple characters were received before this event fires. In this situation, you should read exactly Size characters from the Data parameter. See OnDataTrigger Event on page 305 for additional information.

OutBuffFree property

Description

Returns the number of bytes free in the output buffer. Read-only, run-time.

Data type

`Integer`

Syntax

`expression``.`**`OutBuffFree`**

`expression` `must reference an` **`APAXPort.`**

Remarks

Use OutBuffFree to assure that the output buffer has enough free space to hold data that you are about to transmit.

Example

The following example checks for sufficient output buffer space to transmit a block of NeededSpace bytes:

```
If (MyPort.OutBuffFree >= NeededSpace) Then
  MyPort.PutData(Data, NeededSpace)
End If
```

If enough space is available the block is transmitted. Otherwise a status trigger is added to detect the required free space.

See also

OutBuffUsed

OutBuffUsed property

Description

Returns the number of bytes currently in the output buffer. Read-only, run-time.

Data type

`Integer`

Syntax

`expression``.`**`OutBuffUsed`**

`expression` `must reference an` **`APAXPort.`**

Remarks

Use OutBuffUsed to detect whether or not any outgoing data remains in the output buffer.

See also

OutBuffFree

Parity property

Description

Determines the parity checking mode of the port. Read/write.

Data type

`TParity`

Syntax

`expression``.`**`Parity`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Settings

Valid settings for Parity are:

Constant	Value	Description
`pNone`	`0`	`No parity checking`
`pOdd`	`1`	`Odd parity`

Constant	Value	Description
pEven	2	Even parity
pMark	3	Mark parity
pSpace	4	Space parity

Remarks

Default: pNone

If the port is open when Parity is changed, the line parameters are updated immediately. Parity does not validate the assigned value before passing it on to the communications driver. The driver may reject the value, leading to an exception.

See also

Baud, ComNumber, DataBits, StopBits

PortOpen method

Description

Opens the physical port and initializes it with all current port properties.

Syntax

expression.**PortOpen()**

expression must reference an **APAXPort.**

Remarks

Calling PortOpen sets the DeviceType to dtDirect and opens the COM port specified by the ComNumber property. If ComNumber is zero and the PromptForPort property is True, a port selection dialog is displayed. If the port is already open, the port is first closed and then reopened.

The APAXPort uses all current property settings to allocate input and output buffers, open the physical port, initialize the line settings and flow control settings, and enable or disable logging. The OnPortOpen event is then fired.

The port can be closed by calling the Close method.

Example

The following example creates, configures, and opens a comport component at run time:

```
Dim MyPort as APAXPort
...
Set Myport = new APAXPort
MyPort.ComNumber = 1 'use Com1
MyPort.Baud = 9600
MyPort.Parity = pNone
MyPort.DataBits = 8
MyPort.StopBits = 1

MyPort.OpenPort
  'transmit/receive data via the com port
MyPort.Close
```

See also

Close, ComNumber, DeviceType, OnPortOpen

PromptForPort property

Description

Indicates whether the user should be prompted for the serial port number. Read/write.

Data type

`Boolean`

Syntax

`expression`.**`PromptForPort`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Default: True

Applies only when DeviceType = dtDirect.

If PromptForPort is True and ComNumber is zero, a dialog is displayed to prompt the user for the serial port when the port is opened. If PromptForPort is False and ComPort is zero, an error is raised when the port is opened.

See also

ComNumber, DeviceType

PutData method

Description

Copies a block of data to the output buffer of the Windows communications driver.

Syntax

`expression``.`**`PutData(`*`Data, Size`*`)`**

`expression`` must reference an `**`APAXPort.`**

Part	**Description**	**Data Type**
expression	An expression that returns an **APAXPort** object	**APAXPort**
Data	Refers to the block of data to be transmitted	**OleVariant**
Size	Specifies the size of the data block	**Integer**

Remarks

After a call to PutData, the communications driver transmits the block byte-by-byte as fast as possible. When there is insufficient free space in the output buffer, the documented behavior of the Windows communications driver is to delete old data from the buffer. To avoid this behavior, programs should always check OutBuffFree before calling any PutXxx methods.

Example

The following example transmits a block of 20 characters after assuring that space is available:

```
S = "Guinness Stout"
If (MyPort.OutBuffFree >= Len(S)) Then
  MyPort.PutString(S)
End If
```

See also

OutBuffFree, PutString, PutStringCRLF

PutString method

Description

Copies a string to the output buffer of the Windows communications driver.

Syntax

expression.**PutString(*S*)**

expression must reference an **APAXPort.**

S defines the **String** to be transmitted.

Remarks

After calling this method, the communications driver transmits the string as soon as possible.

Example

The following example transmits a string after assuring that space is available:

```
S = "Guinness Stout"
If (MyPort.OutBuffFree >= Len(S)) Then
  MyPort.PutString(S)
End If
```

See also

OutBuffFree, PutData, PutStringCRLF

PutStringCRLF method

Description

Copies a string and a carriage return/line feed pair to the output buffer of the Windows communications driver.

Syntax

expression.**PutStringCRLF(*S*)**

expression must reference an **APAXPort.**

S defines the **String** to be transmitted.

Remarks

After calling this method, the communications driver transmits the string and the <CR><LF> pair as soon as possible.

Example

The following example transmits a string after assuring that space is available:

```
S = "Guinness Stout"
If (MyPort.OutBuffFree >= Len(S) + 2) Then
  MyPort.PutStringCRLF(S)
End If
```

See also

OutBuffFree, PutData, PutString

RI property

Description

Returns True if the port's ring indicator line (RI) is set. Read-only, run-time.

Data type

`Boolean`

Syntax

`expression``.`**`RI`**

`expression` `must reference an` **`APAXPort.`**

Remarks

Because the ring indicator line fluctuates rapidly as rings occur, reliance upon this property is discouraged.

See also

CTS, DCD, DSR, DTR

RS485Mode property

Description

Determines whether the RTS line should be raised/lowered automatically when transmitting data. Read/write.

Data type

`Boolean`

Syntax

`*expression*.**RS485Mode**[= *value*]`

`*expression* must reference an **APAXPort.**`

Remarks

Default: False

Set this property to True when using an RS-485 board or converter that uses the RTS line to enable the transmit line. In this mode, RTS will be raised whenever the program is transmitting data and lowered at all other times.

This property should be set to True only when a program is using RS-485 ports or converters and only if those ports or converters use RTS for line control. Enabling this property at other times could cause programs to behave erratically or stop working completely.

Because RS-485 mode requires control over the RTS line, the RTS property is set to False and CTS/RTS hardware flow control is disabled whenever RS485Mode is set to True.

RTS property

Description

Determines the current state of the request to send signal (RTS). Read/write.

Data type

`Boolean`

Syntax

`expression``.`**`RTS`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Default: True

This signal is usually used for hardware flow control, in which case your application does not need to set it directly. Less frequently, devices require that your application raise and lower RTS to control the device, or require that RTS be permanently set. Use this property in those cases.

Example

The following example lowers the RTS signal after opening the port and later raises it again:

```
MyPort.Connected = True
MyPort.RTS = False
...
MyPort.RTS = True
```

See also

DTR, HWFlowUseDTR, HWFlowUseRTS, HWFlowRequireDSR, HWFlowRequireCTS

SendBreak method

Description

Transmits a break signal.

Syntax

expression.**SendBreak(*Ticks, Yield*)**

Part	Description	Data Type
expression	An expression that returns an **APAXPort** object	**APAXPort**
Ticks	Determines the duration of the break	**Integer**
Yield	Determines the duration of the break	**Boolean**

Remarks

This method transmits a break signal (the transmit line is held in the marking state) for the number of ticks specified by Ticks. A tick is approximately 55 milliseconds.

When Yield is True, SendBreak yields control back to Windows while sending the break, giving other applications and other parts of this application a chance to run. When Yield is False, SendBreak does not yield.

StopBits property

Description

Determines the number of stop bits of the port. Read/write.

Data type

Integer

Syntax

expression.**StopBits**[= *value*]

expression must reference an **APAXPort.**

Remarks

Default: 1

Acceptable values are 1 and 2. If DataBits equals 5, a request for 2 stop bits is interpreted as a request for 1.5 stop bits, the standard for this data size.

If the port is open when StopBits is changed, the line parameters are updated immediately. StopBits does not validate the assigned value before passing it on to the communications driver. The driver may reject the value, leading to an exception.

See also

Baud, ComNumber, DataBits, Parity

SWFlowOptions property

Description

Determines the software flow control options for the port. Read/write.

Data type

`TSWFlowOptions`

Syntax

*`expression.`***`SWFlowOptions`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Settings

Valid settings for SWFlowOptions are:

Constant	Value	Description
swfNone	0	No software flow control is imposed
swfReceive	1	Stops a remote device from transmitting when the local receive buffer is too full
swfTransmit	2	Stops the local device from transmitting when the remote's receive buffer is too full
swfBoth	3	Imposes both receive and transmit flow control

Remarks

Default: swfNone

This routine turns on one or both aspects of automatic software flow control based on the value assigned to the property.

Receive flow control stops a remote device from transmitting while the local receive buffer is too full. Transmit flow control stops the local device from transmitting while the remote receive buffer is too full.

Receive flow control is enabled by assigning swfReceive or swfBoth to the property. When enabled, an XOff character is sent when the input buffer reaches the 90% level. The remote must recognize this character and stop sending data after it is received.

As the application processes received characters, buffer usage eventually drops below the 10% level. At that point, an XOn character is sent. The remote must recognize this character and start sending data again.

Transmit flow control is enabled by assigning swfTransmit or swfBoth to the property. When transmit flow control is enabled, the communications driver stops transmitting whenever it receives an XOff character. The driver does not start transmitting again until it receives an XOn character or the application sets SWFlowOptions to swfNone.

The default characters are used for XOn and XOff. Later in the application, software flow control is disabled.

See also

FlowState, HWFlowUseDTR, HWFlowUseRTS, HWFlowRequireDSR, HWFlowRequireCTS

TAPIMode property

Description

Determines whether a APAXPort can be controlled by a TAPI device. Read/write.

Data type

`TApxTAPIMode`

Syntax

`expression.`**`TAPIMode`**`[= value]`

`expression` `must reference an` **`APAXPort.`**

Settings

Valid settings for TAPIMode are:

Constant	Description
xtmData	TAPI immediately enters the connected state and opens the serial port.
xtmVoice	This mode allows you to play and record wave files over the connection.

Remarks

Default: xtmData

To use TAPI only to establish connections, TAPIMode must be set to xtmData. To support wave files and DTMF (Dual Tone Multiple Frequency), TAPIMode must be set to xtmVoice.

TAPI itself doesn't implement any of the features necessary for controlling serial ports and telephony devices. The TAPI architecture dictates that the low-level, physical services are provided by a TAPI Service Provider (TSP).

Even if TAPI is properly installed, it will not function unless a service provider is also installed. TSP modules are typically provided by telephony vendors along with their telephony hardware. Windows 95/98 and Windows NT 4.0 install a general-purpose service provider named UNIMDM.TSP, which provides basic dial and answer support for modems. It is this service provider that makes TAPI available to communications programs in Windows 95/98, Windows NT 4.0, and Windows 2000. The lack of this service provider is what makes TAPI less likely to be useful in other Windows environments (which may have TAPI, but don't have a general purpose modem service provider).

Since UNIMDM.TSP is the service provider that your application is most likely to encounter, it's worth noting a few of its limitations here.

UNIMDM does not provide support for caller identification (caller ID). The CallerID property of the APAXPort control always returns an empty string when using UNIMDM.

UNIMDM does not support no dialtone detection. TAPI will attempt to dial whether a dialtone is detected or not.

TAPI Voice support is only available with the Unimodem/V TSP (TAPI Service Provider) and the Unimodem/5 TSP. Unimodem/V is installed by default in Windows 95OSR2, Windows 98 and Windows ME. Unimodem/5 is installed by default in Windows 2000. A voice modem that is TAPI compliant and accepts the AT+V or AT#V command set is also required. Unimodem/V is available for download from Microsoft's web site for Windows 95. A list of voice modems that Microsoft has tested is included in the Unimodem/V and Unimodem/5 installation's Readme. TAPI Voice support is not available in Windows NT 4.0 unless the modem manufacturer provides a voice-enabled TSP.

You cannot assume UNIMODEM/V is installed on your user's machines since it was released after the initial release of Windows 95.

XOffChar property

Description

Determines the character that is sent to disable remote sending when software flow control is active. Read/write.

Data type

`Integer`

Syntax

expression.**XOffChar** [= *value*]

expression must reference an **APAXPort.**

Remarks

Default: ASCII 19 (^S)

Software flow control almost universally uses the XOff (ASCII 19) character to suspend transmission, and this is the default character used by APAX. If you should encounter a device that requires a different character, you can use XOffChar to set it.

See also

SWFlowOptions, XOnChar

XOnChar property

Description

Determines the character that is sent to enable remote sending when software flow control is active. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`XOnChar`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Default: ASCII 17 (^Q)

Software flow control almost universally uses the XOn (ASCII 17) character to enable transmission, and this is the default character used by APAX. If you should encounter a device that requires a different character, you can use XOnChar to set it.

See also

SWFlowOptions, XOffChar

Chapter 6: Winsock Mode

Windows includes routines for network and Internet communications. These routines are contained in DLLs which are collectively called Winsock (for WINdows SOCKets). Winsock is a Windows-specific implementation of the Berkley Sockets API. The Berkley Sockets API was developed as a protocol to allow UNIX machines to communicate with each other over networks. The concept of sockets is analogous to a telephone operator in the early days of telephones. When a call came in, the operator used a patch cord to connect the caller's socket to the socket of the person being called. Winsock does essentially the same thing. It provides a means of connecting a calling computer to a host computer so that the two can exchange information. The calling application is called a client and the host application is called a server.

Understanding Winsock

Before a connection can be established, Winsock needs to know how to find the host computer. Each network computer has an address associated with it. This address, called the IP address, is a 32-bit value that uniquely identifies the machine. Since a number like 32,147,265 is difficult to remember, network addresses are often displayed in dot notation. Dot notation specifies an IP address as a series of four bytes, each separated by a dot. For example, the TurboPower Web site address can be specified in dot notation as 209.151.79.30. Network software translates the address specified in dot notation to a real 32-bit value.

Leading zeros in a dot notation IP address (for example, ‘198.168.010.012’) causes Winsock to interpret the respective portion of the address in octal (the above IP would actually be interpreted by Winsock as ‘198.168.8.10’). APRO does not interfere with this behavior; it simply passes the entered address to Winsock as-is.

While expressing a network address in dot notation is a little better than dealing with a raw 32-bit value, it is still not particularly easy to remember. For that reason a global database gives you the capability to specify an IP address in plain text. This database, called the Domain Name Service (DNS), has text entries that correspond to IP address values. For example, the TurboPower Web site DNS entry is ‘www.turbopower.com’. If Winsock does a lookup for the host name ‘www.turbopower.com’, it gets the IP address 209.151.79.30.

Not all computers have DNS entries. A DNS entry is usually used to provide public access to a computer. Servers that are for private use only don’t publish their IP addresses.

Most software allows you to specify either the host name or the IP address in dot notation when attempting to connect to a server. To illustrate, start your favorite Web browser and type ‘www.turbopower.com’ at the address prompt. When you hit Enter, your browser displays the home page of the TurboPower Web site. Now try again, but this time type ‘209.151.79.30’ at the address prompt. Once again the browser takes you to the TurboPower Web site.

In addition to IP addresses, Winsock uses ports to specify how to connect to a remote machine. Winsock can be thought of as a trunk line with thousands of individual lines (the ports) which are used to connect machines. Some ports are considered “well-known” ports. For example, the port typically used for network mail systems (SMTP) is port 25, the telnet port is port 23, the network news server port (NNTP) is typically port 119, and so on. To see a list of well-known ports, inspect the SERVICES file in the Windows directory (for Windows NT it is in the WINNT\SYSTEM32\DRIVERS\ETC directory). The SERVICES file is a text file used by Winsock to perform port lookups (which return the service name for the specified port) and port name lookups (which return the port number for the specified service name). You can open this file in any text editor to see a list of port numbers and their corresponding service names. While these well-known ports are not set in stone, they are traditional and their use should be reserved for the services that they represent. When writing network applications, you should select a port number that is not likely to be duplicated by other applications on your network. In most cases you can choose a port number other than any of the well-known port numbers.

The IP address and port number are used in combination to create a socket. A socket is first created and then is used to establish connection between two computers. How the socket is used depends on whether the application is a client or a server. If an application is a server, it creates the socket, opens it, and then listens on that socket for computers trying to establish a connection. At this point the server is in a polling loop listening and waiting for a possible connection. A client application, on the other hand, creates a socket using the IP address of a particular server and the port number that the server is known to be listening on. The client then uses the socket to attempt to connect to the server. When the server hears the connection attempt, it wakes up and decides whether or not to accept the connection. Usually this is done by examining the IP address of the client and comparing it to a list of known IP addresses (some servers don't discriminate and accept all connections). If the connection is accepted, the client and server begin "talking" and data is transmitted.

There is one other aspect of Internet communications that should be noted. Telnet is a protocol that allows a computer to connect to a remote server via a terminal screen. When a connection is established, a telnet server sends ASCII data to the client application. The client application then displays the text on the terminal screen. Telnet applications typically use port 23.

The telnet protocol describes option negotiation (typically at the beginning of a session) and escaping of certain characters during the entire communication session. This processing is enabled automatically in APAX.

Winsock support in APAX

The Winsock implementation of the APAXPort allows you to establish a TCP/IP connection and is an implementation of the Winsock version 1.1 API. The APAXPort includes properties to allow you to set the network address, the port number, and the mode of the socket (server mode or client mode). Many of the APAXPort's properties (such as the line parameters) are not applicable when operating in Winsock mode. These properties are simply ignored when operating in Winsock mode. To put the APAXPort control in Winsock mode, simply set the DeviceType property to dtWinsock.

The APAXPort control provides access to most standard Winsock services, however the Winsock support in APAX is not intended as a full-featured Winsock implementation. Rather, it is intended to allow you to perform basic communications operations over local networks or over the Internet. Certain concessions were made (such as allowing only one client connection to a server socket) to allow the Winsock implementation to fit into the existing APAX communications model.

Winsock References

Following is a list of the APAXPort control's properties, methods, and events that pertain to Winsock functionality. This is only a subset of the functionality of the APAXPort functionality. Additional properties, methods, and events are introduced in other chapters.

Properties

WinsockAddress	WinsockPort
WinsockMode	WsTelnet

Methods

OnWinsockAccept	OnWinsockDisconnect	OnWinsockGetAddress
OnWinsockConnect	OnWinsockError	

6

Events

WinsockConnect	WinsockListen

Reference section

OnWinsockAccept event

Description

Defines an event that is fired when a client attempts to connect to a server.

Syntax

```
Private Sub expression_OnWinsockAccept(
  ByVal Addr as String, Accept as Boolean)
```

Part	Description	Data Type
expression	References the **APAXPort** object that fired the event	**APAXPort**
Addr	The network address of the client	**String**
Accept	Allows acceptance or refusal of the connection	**Boolean**

Remarks

This event is generated when an application is acting as a server (WinsockMode = wsServer) and a client application attempts a connection. Addr is the network address of the client. To accept the connection, set Accept to True. To refuse the connection, set Accept to False.

This event is not fired when the application is acting as a client.

See also

WinsockMode, OnWinsockConnect, OnWinsockDisconnect, OnWinsockError

OnWinsockConnect event

Description

Defines an event that is fired when a Winsock connection is established.

Syntax

`Private Sub` *`expression`*`_`**`OnWinsockConnect()`**

`expression` `refers to the` **`APAXPort`** `that fired the event.`

Remarks

When an application is operating as a client (WinsockMode = wsClient) it usually attempts to connect to a server. This event is generated when the server accepts the connection.

This event is not generated when the application is acting as a server.

See also

OnWinsockAccept, OnWinsockDisconnect, OnWinsockError, WinsockMode

OnWinsockDisconnect event

Description

Defines an event that is fired when a Winsock connection is dropped.

Syntax

`Private Sub` *`expression`*`_`**`OnWinsockDisconnect()`**

`expression` `refers to the` **`APAXPort`** `that fired the event.`

Remarks

A connection can be dropped as the result of an error or when a transmission is complete and one end terminates the connection.

If WinsockMode = wsServer, OnWinsockDisconnect is generated when the client is disconnected. If WinsockMode = wsClient, this event is generated when the connection is lost.

See also

OnWinsockAccept, OnWinsockConnect, OnWinsockError, WinsockMode

OnWinsockError event

Description

Defines an event that is fired when Winsock error occurs.

Syntax

Private Sub *expression*_**OnWinsockError(ByVal** *ErrorCode* **as Integer)**

Part	**Description**	**Data Type**
expression	References the **APAXPort** object that fired the event	**APAXPort**
ErrorCode	Indicates the specific error that occurred	**Integer**

Remarks

Refer to the Error Handling table for a comprehensive list of all possible error codes.

See also

WinsockMode, OnWinsockConnect, OnWinsockDisconnect

OnWinsockGetAddress event

Description

Defines an event that is fired to obtain network address information.

Syntax

```
Private Sub expression_OnWinsockGetAddress(
  Address as String, Port as String)
```

Part	Description	Data Type
expression	References the **APAXPort** object that fired the event	**APAXPort**
Address	The network address	**String**
Port	The Winsock port	**String**

Remarks

This event provides a mechanism for obtaining a network address and Winsock port when establishing or listening for Winsock connection. OnWinsockGetAddress is fired after calling either WinsockConnect or WinsockListen. If an event handler is defined, the WinsockAddress and WinsockPort properties are updated prior to creating the socket.

See also

WinsockAddress, WinsockConnect, WinsockListen, WinsockPort

WinsockAddress property

Description

Specifies the network address used to make a Winsock connection. Read/write.

Data type

`String`

Syntax

`expression``.`**`WinsockAddress`**`[= `*`value`*`]`

`expression` `must reference an` **`APAXPort.`**

6

Remarks

The WinsockAddress property accepts the IP address in dot notation (209.151.79.30) or as a host name (telnet.turbopower.com). If a host name is used, APAX does a DNS lookup to determine whether a DNS entry exists for the host name. If an IP address can be found, the port is opened.

Do not add leading zeros in dot notation addresses (e.g. 209.151.210.030). Leading zeros will cause the number to be interpreted as an octal value.

See also

DeviceType, OnWinsockGetAddress, WinsockConnect, WinsockMode, WinsockPort

WinsockConnect method

Description

Attempts to establish a TCP/IP connection to a network server.

Syntax

`expression``.`**`WinsockConnect()`**

`expression` `must reference an` **`APAXPort.`**

Remarks

Calling WinsockConnect sets the DeviceType to dtWinsock, the WinsockMode to wsClient, and attempts to establish a connection with the server specified by WinsockAddress and WinsockPort. The OnGetWinsockAddress event is fired to obtain address and port if desired. If the port is already open, the port is first closed and then reopened. Once a connection has been successfully established, the OnWinsockConnect event is fired. The port can be closed by calling the Close method.

Example

The following example creates a component at run time and establishes a connection to the telnet server at turbopower.com:

```
Dim MyPort as APAXPort
...
Set Myport = new APAXPort
MyPort.WinsockAddress = "turbopower.com"
MyPort.WinsockPort = "telnet"
MyPort.WinsockConnect
  'transmit/receive data with server
MyPort.Close
```

See also

Close, DeviceType, OnWinsockConnect, WinsockAddress, WinsockMode, WinsockPort

WinsockListen method

Description

Listens for possible connections.

Syntax

expression.**WinsockListen()**

expression must reference an **APAXPort.**

Remarks

Calling WinsockListen sets the DeviceType to dtWinsock, the WinsockMode to wsServer, and listens for possible connections on the Winsock port specified by the WinsockPort property. The OnGetWinsockAddress event is fired to obtain the Winsock port if desired. If the port is already open the port is first closed and then reopened. When a connection attempt is detected, the OnWinsockAccept event is fired to accept or deny the connection. The port can be closed by calling the Close method.

Example

The following example creates a component at run time and listens for telnet connections:

```
Dim MyPort as APAXPort
...
Set Myport = new APAXPort
MyPort.WinsockPort = "telnet"
MyPort.WinsockListen
  'transmit/receive data with server
MyPort.Cmlose
```

See also

Close, DeviceType, OnWinsockAccept, WinsockMode, WinsockPort

WinsockMode property

Description

Determines whether the application operates as a server or a client. Read/write.

Data type

TWsMode

Syntax

expression.**WinsockMode** [= *value*]

expression must reference an **APAXPort.**

Settings

Valid settings for WinsockMode are:

Constant	Description
wsClient	Application operates as a client
wsServer	Application operates as a server

Remarks

Default: wsClient

This property has no effect unless the DeviceType property is dtWinsock.

In server mode, the application listens for possible connections on the port specified in the WinsockPort property when the WinsockListen method is called.

In client mode, the client attempts to connect to the server at the address specified in the WinsockAddress property when the WinsockConnect method is called.

See also

DeviceType, WinsockAddress, WinsockConnect, WinsockListen, WinsockPort

WinsockPort property

Description

Specifies the Winsock port used to establish a network connection. Read/write.

Data type

`String`

Syntax

`expression.`**`WinsockPort`**`[= value]`

`expression` `must reference an` **`APAXPort.`**

Remarks

WinsockPort is the Winsock port on which to connect (for a client application) or on which to listen (for a server application). WinsockPort accepts the port as a string representation of an integer or a service name (e.g. telnet). If a service name is used, Winsock performs a lookup to map service name with a port number. For a list of service names and their corresponding port numbers, see the SERVICES file in the Windows directory (for Windows NT it is in the WINNT\SYSTEM32\DRIVERS\ETC directory).

See also

DeviceType, OnWinsockGetAddress, WinsockAddress, WinsockConnect, WinsockListen, WinsockMode

WsTelnet property

Description

Determines whether telnet processing is enabled. Read/write.

Data type

`Boolean`.

Syntax

`expression``.`**`WsTelnet`**`[=` *`value`*`]`

expression `must reference an` **`APAXPort`**`.`

6

Remarks

Default: True

The telnet protocol describes option negotiation and escaping of certain characters. If the client or server you are communicating with does not support telnet processing, you should set WsTelnet to False prior to establishing a connection. Otherwise it may appear that your data is being corrupted because telnet processing modifies the data stream.

Chapter 7: Terminal Window and Emulations

Traditionally, a terminal is a hardware device that communicates with a host computer. The host computer sends data and special control sequences to the terminal and the terminal knows how to process them. The terminal also accepts keystrokes and passes them on to the host computer. The keystrokes are comprised of similar control sequences and actual data. The premise that allows for successful communication between the terminal and the host is that they use the same control sequence translation scheme for the control sequences. The translation scheme defines the terminal type and the control sequences are commonly referred to as escape sequences. Some of the more common terminal types are TTY (teletype), VT100, VT52 and VT220. These dedicated hardware terminals are an older technology and their existence is diminishing. In their place is a software oriented approach that emulates the terminal's behavior. The software approach is known as terminal emulation. APAX provides integrated support for three types of emulation: TTY, VT100, and VT52. Before discussing these emulations in further detail, we need to address a few relevant topics that apply to emulations in general. These topics include escape sequences, buffering, terminal parsing, keyboard mapping, and character set mapping.

Escape sequences

A terminal will transmit and receive two types of data: literal characters and special control sequences. Control sequences are nothing more than one or more characters that have a special meaning to the terminal. The control sequences are also known as escape sequences since the vast majority of them begin with the escape <ESC> character. In the case of literal characters, the process is simple. If a terminal receives a literal character, it processes that character without modification. Similarly, the terminal transmits all literal characters without modification. Escape sequences differ significantly. If a terminal receives an escape sequence, it must interpret the escape sequence, translate it into a command, and perform the command. When a terminal transmits an escape sequence, the characters are transmitted as literal characters. It is the responsibility of the remote computer to recognize and translate these sequences. Terminals and software emulations of terminals are defined by a clear and concise table of escape sequences that perform various operations. These escape sequences may instruct the receiving computer to clear the terminal window, position the cursor at a new location, change the font attributes, or even select characters from a different character set. This requires some intelligence on the receiving computer's part. The receiving computer must intercept and parse the incoming data stream. If the received character is not part of an escape sequence, the receiving terminal simply passes the character on to the computer. If the receiving computer recognizes the character as the beginning of an escape sequence, it must extract the following characters that comprise the remainder of the escape sequence from the input stream. It then translates them into a command and performs this command on the receiving computer's terminal window. In summary, it is the escape sequences and their interpretations that define an emulation type.

Buffering

A terminal window has a fixed number of rows and columns. Typically, a terminal has 24 rows and 80 columns. This typical terminal is capable of displaying exactly 1,920 characters at a given time. Once the terminal window has received 1,920 characters, additional characters received force the terminal window to scroll the contents up line by line to accommodate newer data. How then is the user able to view data that has scrolled off the screen? The answer is in buffering. APAX maintains two internal buffers. One buffer is used to store the contents of the terminal window display. An additional larger buffer is used to store character lines received prior to the lines that are currently displayed in the terminal window. If the user wants to scroll back in the terminal display, data is retrieved from this larger scroll back buffer. APAX allows you to specify the size (in lines) of this scroll back buffer via the ScrollbackRows property. Both of these buffers maintain a set of matrices: one for the displayable characters, one for the attributes (bold, underline, invisible, and so on), one for the color of the text, one for the color of the background, and finally one for the character set identifiers.

Terminal parsing

Terminal parsing refers to the receiving computer's responsibility of monitoring the input data stream, and parsing out the escape sequence. The internal terminal parser converts the escape sequences to commands and executes the command on the terminal window. APAX's internal parser automatically recognizes these escape sequences based on the emulation mode currently selected in the Emulation property, and processes the translated commands accordingly. No user or programmer intervention is required.

Keyboard mapping

All terminals have two input sources: the data stream received at the associated COM port, and user input from the local keyboard. Keyboard mapping requires the emulation to perform a translation of keystrokes into the corresponding escape sequence that will be sent to the host computer. For example, with the standard VT100 mappings provided with APAX, the up arrow key on the PC keyboard is mapped to DEC_UP, the name for the up arrow on the VT100 keyboard. The terminal key name is looked up in a table to obtain the control sequence that needs to be sent to the host computer. APAX maintains internal keyboard mapping tables to store the information required to map a PC keystroke into the control sequence for the supported emulations that has to be sent to the host computer.

Character set mapping

A terminal window is capable of displaying characters from multiple character sets. The character set from which to extract a single character is selected via an escape sequence. For each character set, there may be a different glyph for each character. The classic example is that of the VT100 terminal. This terminal has several character sets, of which two are the most commonly used: the USASCII character set and the special graphics character set. To take as an example, the character 'm' is displayed as a lower-case m in the USASCII character set, but is displayed as a line draw lower left corner (the glyph that looks like an L) in the special graphics character set. APAX maintains internal keyboard mapping tables that automatically select characters from the correct set.

TTY, VT100, and VT52 Emulations

APAX supports TTY (teletype), VT100, and VT52 software emulations. The TTY emulation contains no escape sequences. In TTY mode, the terminal simply echoes typed keys to the host PC and all characters received from the associated serial port are sent directly to the terminal window. VT100 is a mature and well-defined standard. APAX provides full support of the VT100 emulation standard. In addition to the fundamental VT100 standard, APAX also provides support for two extensions of the standard. The first extension allows you to specify the foreground and background colors, and the second extension allow you to insert, delete, and erase characters and lines. VT52 is simply a subset of the VT100 emulation standard and also is fully supported by APAX. With that said, all that remains is a clear definition of the VT100 escape sequences to understand the terminal emulation functionality supported by APAX. The following table serves this purpose:

Control Sequence	Terminal Mode	Description
ENQ ($05)	VT100/52	Generate answerback message
BEL ($07)	VT100/52	Sound bell
BS ($08)	VT100/52	Backspace, i.e. cursor left
HT ($09)	VT100/52	Move to next horizontal tab stop
LF ($0A)	VT100/52	Line feed or new line
VT ($0B)	VT100/52	Processed as LF
FF ($0C)	VT100/52	Processed as LF
CR ($0D)	VT100/52	Move to position 1 on current line
SO ($0E)	VT100	Select G1 character set
SI ($0F)	VT100	Select G0 character set
CAN ($18)	VT100/52	Cancel current escape sequence
SUB ($1A)	VT100/52	Cancel current escape sequence
Esc # 3	VT100	Line is double height, top half
Esc # 4	VT100	Line is double height, bottom half
Esc # 5	VT100	Line is single width, single height

Control Sequence	Terminal Mode	Description
Esc # 6	VT100	Line is double width, single height
Esc # 8	VT100	Fill screen with E's
Esc (A	VT100	Set G0 to UK charset
Esc (B	VT100	Set G0 to US charset
Esc (0	VT100	Set G0 to special linedraw charset
Esc (1	VT100	Set G0 to alternate ROM charset
Esc (2	VT100	Set G0 to alternate ROM LD charset
Esc) A	VT100	Set G1 to UK charset
Esc) B	VT100	Set G1 to US charset
Esc) 0	VT100	Set G1 to special linedraw charset
Esc) 1	VT100	Set G1 to alternate ROM charset
Esc) 2	VT100	Set G1 to alternate ROM LD charset
Esc 7	VT100	Save cursor and attributes
Esc 8	VT100	Restore cursor and attributes
Esc <	VT100	Enter ANSI mode (ignored)
Esc =	VT100	Enter application keypad mode
Esc >	VT100	Enter numeric keypad mode
Esc D	VT100	Index
Esc E	VT100	Next line
Esc H	VT100	Set tab stop at cursor
Esc M	VT100	Reverse index
Esc [Pn @	VT100 enh	Insert characters at cursor
Esc [Pn A	VT100	Cursor up
Esc [Pn B	VT100	Cursor down
Esc [Pn C	VT100	Cursor right

Control Sequence	**Terminal Mode**	**Description**
Esc [Pn D	VT100	Cursor left
Esc [Pr; Pc H	VT100	Cursor position
Esc [Ps J	VT100	Erase part of display to cursor
Esc [Ps K	VT100	Erase part of display from cursor
Esc [Pn L	VT100 enh	Insert lines
Esc [Pn M	VT100 enh	Delete lines
Esc [Pn P	VT100 enh	Delete characters at cursor
Esc [Pn X	VT100 enh	Erase characters at cursor
Esc [Ps c	VT100	What are you?
Esc [Pr; Pc f	VT100	Cursor position
Esc [Ps g	VT100	Clear tab stops
Esc [Ps h	VT100	Set mode
Esc [Ps l	VT100	Reset mode
Esc [Ps; …;Ps m	VT100	Set attributes, including color
Esc [Ps n	VT100	Request terminal report
Esc [Ps; …;Ps q	VT100	Set LEDs
Esc [Pt; Pb r	VT100	Set scrolling region (top, bottom row)
Esc [2; Ps y	VT100	Invoke confidence test
Esc c	VT100	Reset
Esc A	VT52	Cursor up
Esc B	VT52	Cursor down
Esc C	VT52	Cursor right
Esc D	VT52	Cursor left
Esc F	VT52	Set special character set
Esc G	VT52	Set ASCII character set
Esc H	VT52	Cursor to home
Esc I	VT52	Reverse line feed

Control Sequence	**Terminal Mode**	**Description**
Esc J	VT52	Erase to end of screen
Esc K	VT52	Erase to end of line
Esc Y	VT52	Direct cursor address
Esc Z	VT52	Identify
Esc <	VT52	Enter ANSI mode
Esc =	VT52	Enter alternate keypad mode
Esc >	VT52	Exit alternate keypad mode

Note: Use the escape sequence <Esc>[2l to switch into VT52 mode.

Terminal and Emulator References

Following is a list of the APAXPort control’s properties, methods, and events that pertain to terminal and emulator facilities. This is only a subset of the functionality of the APAXPort functionality. Additional properties, methods, and events are introduced in other chapters.

Properties

- CaptureFile
- CaptureMode
- Color
- Columns
- Emulation
- Font
- Rows
- ScrollbackEnabled
- ScrollbackRows
- TerminalActive
- TerminalBlinkTime
- TerminalHalfDuplex
- TerminalLazyByteDelay
- TerminalLazyTimeDelay
- TerminalUseLazyDisplay
- TerminalWantAllKeys
- Visible

Methods

- Clear
- ClearAll
- CopyToClipboard
- GetAttributes
- GetLine
- SetAttributes
- SetLine
- TerminalSetFocus
- TerminalWriteString
- TerminalWriteStringCRLF

Events

- OnCursorMoved

Reference Section

CaptureFile property

Description

Defines the name of the file where the terminal writes captured data. Read/write.

Data type

`String`

Syntax

`*expression*.**CaptureFile**[= *value*]`

`*expression* must reference an **APAXPort.**`

Remarks

Default: "APAX.CAP"

It is possible to change the name of the file where captured data is sent while data is being captured. Internally, the terminal component sets the CaptureMode property to cmOff, changes the value of the CaptureFile property to the new filename, and then sets the CaptureMode property to cmOn again. This means that the file named by the new value of CaptureFile is created afresh: the old file, if it exists, is overwritten. If you wish to append to the file with the new name, you will need to manually set CaptureMode to cmOff, set the value of CaptureFile to the new filename, and then set CaptureMode to cmAppend.

See also

CaptureMode

CaptureMode property

Description

Defines whether the data received by the terminal is captured to file. Read/write.

Data type

`TApxCaptureMode`

Syntax

`*expression*.**CaptureMode** [= *value*]`

`expression` `must reference an` **`APAXPort.`**

Settings

The following are valid settings for the CaptureMode property.

Constant	Description
cmOff	No capturing of data
cmOn	Capture new data to new file
cmAppend	Capture data, appending to old file

Remarks

Default: cmOff

The CaptureMode property has only two values on reading: whether the terminal is capturing data (cmOn will be returned) or not (cmOff will be returned).

It has three possible values on writing: cmOn, cmOff, or cmAppend. If the value written is cmAppend and the current value is cmOff, the capture file is opened in non-sharing mode for appending data and the value of the Capture property is then set to cmOn. Note that if the file doesn't exist at the time the property is set, it will be created. If the value written is cmAppend and the current value is cmOn, the assignment is ignored and nothing happens.

If the value written is cmOn and the current value is cmOff, the file is created. If it existed prior to the assignment, it will be overwritten.

The name of the file where captured data is written is designated by the CaptureFile property.

All data coming into the terminal is written to the file without any effort being made to parse it or identify terminal control sequences. Thus for a complex terminal emulation the data in the capture file will consist of intermingled text and terminal control sequences.

If the CaptureFile property has not been set to the name of a file (i.e., it is the empty string), setting Capture to cmOn or cmAppend will have no effect. The attempt will be ignored and no error will be raised. If the CaptureFile property has been set, an attempt is made to create or open the file so named. This operation can of course fail for any of a number of different reasons.

See also

CaptureFile

Clear method

Description

Clears the terminal display.

Syntax

`expression.Clear()`

`expression must reference an APAXPort.`

Remarks

When clear is called the terminal will internally scroll the window up by Rows lines. This means that the current display will scroll into the non-visible portion of the internal buffer and can still be viewed if ScrollbackEnabled is set to True.

See also

ClearAll

ClearAll method

Description

Clears the entire internal scrollback buffer including the terminal display window.

Syntax

expression.**ClearAll()**

expression must reference an **APAXPort.**

Remarks

When the internal scrollback buffer is cleared, the terminal sets all characters in the buffer to the space character, and sets all attribute values so that they are equivalent to normal text. Additionally, all background colors are set to the Color property setting, and all foreground colors are set to the color of the terminal's Font.Color property.

See also

Clear

Color property

Description

Defines the background color of the terminal window. Read/write.

Data type

TColor

Syntax

expression.**Color**[= *value*]

expression must reference an **APAXPort.**

Remarks

The Color property defines the default background color of the terminal window. Individual row and column attributes take precedence over this property setting.

Example

The following example sets the background color of the terminal window to cyan.

```
  Apax1.Color = ColorConstants.vbCyan
```

See also

SetAttributes

Columns property

Description

Defines the number of columns across the terminal screen. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`Columns`**`[= `*`value`*`]`

`expression`` must reference an `**`APAXPort.`**

Remarks

Default: 80

The Columns property is the number of columns displayed by the terminal screen. For the VT100 terminal, for example, there are two possible values: 80 and 132.

If the Columns property is changed, it is checked to ensure that the value is at least 2, otherwise an exception is raised.

Changing the Columns property for an existing terminal screen does not clear the data being displayed by the screen; you will need to do this as a separate step. The positions of any horizontal tab stops are maintained, except for those that lie outside the new value for Columns. However, any scrolling region is discarded, and the cursor is reset to the home position of the screen.

See also

Rows

CopyToClipboard method

Description

Copies the marked block to the clipboard.

Syntax

```
expression.CopyToClipboard()

expression must reference an APAXPort.
```

Remarks

This routine copies the currently marked block to the Window's clipboard. The block is copied in CF_TEXT format, where a carriage return/line feed follows each line.

To clear the internal scrollback buffer, the terminal sets all characters in the buffer to the space character, and sets all attribute values so that they are equivalent to normal text. Additionally, all background colors are set to the Color property setting, and all foreground colors are set to the color of the terminal's Font.Color property.

When the ScrollbackEnabled property is set to False, only the visible contents of the terminal window can be marked. When the ScrollbackEnabled property is set to True, the visible contents can be marked, and by moving the cursor above or below the terminal window, the window can be scrolled to allow any part of the scrollback buffer to be marked.

Note: Although the CopyToClipboard method will copy the marked text to the clipboard, you should be aware that what you see may not be what you get because APAX supports different character sets. In other words, with certain emulations, the glyphs that are displayed on the terminal display may seem to have no connection with the characters that are actually there. For example, when using the USASCII character set, the character 'm' will be displayed as a lower case 'm'. However, when using the special graphics character set, the character 'm' is rendered as the lower left corner of the line draw set. The problem is that when you copy the marked text to the clipboard, you will lose the character set definition for each character. Hence, you will just get the character 'm' in the clipboard and not know whether it really was shown as an 'm' or some other glyph.

See also

Clear, ClearAll

Emulation property

Description

Determines the emulation to use (TTY or VT100). Read/write.

Data type

`TApxTerminalEmulation`

Syntax

`expression``.`**`Emulation`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Settings

The following settings are valid for the Emulation property:

Constant	Description
teTTY	TTY emulation
teVT100	VT100 emulation

Remarks

When the Emulation property is set to teTTY, the terminal emulates a teletype terminal, one that doesn't support any terminal control sequences and one that merely displays every character received. Similarly there is no conversion of keystrokes either: if a key for a displayable character is pressed, that character is sent to the host without interpretation. If Emulation is set to VT100, then the terminal encapsulates the knowledge of the standard VT100 escape sequences and to which command they refer. VT52 is a subset of VT100. To enable VT52 emulation, then, set Emulation to VT100.

Font property

Description

Determines the font displayed by the terminal window. Read/write.

Data type

TFont

Syntax

expression.**Font**[= *value*]

expression must reference an **APAXPort.**

Remarks

The font assigned to this property is scaled to conform to the restrictions imposed by the terminal window's height and width, and the settings of the Rows and Columns properties.

See also

Color

GetAttributes method

Description

Returns an Integer that represents the attributes of the specified row and column.

Syntax

expression.**GetAttributes(aRow, aCol(**

Part	Description	Data Type
expression	An expression that returns an **APAXPort** object	**APAXPort**
aRow	Specifies the terminal window row	**Integer**
aCol	Specifies the terminal window column	**Integer**

Settings

The return value of GetAttributes should be masked with the global constant tcaxMask to determine the actual attributes. The following table depicts all of the possible attribute settings.

Constant	Description
tcaxNone	No special attributes defined
tcaxBold	Bold font
tcaxUnderline	Underline font
tcaxStrikethrough	Strikethrough font
tcaxBlink	Blinking font
tcaxReverse	Foreground and background colors are reversed
tcaxInvisible	Foreground color equal background color

Remarks

GetAttributes enables you to determine the attributes for characters displayed by the terminal at the specified row and column (aRow, aCol). Both aRow and aCol are one-based: the home position of the terminal display is at row 1 column 1. Note, however, that if you have a scrollback buffer that aRow can take on negative values, as well, to identify non-visible rows in the scrollback buffer.

The result value is an integer that can be tested for various attribute settings as in the following example.

Example

The following example determines whether the character at row 2, column 3 is bold:

```
Attr = GetAttributes(2,3)
If (Attr and tcaxMask) > 0 Then
  'font is bold
Else
  'font is not bold
End If
```

See also

SetAttributes

GetLine property

Description

Returns the text data for a row in the display. Run-time only.

Syntax

expression.**GetLine(Index)**

Part	Description	Data Type
expression	An expression that returns an **APAXPort** object	**APAXPort**
Index	Specifies the line number to return	**Integer**

Remarks

7

GetLine returns the characters that make up a row in the terminal. Index is one-based: the home position of the terminal display is at row 1 column 1. Note, however, that if you have a scrollback buffer that Index can take on negative values, as well, to identify non-visible rows in the scrollback buffer.

Caution: GetLine returns the character values that make up a row. For some terminals, the glyph you see on the terminal for a particular character value is not only based on the character value itself, but also on the character set that is being used to display that character. For example, on a VT100 terminal, if the character ‘m’ is displayed using the USASCII character set, you will see the usual lower case ‘m’ glyph on the display. However, if the same character ‘m’ is displayed using the Special Characters character set, you will see the lower left corner linedraw glyph (the one that looks like an ‘L’). All the Line property will return is the ‘m’ character value at that particular column position.

See also

GetAttributes, SetLine

OnCursorMoved event

Description

Defines an event that is fired when the cursor changes position.

Syntax

```
Private Sub expression_OnTerminalCursorMoved(
  ByVal aRow As Long, ByVal aCol As Long)
```

Part	Description	Data Type
expression	References the **APAXPort** that fired the event	**APAXPort**
aRow	New row of the cursor	**Long**
aCol	New column of the cursor	**Long**

Remarks

This event is triggered whenever the cursor's position (column or row) changes. The parameters aRow and aCol correspond to the cursor's new row and column position within the terminal window.

See also

Rows, Columns

Rows property

Description

Defines the number of rows down the terminal display. Read/write.

Data type

`Integer`

Syntax

`expression.Rows [= value]`

expression must reference an **APAXPort.**

Remarks

Default: 24

The value of the Rows property is the number of standard-sized characters that can be written vertically on the terminal display. In general, it is a value such as 24 or 25. If the original terminal supports double-height characters then the value of Rows still reflects that for standard-sized characters, not the double-height ones.

Notice that Rows is the number of rows on the original terminal, not the number of rows in the scrollback buffer. The terminal display will consist of Rows lines, with Columns characters in each.

Altering the value of Rows may cause the underlying buffer to be resized. The terminal will attempt to save as much of the original data as possible during the resize operation. The data will be preserved from the bottom of the terminal upwards. In other words, if you reduce the number of rows from 20 to 15, say, you will see the bottom 15 rows of the original display after the operation completes, not the top 15.

Setting the value of Rows to less than that supported by the original terminal itself is liable to produce awkward looking displays, since the host computer will assume that the terminal is the correct size and position text accordingly.

The rows in the terminal display are counted from 1, with the top row of the terminal being row 1.

See also

Columns

ScrollbackEnabled property

Description

Defines whether the terminal is in scrollback mode. Read/write.

Data type

`Boolean`

Syntax

`expression.`**`ScrollbackEnabled`**`[= value]`

`expression must reference an `**`APAXPort.`**

Remarks

Default: False

If ScrollbackEnabled is set to True, the terminal is placed into scrollback mode. In this mode, keystrokes are no longer translated into their terminal equivalents and instead serve to navigate through the scrollback buffer. Hence, in scrollback mode, the Page Up/Page Down keys will move the user through the scrollback buffer, as will the standard arrow keys.

If scrollback mode is activated, the terminal will also no longer receive data from the serial device. It is recommended that you impose flow control on the serial port when you switch into scrollback mode (either send an XOFF character or drop a hardware flow control signal), to help avoid the dispatcher's input buffer overflowing. The terminal does not do this itself. Imposing flow control helps keep the terminal data stable to allow the user to navigate though the scrollback buffer in a profitable manner. Obviously, when you leave scrollback mode, you would end the flow control condition to allow more data to come through and be processed.

See also

ScrollbackRows

ScrollbackRows property

Description

Defines the number of rows in the scrollback buffer. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`ScrollbackRows`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Default: 200

The scrollback buffer consists of the visible part of the terminal display, together with the previous data that has scrolled off the top of the terminal display. In general you would set ScrollbackRows such that you could hold four or five screens' worth of previous data.

The value of the ScrollbackRows property must be greater than or equal to the value of Rows. If you attempt to set ScrollbackRows to a value less than Rows, the new value is adjusted to be equal to Rows. No error is generated in this situation. If the original terminal supports double-height characters then the value of ScrollbackRows still reflects that for standard-sized characters, not the double-height ones.

Altering the value of ScrollbackRows may cause the underlying buffer to be resized. The terminal will attempt to save as much of the original data as possible during the resize operation. If the value of ScrollbackRows is reduced the data is removed from the top of the buffer rather than the bottom.

The rows in the terminal display are counted from 1, with the top row of the terminal being row 1. The rows above the actual terminal display in the scrollback area are counted backwards from 1. Hence, the row above the top row of the actual terminal display is row 0, the one above that row –1, and so on.

See also

ScrollbackEnabled

SetAttributes method

Description

Sets the font attributes at the specified row and column to the value specified by Value.

Syntax

expression.**SetAttributes(aRow, aCol, Value)**

Part	Description	Data Type
expression	An expression that returns an **APAXPort** object	**APAXPort**
Row	Specifies the terminal window row	**Integer**
Column	Specifies the terminal window column	**Integer**
Value	Determines the actual font attributes	**Integer**

Settings

Value can be set to any combination of the following.

Constant	Value	Description
tcaxNone	**0**	No special attributes defined
tcaxBold	**1**	Bold font
tcaxUnderline	**2**	Underline font
tcaxStrikethrough	**4**	Strikethrough font

Constant	Value	Description
tcaxBlink	**8**	Blinking font
tcaxReverse	**16**	Foreground and background colors are reversed
tcaxInvisible	**32**	Foreground color equal to background color

Remarks

SetAttributes enables you to set the attributes for characters displayed by the terminal at the specified row and column (aRow, aCol). Both aRow and aCol are one-based: the home position of the terminal display is at row 1 column 1. Note, however, that if you have a scrollback buffer that aRow can take on negative values, as well, to identify non-visible rows in the scrollback buffer.

Example

The following example sets the character attributes of row 5, column 3 to reverse bold:

```
  SetAttributes(5, 3, tcaxBold + tcaxReverse)
```

However, do notice that this direct manipulation is fairly inefficient.

See also

GetAttributes

SetLine method

Description

Sets the text data for a row in the display. Run-time only.

Syntax

expression.**SetLine(Index, Value)**

Part	Description	Data Type
expression	An expression that returns an **APAXPort** object	**APAXPort**
Index	Specifies the line number to return	**Integer**
Value	Specifies the string to write	**String**

Remarks

SetLine sets the string that makes up a row in the terminal. Index is one-based: the home position of the terminal display is at row 1 column 1. Value determines the string that will be displayed in the terminal window at line Index. Note, however, that if you have a scrollback buffer that Index can take on negative values, as well, to identify non-visible rows in the scrollback buffer.

Caution: SetLine sets the character values that make up a row. For some terminals, the glyph you see on the terminal for a particular character value is not only based on the character value itself, but also on the character set that is being used to display that character. For example, on a VT100 terminal, if the character ‘m’ is displayed using the USASCII character set, you will see the usual lower case ‘m’ glyph on the display. However, if the same character ‘m’ is displayed using the Special Characters character set, you will see the lower left corner linedraw glyph (the one that looks like an ‘L’). All the Line property will return is the ‘m’ character value at that particular column position.

See also

GetAttributes, GetLine

TerminalActive property

Description

Determines whether the terminal is accepting serial and keyboard events. Read/write.

Data type

`Boolean`

Syntax

*`expression.`***`TerminalActive`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Setting TerminalActive to True causes the terminal to start processing serial and keyboard data and to display this information in the terminal window.

See also

TerminalSetFocus

TerminalBlinkTime property

Description

Defines the time in milliseconds between cycles for blinking text. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`TerminalBlinkTime`**`[= `*`value`*`]`

`expression`` must reference an `**`APAXPort.`**

Remarks

Default: 500

Some terminals allow blinking text to be displayed. The TerminalBlinkTime property defines the elapsed time for a full cycle for the text being displayed, being invisible, and being displayed again.

Note that, to provide this functionality, the terminal sets up a timer to tick at this rate. A Windows timer is low-priority: if the PC is performing other work, it will seem as if the blinking text has either stopped blinking or has disappeared completely. Also, if you set the TerminalBlinkTime property too low, the terminal and emulator will spend most of their time updating the window, especially if there's a lot of blinking text.

TerminalHalfDuplex property

Description

Determines whether local data is echoed to the terminal display. Read/write.

Data type

`Boolean`

Syntax

`expression``.`**`TerminalHalfDuplex`**`[= `*`value`*`]`

`expression`` must reference an `**`APAXPort.`**

Remarks

Default: False

If TerminalHalfDuplex is False (the default), data entered at the local keyboard is displayed only if the remote host computer echoes it back.

If TerminalHalfDuplex is True, data entered at the local keyboard is automatically displayed in the terminal window. If the host computer is echoing the input data back as well, each character is displayed twice.

TerminalLazyByteDelay property

Description

Determines the number of bytes received before the display is forcibly repainted. Read/write.

Data type

`Integer`

Syntax

```
expression.TerminalLazyByteDelay[= value]
```

```
expression must reference an APAXPort.
```

Remarks

Default: 128

The APAXPort control supports a “lazy writing” mode. When this mode is active, rather than update the display every time a new character appears from the serial device, the terminal will only display new data after a certain amount of time, or after a certain number of bytes have been received, or both. This gives the terminal a more efficient and smoother “feel”. The value of TerminalLazyByteDelay defines how many bytes must be received before the terminal window is updated. The default value is a compromise between efficiently handling the display and providing timely visual feedback to the user.

See also

TerminalLazyTimeDelay, TerminalUseLazyDisplay

TerminalLazyTimeDelay property

Description

Determines the number of elapsed milliseconds before the display is forcibly repainted. Read/write.

Data type

`Integer`

Syntax

`expression.`**`TerminalLazyTimeDelay`**`[= value]`

`expression must reference an` **`APAXPort.`**

Remarks

Default: 250

The APAXPort control supports a “lazy writing” mode. When this mode is active, rather than update the display every time a new character appears from the serial device, the terminal will only display new data after a certain amount of time, or after a certain number of bytes have been received, or both. This gives the terminal a more efficient and smoother “feel”. The value of TerminalLazyTimeDelay defines how many milliseconds must pass before the terminal window is updated. The default value is a compromise between efficiently handling the display and providing timely visual feedback to the user.

See also

TerminalLazyByteDelay, TerminalUseLazyDisplay

TerminalSetFocus method

Description

Sets the terminal window as the active control.

Syntax

expression.**TerminalSetFocus()**

expression must reference an **APAXPort.**

Remarks

This method is useful to transfer focus to the terminal window. After calling this method, all keyboard strokes are sent directly to the terminal window, circumventing the need for the user to click on the terminal window.

See also

TerminalActive

TerminalUseLazyDisplay property

Description

Defines whether the terminal immediately displays new incoming data or not. Read/write.

Data type

Boolean

Syntax

expression.**TerminalUseLazyDisplay**[= *value*]

expression must reference an **APAXPort.**

Remarks

Default: True

The APAXPort control supports a "lazy writing" mode. When this mode is active, rather than update the display every time a new character appears from the serial device, the terminal will only display new data after a certain amount of time, or after a certain number of bytes have been received, or both. This gives the terminal a more efficient and smoother "feel".

If TerminalUseLazyDisplay is False, the terminal will display every incoming character as and when it arrives. If TerminalUseLazyDisplay is True, the terminal will display the new data on the screen after TerminalLazyByteDelay bytes have been received since it last updated the window, or after TerminalLazyTimeDelay milliseconds have elapsed.

The lazy writing mode only applies to data written to the terminal, either from the serial device, or from the keyboard in half duplex mode, or from data explicitly written from calling PutString (see page 111) or PutData (see page 110). If the terminal component's window is invalidated due to another window covering it and then being moved, or from the application being minimized and then restored, the terminal display is immediately repainted.

See also

TerminalLazyByteDelay, TerminalLazyTimeDelay

TerminalWantAllKeys property

Description

Defines whether the terminal hooks and retrieves all keystrokes. Read/write.

Data type

`Boolean`

Syntax

`expression``.`**`TerminalWantAllKeys`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Default: True

Part of the job of a terminal is the ability to map the PC keyboard onto a terminal keyboard. This latter keyboard might be a completely different layout than the PC keyboard and, apart from the alphabetic key section, have different keys for different host functions. The keyboard mapping should attempt to match PC keys (whether they are alt-shifted, ctrl-shifted, or whatever) onto appropriate terminal keys.

A problem that will occur is that keys like F1, F10, Enter, Tab, and so on, have a well-defined meaning in the Windows world. Normally, controls on a form would ignore these keys since they have dialog specific or application wide meanings. However, for a terminal it often makes sense to have these keys perform a terminal related function and to suppress the standard Windows meaning. If TerminalWantAllKeys is True, the terminal will attempt to hook and trap all keystrokes generated while it has focus. Hence, for example, F1 will not bring up the help system (it will not cause a WM_HELP message to be sent to the control), F10 will not activate the main menu of the application, and so on.

If TerminalWantAllKeys is False, the terminal will not perform anything special with regard to the keyboard. It will just trap WM_KEYDOWN and WM_SYSKEYDOWN messages and pass them on to the emulator for processing. Standard Windows keys will perform their usual functions.

TerminalWriteString method

Description

Writes a string to the terminal.

Syntax

`expression.`**`TerminalWriteString(`**`aStr`**`)`**

Part	Description	Data Type
expression	An expression that returns an APAXPort object.	**APAXPort**
aSt	Defines the string to write	**String**

Remarks

The string written to the terminal will go through the same steps that characters that have arrived from the serial device would go through. In other words, the characters in the string are first passed to the internal keyboard emulator, which decides what to do with them. If the emulator decides that certain characters are part of a terminal control sequence, it may appear as if the string had not fully been accepted (it would not appear on the display) when in reality it had. You can therefore use TerminalWriteString to send terminal control sequences to the terminal to alter its behavior.

The terminal window will accept a string written with TerminalWriteString at any time, even when it is actively receiving data from the serial device. Be aware that under these circumstances, the characters written with TerminalWriteString will intermingle with data from the serial device and may cause some bizarre behavior and displays.

See also

TerminalWriteStringCRLF

TerminalWriteStringCRLF method

Description

Writes a string followed with a carriage return/line feed pair to the terminal.

Syntax

expression.**TerminalWriteStringCRLF**(***aStr***)

Part	**Description**	**Date Type**
expression	An expression that returns an APAXPort object.	**APAXPort**
aSt	Defines the string to write	**String**

Remarks

The string passed to this method will have a carriage return/line feed pair added internally. The string will then be written to the terminal and will go through the same steps that characters that have arrived from the serial device would go through. In other words, the characters in the string are first passed to the internal keyboard emulator, which decides what to do with them. If the emulator decides that certain characters are part of a terminal control sequence, it may appear as if the string had not fully been accepted (it would not appear on the display) when in reality it had. You can therefore use TerminalWriteString to send terminal control sequences to the terminal to alter its behavior.

The terminal window will accept a string written with TerminalWriteStringCRLF at any time, even when it is actively receiving data from the serial device. Be aware that under these circumstances, the characters written with TerminalWriteStringCRLF will intermingle with data from the serial device and may cause some bizarre behavior and displays.

See also

TerminalWriteString

Visible property

Description

Determines whether the APAXPort control is visible. Read/write.

Data type

`Boolean`

Syntax

```
expression.Visible[= value]
```

```
expression must reference an APAXPort.
```

Remarks

The Visible property setting determines the visibility of the entire control including the tool bar, status bar, and terminal window.

See also

ShowToolBar, ShowStatusBar

Chapter 8: TAPI Devices

The Telephony Application Programming Interface (TAPI) is a collection of DLLs and a documented programming interface for centralizing and controlling telephony communications services. TAPI was developed by Microsoft primarily for Computer Telephony Integration (CTI) applications. TAPI provides the services that telephone equipment and system providers need to integrate Windows programming and telephone hardware.

TAPI also provides a smaller, though much more visible, service in managing modems as system devices. This is a tremendous boon to communications programmers. No longer do communications applications need to search serial ports for modems, try to identify modems, burden the user with questions about their modem, or try any of the other traditional approaches to supporting modems. Under TAPI, that task is handled by the operating system. Programs make a few simple TAPI calls to determine what modems are available.

Another advantage of TAPI is that applications can share serial ports. For example, assume that an application opens a TAPI device to accept incoming fax or data calls. A second application, if it also uses TAPI, can safely open the same TAPI device for an outgoing call. When the outgoing call is over, TAPI resumes monitoring incoming calls without any further action required by either application.

The major disadvantage of TAPI is that it doesn't provide support for direct, modemless connections. TAPI applications usually must still include logic for the direct opening of serial ports.

The second major disadvantage of TAPI is the difficulty of assuring proper modem configuration. TAPI modems are detected and installed by Windows during the installation process. If a new modem is added later, the instructions provided with the modem usually direct the user to run one of the Control Panel applets "Add New Hardware" or "Modems." The Add New Hardware applet found in Windows scans all available serial ports for attached modems. When it finds a modem, it sends a variety of commands to the modem and compares the responses against a large database of known modems, usually resulting in an unambiguous choice. The Modems applet (or a setup program provided by the modem vendor), can skip this detection process if the modem type is already known.

No matter how the modem is installed, the end result is that TAPI now knows everything it needs to about that modem (serial port, baud rate, and the specific configuration commands/responses). Property sheets are available to the user for changing the configuration of the modem. For example, the user can turn the speaker on or off, change the attached serial port, enable/disable flow control, and so on.

The default property values for the modem chosen by Microsoft or the modem vendor are the values that provide the best results in the widest variety of situations. These properties, however, are available to the user via the Modem applet's modem property sheets. If the user changes a critical value (say, the serial port number or perhaps flow control) it's likely that your application won't operate properly when using that modem. Unfortunately, there isn't much you can do to protect against this. The responsibility for assuring the modem is properly configured is in the user's hands, not the application.

The final major disadvantage of TAPI is that resources often do not get properly released following an abnormal termination of an application using TAPI. To counter this problem, APAX includes a TAPI crash recovery mechanism which will automatically detect this situation and release TAPI resources.

All of these issues are likely to improve over time, making TAPI the choice for current and future communications programs. The best recommendation at this point is to use TAPI for applications targeted for Windows, and make TAPI an option for all other operating system targets.

The APAX Port control provides all the services necessary for selecting TAPI devices, dialing and answering calls, and receiving status information about TAPI calls in progress. The TAPI system itself provides even more services. For more information about TAPI, refer to the following sources:

- *Windows Telephony Programming: A Developer's Guide to TAPI. (*Addison-Wesley). ISBN 0/201/63450-3.
- *TAPI Reference Manual*, included with the TAPI SDK and with the Windows 95/98 SDK.
- "Create Communications Programs for Windows 95 with the Win32 Comm API", Microsoft Systems Journal. 1994. #12. (Microsoft).
- *Programming Windows 95 Unleashed* (SAMS). ISBN 0-672-30474-0. Although only one out of 37 chapters is devoted to TAPI, it provides a nicely condensed version of much of the information in the TAPI Reference Manual, plus some helpful C++ example programs.
- The C++ sample program TAPICOMM, available on the Microsoft Developer Network CD.

TAPI Device Control from an Application

Without TAPI, the APAXPort control opens the physical serial port directly using the appropriate Windows API call, which returns a handle to that port. The APAXPort then uses the handle to send and receive data and otherwise control the serial port. Configuring, dialing, or answering the modem requires sending explicit ATXxx commands to the modem, interpreting the responses, and dealing with the myriad of differences among currently available modems.

With TAPI, an application calls the TAPIDial method to place an outgoing call or calls the TAPIAnswer method to wait for an incoming call. TAPI sends the appropriate ATXxx commands, interprets the responses, and establishes the modem connection.

Once the connection is established, TAPI's role is essentially over. TAPI remains in charge of the call until the modem connection is broken. From this point on the basic serial port methods control the port and send/receive data, just as if TAPI is not involved. The serial port is automatically opened by the APAXPort control when the modem connection is established. Several APAXPort properties are updated with appropriate information from the TAPI device, notably Baud and ComNumber.

When the modem connection is broken, the APAXPort control automatically closes the associated serial port. The serial port cannot be used for input/output unless the modem connection is re-established by the TAPI device or unless the program bypasses TAPI and opens and uses the serial port directly.

The DeviceType property determines whether the port is in charge of the physical serial port or whether TAPI is in charge of the port.

When DeviceType is xdDirect or xdWinsock, TAPI methods and properties have no effect. Only when the DeviceType property is set to sdTAPI does TAPI take charge of configuring and controlling the serial port.

TAPI Events

TAPI dials outgoing calls and waits for incoming calls in the background. Applications are informed of progress through a callback procedure. The APAXPort control installs a hidden callback and translates these progress callbacks into the following events:

OnTAPIStatus

Generated at various intervals while dialing an outgoing call or answering an incoming call. The parameters mirror the parameters passed directly to the TAPI callback. You will usually need to reference only the Message and Param1 fields. The other fields are supplied for applications that extend the services provided by APAXPort control.

OnTAPIPortOpen

Generated immediately after TAPI has established a connection and has the serial port available for data communications.

OnTAPIPortClose

Generated immediately after TAPI closes the serial port due to a broken connection.

OnTAPIConnect

Generated immediately after TAPI establishes a modem connection.

OnTAPIFail

Generated immediately after TAPI tries but fails to establish a modem connection.

OnTAPICallerID

Generated after a connection is made and both a Caller ID string and a Caller ID Name string are returned.

OnTAPIDTMF

Generated as soon as a DTMF tone is detected.

OnTAPIGetNumber

Generated prior to opening the port to obtain the phone number to dial.

OnTAPIWaveNotify

Generated whenever a the status of a wave file changes state.

OnTAPIWaveSilence

Generated whenever silence is detected while recording a wave file.

TAPI Status Processing

TAPI handles the details of controlling, dialing, and answering the modem. The OnTAPIStatus event is provided to let you know what's happening as the connection progresses.

TAPI actually uses a callback procedure to inform an application program of its progress. The APAXPort control installs a hidden callback and translates all calls into OnTAPIStatus events for easier processing. The format of the event handler is shown below and is identical to the internal TAPI callback.

```
expression_OnTAPIStatus(ByVal First As Boolean,
  ByVal Last As Boolean, ByVal Device As Long,
  ByVal Message As Long, ByVal Param1 As Long,
  ByVal Param2 As Long, ByVal Param3 As Long)
```

Part	Description	Data Type
expression	References the **APAXPort** that fired the event	**APAXPort**
First	Indicates if this is the first TAPI status event	**Boolean**
Last	Indicates if this is the last TAPI status event	**Boolean**
Device	Additional parameter for extended use	**Long**
Message	Indicates the state of the current call	**Long**
Param1	Used in conjunction with Message to further define the state of the current call	**Long**
Param2	Additional parameter for extended use	**Long**
Param3	Additional parameter for extended use	**Long**

First is True for the first status event of the current call. Last is True for the last status event of the current call. These parameters can be used to initialize and cleanup resources used when displaying status information.

The remaining parameters mirror the parameters that TAPI sends to the hidden callback procedure. Only Message and Param1 are used by APAX. The other parameters are provided in case you extend APAX beyond making and answering calls. Only Message and Param1 are described in detail; the rest are mentioned only briefly.

Message is a constant that describes the class of change since the previous OnTAPIStatus event was generated. The possible values are:

TAPI Message	Value	Explanation
Line_CallState	2	The state of the call changed
Line_LineDevState	8	The device state changed
Line_Reply	12	The previous request was accepted
Line_APDSpecific	32	APAX-specific status

The first three values are a subset of the possible TAPI messages. These are the only values that dialing and answering generate. The final value, Line_APDSpecific isn't really a TAPI status message. It's a pseudo state change that APAX generates to provide more information about the progress of a call.

Line_CallState indicates that the progress of the call, what TAPI calls the "state" of the call, changed. For example, the line was idle, but now it is dialing or the line was dialing but now it is proceeding (the TAPI term for waiting for the connection).

Line_LineDevState indicates that the state of the device (modem, phone, or whatever) changed. APAX dial/answer actions generate this message only to indicate that the modem is ringing.

Line_Reply indicates that TAPI has accepted, but not necessarily completed, the requested background. For example, it is generated just after a request to dial a number.

Line_APDSpecific is generated during periods when TAPI does not generate events, such as after dialing and waiting for a connection, or when answering and waiting for a connection. The primary purpose of Line_APDSpecific is to give the status event an opportunity to update a timer.

Param1 provides additional information about Message. For example, when Message is Line_CallState, Param1 contains a constant describing the change in state. Following are the values of Param1 that are generated by APAX for each value of Message:

Param1 Value	Explanation
For Message = Line_CallState	
LineCallState_Idle	No call in progress
LineCallState_Offering	Call starting
LineCallState_Accepted	Incoming call accepted
LineCallState_DialTone	Dialtone detected

Param1 Value	**Explanation**
LineCallState_Dialing	Dialing the outgoing number
LineCallState_Proceeding	Handshaking with the remote modem
LineCallState_RingBack	Detected a remote ring
LineCallState_Busy	Detected a busy signal
LineCallState_Connected	Connected with the remote modem
LineCallState_Disconnected	Disconnected from remote modem

For Message = Line_DevState	
LineDevState_Ringing	Ring detected for incoming call

For Message = Line_Reply	
(Param1 not used for Line_Reply)	

For Message = Line_APDSpecific	
APDSpecific_DialFail	Dial failed due to error
APDSpecific_RetryWait	Waiting for next retry
APDSpecific_TAPIChange	Unknown state change

These are only subsets of the possible values of Param1 for each of the Message states, but they are the only values generated for the dial/answer actions performed by the APAXPort control.

Other APAX properties can also be used in OnTAPIStatus event handler. Some examples are TAPINumber, which contains the number just dialed, and TAPISelectedDevice, which contains the name of the TAPI device.

Making Calls

The APAXPort control provides a dial method for placing outgoing calls. When TAPIDial is called, TAPI sends the appropriate modem configuration commands to the modem, then dials the number specified by TAPINumber.

The number passed to Dial should not contain any modem commands. It should contain only the telephone number to dial, exactly as it would be dialed from a telephone handset.

The APAXPort control generates OnTAPIStatus events during the dialing process. If a connection is not established, an OnTAPIFail event is generated. If a connection is established, an OnTAPIConnect event is generated, the associated serial port is opened, and the OnTAPIPortOpen event is generated.

Once the connection is established, all subsequent port control and input/output operations use the basic serial port properties and methods. The connection can be closed by calling the Close method. The APAXPort control then breaks the connection, closes the associated serial port, and generates the OnTAPIPortClose event.

If a dial attempt fails due to a busy signal or other error, APAX can try the call again. This is controlled by the MaxAttempts property, which determines how many times TAPIDial tries the call, and TAPIRetryWait, which determines how long (in seconds) TAPIDial waits before retrying a failed call.

Answering Calls

The process of answering the modem is very similar to dialing. The APAXPort control generates the same events as it does when dialing.

Calling TAPIAnswer causes the APAXPort control to wait for incoming calls in the background. No events are generated while waiting for calls. When an incoming call is detected, APAX begins generating OnTAPIStatus events at regular intervals. If a connection is not established, an OnTAPIFail event is generated. If a connection is established, an OnTAPIConnect event is generated, the associated serial port is opened, and the OnTAPIPortOpen event is generated.

Once the connection is established, all subsequent port control and input/output operations use the basic serial port properties and methods. The exception to this occurs when the call is terminated. The application should not simply close the APAXPort control because that would not disconnect the modem connection. Instead, the application must direct TAPI to close the connection by calling the Close method. The APAXPort control then breaks the connection, closes the associated serial port, and generates the OnTAPIPortClose event.

TAPI Service Providers

TAPI itself doesn’t implement any of the features necessary for controlling serial ports and telephony devices. The TAPI architecture dictates that the low-level, physical services are provided by a TAPI Service Provider (TSP).

Even if TAPI is properly installed, it will not function unless a service provider is also installed. TSP modules are typically provided by telephony vendors along with their telephony hardware. Windows installs a general purpose service provider named UNIMDM.TSP, which provides basic dial and answer support for modems. It is this service provider that makes TAPI available to communications programs in Windows. The lack of this service provider is what makes TAPI less likely to be useful in other Windows environments (which may have TAPI, but don’t have a general purpose modem service provider).

Since UNIMDM.TSP is the service provider that your application is most likely to encounter, it’s worth noting a few of its limitations here.

UNIMDM does not provide support for caller identification (caller ID). The CallerID property of The APAXPort control always returns an empty string when using UNIMDM.

UNIMDM does not support “no dialtone” detection. TAPI will attempt to dial whether a dialtone is detected or not.

See the TAPI Voice Support section on page 48 for an explanation of operating systems and voice capability issues.

Using TAPI for Configuration Only

Although UNIMDM.TSP provides basic dial and answer services it does not provide all of the modem services an application might need. UNIMDM.TSP cannot be used to place a faxmodem in fax answer mode or in adaptive answer mode (in which incoming calls are accepted).

However, TAPI (along with UNIMDM and modem information files) contains a wealth of configuration information and it is worthwhile to use TAPI to configure the modem and control the call, even if TAPI doesn't dial or answer the modem. This is provided by a feature called passthrough mode. In passthrough mode, TAPI immediately enters the connected state and opens the associated serial port.

Although, TAPI doesn't send any modem initialization commands in passthrough mode, APAX uses a two-step process to enter passthrough mode, which forces TAPI to send its modem initialization commands. When the TAPIConfigAndOpen method is called, APAX first initializes TAPI in answer mode, which forces TAPI to send its initialization commands. APAX then immediately closes the port and reopens it in passthrough mode.

After calling TAPIConfigAndOpen, TAPI is in control of the call, just as though it had dialed or answered the modem. No OnTAPIStatus events are generated, but OnTAPIPortOpen is generated. To close the call, use the Close method. If TAPI ever aborts or closes the call itself, the APAXPort control generates the OnTAPIPortClose event.

You should use TAPI passthrough mode if you need to support TAPI, but require modem operations that UNIMDM.TSP doesn't provide.

Wave File Support

The APAXPort control includes the ability to play and record wave files through a TAPI device (over the phone line). This feature, along with the new DTMF feature, allows you to created an automated voice answering system with APAX. To play and record wave files through the TAPI device, you must have the following:

- UNIMODEM/V on Windows 95/98 and Windows ME or UNIMODEM/5 on Windows 2000
- A voice modem with a wave driver
- A wave file

UNIMODEM/V is a set of DLLs that provides voice support for voice modems under Windows 95/98 and Windows ME. Voice support includes DTMF tone detection and generation, and wave file playback and recording. UNIMODEM/V is currently available only for Windows 95/98 and Windows ME. You can get UNIMODEM/V for Windows 95/98 from the Microsoft web site.

To use the voice extensions provided by UNIMODEM/V, you must have a voice modem. For wave support, it is important that you have the wave driver for the modem installed. Consult your modem documentation to install the wave device properly.

The APAXPort control allows you to set the wave file format used for playback and recording. The default wave format is PCM, 8KHz, 16 bit, mono. This format was chosen because it is supported by the majority of voice modems. Some voice modems support other wave file formats.

Wave files used for playback with APAX can be created with the Microsoft Sound Recorder program. Wave files for use with TAPI which will be played over general telephone lines (POTS) must be recorded in a PCM format compatible with your voice modem (here again, the attributes 8,000 Hz (8Khz), 16 Bit, mono are a good choice). Sound Recorder also allows for the conversion of existing wave files.

Recording options include the ability to detect silence on the line and take action when silence is detected (such as hanging up the call). This is desirable because TAPI does not have the ability to detect a hangup for a voice call. This option allows you to save disk space by saving only the portion of the call which contains data.

Dual Tone Multiple Frequency (DTMF)

Dual Tone Multiple Frequency (DTMF) tones are generated by a telephone touch pad over telephone lines. With compatible drivers and modems, APAX can detect (receive) and generate these tones. APAX notifies an application when it receives a tone by generating an OnTAPIDTMF event. Tones are generated using the TAPISendTone method. See the TAPI Voice Support section on page 48 for an explanation of operating systems and voice capabily issues.

TAPI References

Following is a list of the APAXPort control's properties, methods, and events that pertain to TAPI facilities. This is only a subset of the functionality of the APAXPort functionality. Additional properties, methods, and events are introduced in other chapters.

Properties

AnswerOnRing	MaxMessageLength	TAPIRetryWait
CallerID	SelectedDevice	TAPIState
Dialing	SilenceThreshold	TrimSeconds
EnableVoice	TAPIAttempt	UseSoundCard
InterruptWave	TAPICancelled	WaveFileName
MaxAttempts	TAPINumber	WaveState

Methods

TAPIAnswer	TAPIRecordWaveFile	TAPIShowConfigDialog
TAPIConfigAndOpen	TAPISelectDevice	TAPIStopWaveFile
TAPIDial	TAPISendTone	TAPITranslatePhoneNumber
TAPIPlayWaveFile	TAPISetRecordingParams	

Events

OnTAPICallerID	OnTAPIGetNumber	OnTAPIWaveNotify
OnTAPIConnect	OnTAPIPortClose	OnTAPIWaveSilence
OnTAPIDTMF	OnTAPIPortOpen	
OnTAPIFail	OnTAPIStatus	

Reference section

AnswerOnRing property

Description

The number of times the TAPI device should allow the incoming call to ring before answering it. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`AnswerOnRing`**`[= `*`value`*`]`

`expression`` must reference an `**`APAXPort`**`.`

Remarks

Default: 2

The default for AnswerOnRing is two rings because problems can occur with caller-ID enabled modems if the call is answered after the first ring.

CallerID property

Description

Contains the caller identification string of the current incoming call. Read-only, run-time.

Data type

`String`

Syntax

`expression``.`**`CallerID`**

`expression`` must reference an `**`APAXPort`**`.`

Remarks

Many telephony environments make a caller identification string available. This string usually contains the phone number of the incoming call, but can contain other information as well, if supplemented by an office telephony system.

If the telephony environment doesn't supply caller identification information, CallerID is an empty string.

Caller ID requires voice capabilities (i.e.: UnimodemV, voice modem).

Example

The following example shows an OnTAPIConnect event handler that updates a TLabel on the current form with the caller ID information:

```
Private Sub Apax1_OnTAPIConnect()
  Label1.Caption = Apax1.CallerID
End Sub
```

Dialing property

Description

Determines whether TAPI is placing an outgoing call or listening for an incoming call. Read-only, run-time.

Data type

`Boolean`

Syntax

`expression``.`**`Dialing`**

`expression` `must reference an` **`APAXPort`**`.`

Remarks

Dialing is True when TAPI is placing an outgoing call, False when TAPI is listening for or answering incoming calls. Dialing is intended primarily for use in status routines to distinguish between status events for incoming calls and status events for outgoing calls.

Example

The following example shows an OnTAPIStatus event handler that uses the Dialing property to update a TLabel on the current form:

```
Private Sub Apax1_OnTAPIStatus(ByVal First As Boolean, ByVal Last
As Boolean, ByVal Device As Long, ByVal Message As Long, ByVal
Param1 As Long, ByVal Param2 As Long, ByVal Param3 As Long)
  If Apax1.Dialing = True Then
    Label1.Caption = "Dialing"
  Else
    Label1.Caption = "Idle"
  End If
End Sub
```

See also

TAPINumber

EnableVoice property

Description

Determines whether the initial mode is DataModem or AutomatedVoice (Voice/DTMF). Read/write.

Data type

`Boolean`

Syntax

`*expression*.**EnableVoice**[= *value*]`

`*expression* must reference an **APAXPort**.`

Remarks

Default: False

If EnableVoice is True and a TAPI device is selected, APAX first verifies that AutomatedVoice capabilities exist for the selected device. If so, voice extensions such as DTMF and wave files are supported. Otherwise, EnableVoice is set to False.

See also

OnTAPIDTMF, TAPIPlayWaveFile, TAPIRecordWaveFile, TAPISendTone

InterruptWave property

Description

Indicates whether the current wave file should stop when a DTMF tone is detected. Read/write.

Data type

`Boolean`

Syntax

`expression``.`**`InterruptWave`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort`**`.`

Remarks

Default: True

If InterruptWave is True, the currently playing wave file will stop when a DTMF tone is detected. This allows you to have an automated phone answering system in which user is allowed to interrupt the currently playing wave file with a DTMF selection. If InterruptWave is False, DTMF tones do not stop the currently playing wave file. If you want the caller to hear a wave file in its entirety, set InterruptWave to False before you start playing it.

See also

OnTAPIDTMF, TAPIPlayWaveFile, TAPIStopWaveFile

MaxAttempts property

Description

Determines the number of times TAPIDial automatically dials a number. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`MaxAttempts`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort`**`.`

Remarks

Default: 3

This is the number of times a phone number is dialed, it is not the number of retries. When MaxAttempts is one, for example, the number is dialed only once. If the line is busy, it is not tried again.

See also

TAPIAttempt, TAPIRetryWait

MaxMessageLength property

Description

The maximum allowed message length, in seconds, for messages recorded over the TAPI waveform audio device. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`MaxMessageLength`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort`**`.`

Remarks

Default: 60

Use this parameter to specify the maximum length of recorded messages. A 60-second message will require about 950K of disk space given the default recording parameters. When the specified length of time passes, the OnTAPIWaveNotify event will be generated with a Msg parameter of waDataReady. You could then save the wave file and terminate the call.

If the TrimSeconds property is set to a non-zero value, wave recording may terminate before MaxMessageLength is reached.

OnTAPICallerID event

Description

Defines an event handler that is called after a connection is made and caller ID information is available.

Syntax

```
Private Sub expression_OnTAPICallerID(
  ByVal ID As String, ByVal IDName As String)
```

Part	Description	Data Type
expression	References the **APAXPort** that fired the event	**APAXPort**
ID	The ID of the caller	**String**
IDName	The name of the caller	**String**

Remarks

The OnTAPICallerID event makes it easy to access Caller ID information without having to know when it might be available on a call. Caller ID information is available only if it is supported on the selected device and by the telephone service.

Example

The following example shows how to use the OnTAPICallerID event to get the Caller ID information and display it in labels:

```
Private Sub Apax1_OnTAPICallerID(ByVal ID As String, ByVal IDName
As String)
  Label1.Caption = ID
  Label2.Caption = IDName
End Sub
```

See also

CallerID

OnTAPIConnect event

Description

Defines an event handler that is called when a connection is established.

Syntax

`Private Sub` *`expression`*`_`**`OnTAPIConnect()`**

`expression` `must reference an` **`APAXPort`**`.`

Remarks

TAPIDial and TAPIAnswer operations take place in the background. If a connection is established after a call to TAPIDial or TAPIAnswer, the APAXPort generates the OnTAPIConnect event.

No parameters are passed to OnTAPIConnect. It is a notification to the application that a connection was successfully established. You can use this event to perform connection start-up activities (e.g., activating and displaying a terminal window).

OnTAPIDTMF event

Description

Defines an event handler that is called when a DTMF tone is detected.

Syntax

```
Private Sub expression_OnTAPIDTMF(
  ByVal Digit As Byte, ByVal ErrorCode As Long)
```

Part	Description	Data Type
expression	References the **APAXPort** that fired the event	**APAXPort**
Digit	The phone button that was pressed on the remote phone device	**Byte**
ErrorCode	Indicates the error or success status	**Long**

8

Remarks

Digit is a character that represents the phone button that was pressed on the remote phone device. The possible values are '0' through '9', '*', and '#'. ErrorCode is non-zero if an error occurs when a TAPI connection is made (in this case the OnTAPIDTMF is generated just before the OnTAPIConnect event).

Example

The following example builds a string of up to ten DTMF tones (characters) in the global variable S:

```
Private Sub Apax1_OnTAPIDTMF(ByVal Digit As Byte, ByVal ErrorCode
As Long)
Dim s As String

  If Length(s) < 11 Then
    s = s + Digitm
  End If
End Sub
```

See also

EnableVoice

OnTAPIFail event

Description

Defines an event handler that is called when a connection attempt fails.

Syntax

`Private Sub` *`expression_`***`OnTAPIFail()`**

`expression` `must reference an` **`APAXPort`**`.`

Remarks

TAPIDial and TAPIAnswer operations take place in the background. If an attempt to establish a connection fails, the APAXPort control generates the OnTAPIFail event.

No parameters are passed to OnTAPIFail. It is a notification to the application that a connection attempt failed.

OnTAPIGetNumber event

Description

Defines an event handler that is fired prior to TAPI attempting to dial.

Syntax

`Private Sub` *`expression_`***`OnTAPIGetNumber(`***`PhoneNum`* **`as String)`**

Part	Description	Data Type
expression	References the APAXPort object that fired the event	**APAXPort**
PhoneNum	Defines the phone number to dial	**String**

Remarks

The PhoneNum parameter passed to this event will reflect the current TAPINumber property. If TAPINumber is blank, you must assign a valid phone number to the PhoneNum parameter passed to this event.

See also

TAPINumber

OnTAPIPortClose event

Description

Defines an event handler that is called immediately after the port is closed.

Syntax

`Private Sub` *`expression`*`_`**`OnTAPIPortClose()`**

`expression` `must reference an` **`APAXPort`**`.`

Remarks

The APAXPort control is responsible for opening and closing the associated serial port at the appropriate times (when a connection is established or broken).

Applications can use this event to perform additional port cleanup activities.

See also

OnTAPIPortOpen

OnTAPIPortOpen event

Description

Defines an event handler that is called immediately after the port is opened.

Syntax

`Private Sub` *`expression`*`_`**`OnTAPIPortOpen()`**

`expression` `must reference an` **`APAXPort`**`.`

Remarks

The APAXPort control is responsible for opening and closing the associated serial port at the appropriate times (when a connection is established or broken).

Applications can use this event to perform additional port setup activities.

See also

OnTAPIPortClose

OnTAPIStatus event

Description

Defines an event handler that is called regularly during a TAPI dial or answer attempt.

Syntax

```
Private Sub expression_OnTAPIStatus(
  ByVal First As Boolean, ByVal Last As Boolean,
  ByVal Device As Long, ByVal Message As Long,
  ByVal Param1 As Long, ByVal Param2 As Long,
  ByVal Param3 As Long)
```

Part	Description	Data Type
expression	References the **APAXPort** that fired the event	**APAXPort**
First	Indicates if this is the first TAPI status event	**Boolean**
Last	Indicates if this is the last TAPI status event	**Boolean**
Device	Additional parameter for extended use	**Long**
Message	Indicates the state of the current call	**Long**
Param1	Used in conjunction with Message to further define the state of the current call	**Long**
Param2	Additional parameter for extended use	**Long**
Param3	Additional parameter for extended use	**Long**

Remarks

TAPI performs dial and answer activities in the background, calling a callback routine whenever the state of the line or call changes. APAX installs a hidden callback routine and translates all callback calls into OnTAPIStatus events.

First is True on the first OnTAPIStatus event to signal the status routine to perform its start-up activities (e.g., make the status display visible). Last is True on the last OnTAPIStatus event to signal the status routine to perform its cleanup activities (e.g., remove the status display). First and Last are False on all other OnTAPIStatus events.

The other parameters are passed by TAPI to the callback routine. The only parameters that are of interest to most APAX programs are Message and Param1, which indicate the state of the current call. See page 172 for an explanation of the values passed for the Message and Param/ parameters.

The remaining parameters (Device, Param2, and Param3) are intended for use in applications that extend the features provided by APAX.

TAPI generates callbacks only when it perceives a change in the state of the line or call. TAPI, therefore, does not generate callbacks when the modem stays in a single state for an extended period of time. For example, after dialing a number TAPI reports that the call is in the "proceeding" phase. It generates no further status callbacks until the call succeeds or fails. Since this can take many seconds, as much as 20 or even 30 seconds, the user might become concerned about the lack of positive feedback (is it still working?).

To solve this problem, APAX generates additional OnTAPIStatus events, based on an internal timer (once per second). These status calls give the status routine an opportunity to update a timer.

OnTAPIWaveNotify event

Description

Defines an event handler that is called when a wave file status changes.

Syntax

`Private Sub` *`expression`*`_`**`OnTAPIWaveNotify(`**
  `ByVal` *`Msg`* **`As TxWaveMessage)`**

Part	Description	Data Type
expression	References the **APAXPort** that fired the event	**APAXPort**
Msg	Describes the status of the wave file/device	**TxWaveMessage**

Settings

Possible values for Msg are:

Constant	Description
waPlayDone	The wave file is finished playing
waPlayOpen	The wave file is open
waPlayClose	The wave file is closed
waRecordOpen	The wave device is open for recording
waRecordClose	The wave device is closed
waDataReady	The wave device has recorded data

Example

The following example sets the Caption of a label after a wave file has finished playing:

```
Private Sub Apax1_OnTAPIWaveNotify(ByVal Msg As TxWaveMessage)
  If Msg = waPlayDone Then
    Label1.Caption = "Wave Device Idle..."
  End If
End Sub
```

See also

TAPIPlayWaveFile, TAPIRecordWaveFile, TAPIStopWaveFile

OnTAPIWaveSilence event

Description

Defines an event handler that is called when silence is detected while recording a wave file.

Syntax

```
Private Sub expression_OnTAPIWaveSilence(
  StopRecording As Boolean, Hangup As Boolean)
```

Part	Description	Data Type
expression	References the **APAXPort** that fired the event	**APAXPort**
StopRecording	Determines whether wave recording should stop	**Boolean**
HangUp	Determines whether the call should be terminated	**Boolean**

Remarks

StopRecording is a Boolean parameter that determines whether wave recording should stop. This parameter is True by default. Hangup is a Boolean parameter that determines whether the call should be terminated. The parameter is also True by default.

This event works in conjunction with the TrimSeconds property. If TrimSeconds is 0 then OnTAPIWaveSilence will not be generated. If you do not respond to this event, recording will stop and the call will be terminated when silence is detected. This is probably the desired behavior in most applications.

See also

TAPIRecordWaveFile

SelectedDevice property

Description

Determines the TAPI device to be used for dialing and answering. Read/write.

Data type

`String`

Syntax

```
expression.SelectedDevice[= value]
```

```
expression must reference an APAXPort.
```

Remarks

TAPI assigns names to each installed modem. APAX selects among those devices by setting SelectedDevice to the name of the desired TAPI device. Since these names can sometimes be rather lengthy and cumbersome to type, a property editor is provided for easier selection of devices.

Because the name specified in SelectedDevice must exactly match a TAPI device name, you should use this property editor in your application if you need to allow users to select a TAPI device.

SelectedDevice must be set before calling TAPIDial, TAPIAnswer, or TAPIShowConfigDialog.

See also

TAPISelectDevice

SilenceThreshold property

Description

Specifies a value that is used as a measure of silence. Read/write.

Data type

`Boolean`

Syntax

`expression.`**`SilenceThreshold`**`[= value]`

`expression` `must reference an` **`APAXPort`**`.`

Remarks

Default: 50

When the TrimSeconds property is set to a non-zero value, the wave data is examined as it is recorded. Silence is determined by comparing the average of the wave data for one second to a silence threshold as defined by SilenceThreshold. PCM data recorded by the TAPI wave driver generally has an amplitude of 400 to 800 for normal speech. A silence threshold of 50 (the default) is conservative. True silence on the phone line is probably less than 20, although anything under 200 could probably be considered silence. Modify the SilenceThreshold property if your phone lines contain more or less noise.

See also

TAPIRecordWaveFile, TrimSeconds

TAPIAnswer method

Description

Instructs TAPI to listen for and answer incoming calls.

Syntax

`expression.TAPIAnswer`

`expression must reference an APAXPort.`

Remarks

TAPIAnswer returns immediately after instructing TAPI to listen for calls. TAPI listens for calls in the background. When an incoming call is detected, it answers the call, generating appropriate events as it does.

Calling the Close method terminates the current call.

TAPIAttempt property

Description

Indicates the number of times the current number has been dialed. Read-only, run-time.

Data type

`Integer`

Syntax

`expression.TAPIAttempt`

`expression must reference an APAXPort.`

Remarks

If the dialed number is busy, TAPI waits briefly and calls the number again. It tries up to MaxAttempts times. The TAPIAttempt property returns the number of the current attempt. TAPIAttempt is incremented immediately upon encountering a busy line. TAPIAttempt is primarily for use in OnTAPIStatus event handlers.

See also

MaxAttempts, OnTAPIStatus, TAPIRetryWait

TAPICancelled property

Description

Returns True if an OnTAPIFail event fires due to user action. Read-only, run-time.

Data type

`Boolean`

Syntax

`expression``.`**`TAPICancelled`**

`expression` `must reference an` **`APAXPort`**`.`

Remarks

An OnTAPIFail event is generated any time a call is terminated before the final connection is made—even if Close is used to terminate the call.

See also

Close, OnTAPIFail

TAPIConfigAndOpen method

Description

Configures the modem and leaves the port open in passthrough mode.

Syntax

`expression``.`**`TAPIConfigAndOpen`**

`expression` `must reference an` **`APAXPort`**`.`

Remarks

TAPIConfigAndOpen takes advantage of the TAPI modem configuration facilities, even though TAPI isn't used for dialing or answering a call. TAPIConfigAndOpen does, however, require a short period of background processing before the associated APAXPort control is open. See page 176 for more information on passthrough mode.

TAPIDial method

Description

Dials a phone number in the background.

Syntax

expression.**TAPIDial()**

Remarks

TAPIDial instructs TAPI to prepare the modem for dialing, then to dial the number specified by the TAPINumber property. The OnTAPIGetNumber event is fired to obtain a phone number if desired. All of these operations take place in the background. APAX generates the OnTAPIStatus event to keep the program apprised of the dialing progress.

If a modem connection is established, the OnTAPIConnect and OnTAPIPortOpen events are generated. If a modem connection is not established, the OnTAPIFail event is generated.

If a busy signal is detected and MaxAttempts is greater than one, TAPIDial redials the number after waiting the number of seconds set in TAPIRetryWait. This continues until a connection is established or MaxAttempts dial attempts fail.

Example

The following example shows how to dial the U.S. Robotics BBS, waiting 5 minutes after a busy signal and retrying up to 10 times.

```
Apax1.TAPIRetryWait = 300
Apax1.MaxAttempts = 10
Apax1.TAPINumber = "1-847-262-2000"
Apax1.TAPIDial
```

See also

TAPIAnswer, MaxAttempts, TAPINumber, TAPIRetryWait

TAPINumber property

Description

Specifies the phone number used when dialing. Read-only, run-time.

Data type

`String`

Syntax

`expression``.`**`TAPINumber`**

`expression` `must reference an` **`APAXPort`**`.`

Remarks

The value contained by TAPINumber is the phone number that is dialed by the TAPIDial method.

TAPINumber should not contain any modem commands. It should contain the telephone number exactly as it would be dialed from a telephone handset.

See also

TAPIDial, Dialing, OnTAPIGetNumber

TAPIPlayWaveFile method

Description

Plays a wave file.

Syntax

`expression``.`**`TAPIPlayWaveFile(`*`FileName`*`)`**

Part	**Description**	**Data Type**
expression	An expression that returns an **APAXPort** object	**APAXPort**
FileName	Specifies the wave file name	**String**

Remarks

FileName is the name of the wave file. The wave file starts playing immediately if there is not a wave file currently playing. If another wave file is currently playing and InterruptWave is True, the current wave file is stopped and the new wave file is played. If another wave file is playing and InterruptWave is False, TAPIPlayWaveFile returns without playing the new wave file. The wave file is played through the TAPI device if the UseSoundCard property is False (the default), or through the sound card if UseSoundCard is True.

Example

The following example plays a wave file through the TAPI device:

```
Apax1.TAPIPlayWaveFile("greeting.wav")
```

See also

InterruptWave, OnTAPIWaveNotify, TAPIStopWaveFile, UseSoundCard

TAPIRecordWaveFile method

Description

Starts the wave device recording.

Syntax

`expression.`**`TAPIRecordWaveFile(`*`FileName, Overwrite`*`)`**

Part	Description	Data Type
expression	An expression that returns an **APAXPort** object	**APAXPort**
FileName	Specifies the wave file name	**String**
Overwrite	Determines whether or not to overwrite an existing file	**Boolean**

Remarks

Use TAPIRecordWaveFile to begin recording a wave file using the TAPI waveform audio device. Recording stops when the TAPIStopWaveFile method is called, when the wave input buffer is full, or when silence is detected on the line. The size (in seconds) of the wave input buffer is determined by MaxMessageLength. Silence detection is controlled through the TrimSeconds property.

When recording stops, the OnTAPIWaveNotify event is generated.

Example

The following example sets the maximum message length to 45 seconds and starts recording the wave data on a button click:

```
Private Sub Command1_Click()
  Apax1.MaxMessageLength = 45
  Apax1.TAPIRecordWaveFile("call01.wav", True)
End Sub

Private Sub Apax1_OnTAPIWaveNotify(ByVal Msg As TxWaveMessage)
  If Msg = waDataReady Then
    Apax1.TAPIRecordWaveFile("Call01.wav",True)
  End If
End Sub
```

See also

MaxMessageLength, TAPIPlayWaveFile, TAPIStopWaveFile, TrimSeconds

TAPIRetryWait property

Description

The number of seconds to wait after a busy signal before trying the number again. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`TAPIRetryWait`**`[= `*`value`*`]`

`expression`` must reference an `**`APAXPort`**`.`

Remarks

Default: 60

After encountering a busy signal, APAXPort control checks to see if it should try this number again by comparing Attempts to MaxAttempts. If more attempts are required, it first waits the number of seconds set in TAPIRetryWait before dialing again to give the dialed machine time to complete the current session.

See also

TAPIAttempt, MaxAttempts

TAPISelectDevice method

Description

Displays a dialog box to select a TAPI device.

Syntax

`expression.`**`TAPISelectDevice`**

expression must reference an **APAXPort**.

Remarks

TAPISelectDevice is used to update SelectedDevice. You can call TAPISelectDevice to prompt the user for a TAPI device to use for subsequent dial or answer operations.

TAPISendTone method

Description

Sends a DTMF tone to a remote telephone.

Syntax

expression.**SendTone(*Digits*)**

Part	Description	Data Type
expression	An expression that returns an **APAXPort** object	**APAXPort**
Digits	Specifies the digit(s) to send	**String**

Remarks

TAPISendTone replicates the press of a telephone touch pad button from within an application. Digits should consist of valid telephone touch pad buttons (i.e., '1' through '9', '*', and '#').

You can also use a comma (,) between characters for a short delay between the tones. Multiple comma characters can be used to create a longer delay.

Example

The following example demonstrates how to use TAPISendTone to send multiple tones with a delay:

```
TAPISendTone("123456789,,0")
```

TAPISetRecordingParams method

Description

Sets the parameters used to record a wave file.

Syntax

```
expression.TAPISetRecordingParams(
  NumChannels, NumSamplesPerSecond, NumBitsPerSample)
```

Part	Description	Data Type
expression	An expression that returns an APAXPort object	**APAXPort**
NumChannels	Determines mono or stereo recording	**Integer**
NumSamplesPerSecond	Determines number of samples per second	**Integer**
NumBitsPerSample	Determines the recording resolution	**Integer**

8

Remarks

NumChannels is the number of channels to use for recording. A value of 1 indicates mono, and a value of 2 indicates stereo. Due to the nature of telephony, it is unlikely any TAPI devices support stereo recording. NumSamplesPerSecond is the number of samples per second to use for recording. NumBitsPerSample is the number of bits of data to record per sample.

By default recording parameters are set to 1 channel (mono), 8000 samples per second, 16 bits per sample. If your TAPI device supports other recording formats you can use this method to change the recording format. It is unlikely that you will need to change the recording parameters.

See also

TAPIRecordWaveFile

TAPIShowConfigDialog method

Description

Shows a TAPI configuration dialog for the selected device.

Syntax

expression.**TAPIShowConfigDialog(*AllowEdit*)**

Part	Description	Data Type
expression	An expression that returns an **APAXPort** object	**APAXPort**
AllowEdit	Determines whether the dialog is displayed read-only	**Boolean**

Remarks

If AllowEdit is True, the resulting dialog displayed will allow users to modify the TAPI settings for the selected device. If AllowEdit is False, the dialog will be read only.

See also

MaxMessageLength, TAPIPlayWaveFile, TAPIRecordWaveFile, TAPIStopWaveFile, TrimSeconds

TAPIState property

Description

The state of the TAPI operation. Read-only, run-time.

Data type

TTAPIState

Syntax

expression.**TAPIState**

expression must reference an **APAXPort**.

Settings

Possible values of TAPIState are:

Constant	Value	Description
tsIdle	0	No TAPI operations in progress
tsOffering	1	TAPI is offering an incoming call
tsAccepted	2	APAX has accepted an incoming call
tsDialTone	3	Dial tone detected
tsDialing	4	Waiting for dial to complete or fail
tsRingback	5	Ringback detected
tsBusy	6	Line is busy
tsSpecialInfo	7	TAPI service provider specific
tsConnected	8	Call is connected
tsProceeding	9	Call is proceeding
tsOnHold	10	Call has been placed on hold
tsConferenced	11	Call is conferenced
tsOnHoldPendConf	12	Call is being conferenced
tsOnHoldPendTransfer	13	Call is being transferred
tsDisconnected	14	Call has been disconnected
tsUnknown	15	TAPI state unknown to APAX

Remarks

Default: tsNone

When TAPIState is referenced, APAX retrieves state information from TAPI and returns the result as TAPIState. For completeness, the TTAPIState constants contains all possible returns from TAPI (which is a superset of values that you will likely see in an APAX application).

Since APAX retrieves this value from TAPI every time you check it, you should avoid calling it too often. In other words, sitting in a loop continuously polling TAPIState would not be a good idea.

TAPIStopWaveFile method

Description

Stops the wave file that is currently playing or recording.

Syntax

expression.**TAPIStopWaveFile**

expression must reference an **APAXPort**.

Remarks

The wave file is halted regardless of the value of the InterruptWave property. TAPIStopWaveFile generates two OnTAPIWaveNotify events. The first has a Code of WOM_DONE and the second has a Code of WOM_CLOSE. If no wave file is currently playing then TAPIStopWaveFile returns silently.

Example

The following example stops a wave file, if one is currently playing:

```
if Apax1.WaveState = wsPlaying then
  Apax1.TAPIStopWaveFile
End If
```

See also

InterruptWave, OnTAPIWaveNotify, TAPIPlayWaveFile, WaveState

TAPITranslatePhoneNumber method

Description

Translates a canonical address into a dialable address. Returns a string representing the dialable address.

Syntax

expression.**TAPITranslatePhoneNumber(*CanonicalAddr*)**

Part	Description	Data Type
expression	An expression that returns an **APAXPort** object	**APAXPort**
CanonicalAddr	Specifies the canonical address	**String**

Remarks

A canonical address is an address that contains the country code as well as the phone number. TAPI also takes into account any settings you have made to your modem properties in the Control Panel. For example, if you have call waiting enabled and the code to disable call waiting is *70, TAPI adda *70 to the beginning of the dialable address string when you call TAPITranslatePhoneNumber.

See also

TAPIDial

TrimSeconds property

Description

Sets the number of seconds of silence to detect when recording wave files. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`TrimSeconds`**`[= `*`value`*`]`

`expression`` must reference an `**`APAXPort`**`.`

Remarks

Default: 2

Wave recording can be terminated in one of three ways. First, you can manually terminate recording by calling TAPIStopWaveFile. Second, recording will automatically terminate when the amount of time specified by MaxMessageLength has passed. Finally, wave recording can terminate as a result of silence detected on the line.

When TrimSeconds is set to a non-zero value, the wave data is examined as it is recorded. Silence is determined by comparing the average of the wave data for one second to a silence threshold as defined by the SilenceThreshold property. If TrimSeconds seconds of silence is detected, the OnTAPIWaveSilence event is generated. If no OnTAPIWaveSilence event is defined then the recording is stopped and the call is terminated. Even after a hangup, a telephone line contains a good deal of random noise so it is not guaranteed that silence will be detected immediately after a hangup.

See also

OnTAPIWaveSilence, SilenceThreshold, TAPIRecordWaveFile

UseSoundCard property

Description

Determines where the output from TAPIPlayWaveFile is sent. Read/write.

Data type

`Boolean`

Syntax

`expression``.`**`UseSoundCard`**`[= `*`value`*`]`

`expression`` must reference an `**`APAXPort`**`.`

Remarks

Default: False

UseSoundCard determines whether the output from TAPIPlayWaveFile goes to the TAPI device or to the sound card. By default the output is sent to the TAPI waveform audio device (through the phone). Set UseSoundCard to True to play the wave file through the sound card.

8

Example

The following example plays a wave file through the sound card rather than over the phone line and then resets the device so that subsequent sounds are played through the TAPI device:

```
  Apax1.UseSoundCard = True
  Apax1.TAPIPlayWaveFile("Call01.wav")
  Apax1.UseSoundCard = False
```

See also

TAPIPlayWaveFile

WaveFileName property

Description

The name of the current wave file. Read/write.

Data type

`String`

Syntax

`expression``.`**`WaveFileName`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort`**`.`

Remarks

If a wave file is currently playing, WaveFileName is the name of the file. If no wave file is currently playing, WaveFileName is the name of the last wave file that was played. WaveFileName is automatically set when you use TAPIPlayWaveFile to play a file.

Example

The following example sets a label's caption to the name of the current wave file:

```
  Label1.Caption = Apax1.WaveFileName
```

See also

TAPIPlayWaveFile

WaveState property

Description

The current state of the TAPI waveform device. Read/write.

Data type

`TWaveState`

Syntax

`expression``.`**`WaveState`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort`**`.`

Settings

Possible values for WaveState are:

Constant	Value	Description
wsIdle	0	The wave device is not in use
wsPlaying	1	The wave device is playing a wave file
wsRecording	2	The wave device is recording a wave file
wsData	3	Data is available in the wave buffer

Example

The following example stops a wave file if one is currently playing:

```
if Apax1.WaveState = wsPlaying then
  Apax1.TAPIStopWaveFile
End If
```

See also

TAPIPlayWaveFile, TAPIRecordWaveFile, TAPIStopWaveFile, TAPIStopWaveFile

Chapter 9: File Transfer Protocols

Many communications applications need to transfer files or other large amounts of data from one machine to another. This could be accomplished by having the sender call PutData repeatedly and the receiver reading the data in an OnRXD event handler correspondingly. However, the application would have a tremendous amount of detail work still to do. It would need logic to transfer file name and size information, to check for and recover from transmission errors, to handle file I/O, and so on.

That's why APAX provides standard, tested, reliable, high performance file transfer protocols. The term "protocol" means that both sides of the communication link behave in a clear, well-defined manner following agreed-upon rules. The rules vary among the different protocols and some protocols offer more control and features than others. At a minimum, each protocol handles file I/O and serial port I/O and checks for errors. Some protocols also include error correcting logic, multi-file transfers, and automatic recovery after partial file transfers.

APAX offers the most widely used industry standard file transfer protocols, as shown in the following table.

Protocol	Description
Xmodem	128 byte blocks with checksum block checking
XmodemCRC	128 byte blocks with CRC block checking
Xmodem1K	1024 byte blocks with CRC block checking
Xmodem1KG	Streaming Xmodem1K
Ymodem	1024 byte blocks, batch
YmodemG	Streaming Ymodem
Zmodem	1024 byte blocks, batch, streaming, restartable
Kermit	80 byte blocks, batch, with long blocks and windowing
ASCII	ASCII stream with inter-character and inter-line delays

General Issues

The APAX control encapsulates the protocol engine in a simple to use interface. The protocol engine works in the background by using an internal APAX timer, and various data triggers. Windows can continue with other tasks while a file transfer is in progress as long as the other tasks yield properly for other events.

The following subsections document issues that arise for all types of file transfers that use the APAX protocol engine.

Buffer sizes

When an APAXPort control is created, input and output buffers of sufficient size to handle any protocol are allocated automatically. The following is a brief explanation of why APAX requires relatively large buffers for file transfers.

Once a file transfer starts, it is likely that the user will work in another window until the file transfer finishes. Because the transfer is running as a background application, it is at the mercy of other Windows tasks. Many Windows applications and built-in Windows operations can hog the CPU to an extent that prevents the background transfer from succeeding.

To different degrees, all file transfer protocols are time-critical. They must respond to incoming events in a timely fashion, usually within a few seconds. If they fail to respond in the required time, the remote protocol software times out and repeats the failed operation. Such time-outs and repetitions at best reduce the protocol transfer rate and at worst can cause the protocol to fail.

In practice, it takes a very ill-behaved program or unusual user (e.g., someone who spends 30 seconds to move a window) to cause most protocols to fail. But this can happen and your application should do whatever it can to minimize the chances.

One of the things that APAX does for you is to use a large input buffer. The Windows communications driver continues to receive data and store it in the input buffer even if the associated application program isn't getting any time to run. With a 30K byte input buffer and a data rate of 1600 characters per second, the buffer can hold 19 seconds worth of incoming data before overflowing. When the application eventually regains control it processes all received data before relinquishing control. The input buffer is then ready to hold another 19 seconds worth of data.

A large output buffer is also valuable when transmitting files. Streaming protocols such as YmodemG and Zmodem, and Kermit to a lesser degree, typically transmit until they fill the output buffer, then relinquish control. They don't regain control until the buffer drains enough to hold another data block and Windows can process the associated status trigger message.

If the status update is delayed because another Windows application didn't yield, the Windows communications driver continues to transmit the data remaining in the output buffer. Using the same numbers as the input buffer example, the driver can transmit independently for up to 19 seconds before running out of data.

In summary, APAX automatically allocates buffer sizes large enough to transfer files using any protocol. You need not concern yourself with buffer sizes.

Protocol events

The APAXPort control generates several events related to file transfers. General descriptions of these events follow.

OnProtocolAccept

Generated as soon as the protocol window knows the name of an incoming file. This provides an opportunity to accept or reject the file, or to change its name.

OnProtocolFinish

Generated after all files have been transferred or after the protocol terminates due to an unrecoverable error. This event also sends the final result code of the protocol.

OnProtocolLog

Generated at the start and end of transferring each file. This provides an opportunity to log the status of each file transfer.

OnProtocolStatus

Generated at regular intervals so that programs can display the progress of the protocol.

Aborting a protocol

There will certainly be times when a protocol in progress must be canceled (e.g., when something goes wrong at the remote computer or the user simply decides not to continue the transfer). APAX protocols provide for this situation with the CancelProtocol method.

To cancel any protocol simply call the CancelProtocol method of the APAXPort control. This method sends an appropriate cancel sequence to the remote computer and terminates. The APAXPort control remains ready to handle additional protocol transfers.

When protocol transfers take place over a modem link it is a good idea to monitor the data carrier detect (DCD) line and abort the protocol if carrier is lost. The DCD line goes high when modems first connect and remains high until one of the modems hangs up. Occasionally, line noise or other disturbances in the telephone network break the connection between the modems, causing DCD to go low.

Some protocols quickly detect that the remote isn't acknowledging after the connection is broken. These protocols soon abort with an error code of ecTimeout or ecTooManyErrors. By contrast, streaming protocols can take a very long time to notice that the connection is broken because they don't require acknowledgments.

The APAX protocol engine provides an option to handle dropped carrier automatically. Set the AbortNoCarrier property to True before calling StartTransmit or StartReceive and the protocol engine automatically aborts if the DCD line is not high at any point during the protocol. The protocol cancels itself immediately and generates an OnProtocolFinish event with an error code of ecAbortNoCarrier.

Using the AbortNoCarrier property is better than checking DCD and calling CancelProtocol in your own code. When you do this, the protocol engine sends a cancel sequence to the remote computer. If hardware flow control is enabled and the modem has lowered the DSR or CTS signals as well as DCD, the protocol waits several seconds before deciding it can't send the cancel command, leading to an unnecessary delay for the application. The AbortNoCarrier option prevents the protocol engine from sending the cancel sequence, so the protocol stops immediately.

Error handling

All protocol transfers are subject to errors, including parity errors, files not found, and other file I/O errors. Whenever possible the protocol window handles errors internally by retrying an operation or requesting the remote computer to retry. At some point, however, it determines that the situation is unrecoverable and terminates, generating an OnProtocolFinish event. An application should include a handler for this event and check the error code to determine success or failure of the protocol.

Protocol status

A protocol transfer can last a few seconds or several hours depending on the size and speed of the transfer. Because the protocol component handles the details of the transfer from start to finish, your application's code is not executing during this entire time. You and your users certainly want to know what's happening as the transfer progresses, so APAX provides a hook for your application to regularly regain control during this time.

During a protocol transfer the protocol window frequently generates an OnProtocolStatus event. This gives your code the opportunity to monitor and display the progress of the protocol. The following code fragment illustrates how.

```
If Options = apFirstCall
  ...do setup stuff
else
  ...do cleanup stuff
```

The Options parameter passed to this event handler routine can take on two special values:

```
apFirstCall = 1
apLastCall  = 2
```

Options is set to apFirstCall the first time the protocol generates the event after being started by StartTransmit or StartReceive. Options is set to apLastCall the last time it generates the event, when the protocol is finished. Options equals zero for all other times.

The rest of the information about protocol progress is obtained by reading the values of various APAXPort properties, including:

BlockCheckMethod: the type of block check calculation used by the protocol.

BlockErrors: the number of errors for the current block. This is the number of times the protocol has unsuccessfully tried to transmit or receive the current block. It is reset to zero when the block is finally accepted.

BlockLength: the current transfer block length. Although this value is usually static, some protocols modify the length of the block on the fly. Zmodem in particular reduces the block length after several block errors in a row and raises it again after several good blocks.

BlockNumber: the number of blocks transmitted so far. This is obtained by dividing the number of bytes transferred by the current block length, so it will change if the block length changes.

BytesRemaining: the size of the file minus BytesTransferred. When the file size isn't known, BytesRemaining returns zero.

BytesTransferred: the number of bytes transmitted or received so far. When transmitting, this number is sometimes only an estimate. The uncertainty comes from the fact that the protocol window doesn't know when a particular byte has actually been transferred. BytesTransferred is the number of bytes the protocol window has transferred to the output buffer of the communications driver, minus the number of bytes that the driver reports are currently in the buffer. Unfortunately, this calculation is still imperfect because it's impossible to know how much of the output buffer holds actual file data and how much holds overhead characters needed by the protocol. Each protocol has a few simple rules it uses to estimate this proportion, which in practice yield good estimates.

ElapsedTicks: the number of ticks elapsed since the protocol started. In order to provide accurate CPS values, the protocol engine doesn't start the timer until it receives the first block from the remote computer.

FileDate: the date and time of the file being transmitted or received. If the protocol does not support this feature, FileDate returns zero.

FileLength: the size of the file being transmitted or received. For transmitted files the file size is always known. For received files the file size is known only if the protocol supports this feature and the receiver has received this information. If the file size is not known, FileLength returns zero.

FileName: the fully qualified name of the file that is being received or transmitted. When receiving with a protocol that does not transfer the file name, FileName simply returns the name previously assigned to it.

InitialPosition: used only for resumed file transfers using the Zmodem protocol. To display an accurate transfer rate (CPS, or character per second, rate), status routines for these protocols must subtract InitialPosition from BytesTransferred to obtain the actual number of bytes transferred during this session. If this is not a resumed file transfer, InitialPosition returns zero.

ProtocolStatus: a code that indicates the current state of the protocol. The following table shows all of the possible values. The usual status value is psOK, which means that the protocol is operating normally. Other status values indicate recoverable error conditions, protocol resume conditions, protocol start-up states, and internal protocol states. Fatal protocol errors are not represented by protocol states.

Status Code	Value	Explanation
psOK	0	Protocol is OK
psProtocolHandshake	1	Protocol handshaking in progress
psInvalidDate	2	Bad date/time stamp received and ignored
psFileRejected	3	Incoming file was rejected
psFileRenamed	4	Incoming file was renamed
psSkipFile	5	Incoming file was skipped
psFileDoesntExist	6	Incoming file was skipped; doesn't exist on sender's hard disk
psCantWriteFile	7	Incoming file skipped due to Zmodem options
psTimeout	8	Timed out waiting for something
psBlockCheckError	9	Bad checksum or CRC
psLongPacket	10	Block too long
psDuplicateBlock	11	Duplicate block received and ignored
psProtocolError	12	Error in protocol
psCancelRequested	13	Cancel requested
psEndFile	14	At end of file
psSequenceError	16	Block was out of sequence
psAbortNoCarrier	17	Aborting on carrier loss
psGotCrcE	18	Received Zmodem CrcE packet
psGotCrcG	19	Received Zmodem CrcG packet
psGotCrcW	20	Received Zmodem CrcW packet
psGotCrcQ	21	Received Zmodem CrcQ packet

ProtocolType: the protocol type, which is one of ptXmodem, ptXmodemCRC, ptXmodem1K, ptXmodem1KG, ptYmodem, ptYmodemG, ptZmodem, ptKermit, and ptAscii.

TotalErrors: the number of errors encountered since the current file was started. It is reset only when a new file is started.

Various properties that describe the option settings for the protocol may also be used within the status routine. These include HonorDirectory, IncludeDirectory, RTSLowForWrite, AbortNoCarrier, and other options that are specific to particular protocols.

The StatusInterval property, which defaults to 18, is the maximum number of ticks between OnProtocolStatus events. The protocol generates an OnProtocolStatus event after every significant event (received a file name, received a complete block, etc.) or after at most StatusInterval ticks.

APAX includes a mechanism for providing an automatic protocol status display without programming, through the ProtocolStatusDisplay property. For each OnProtocolStatus event the protocol checks whether ProtocolStatusDisplay is set to True. If it is, the protocol automatically updates the display. It then calls the OnProtocolStatus event if one was implemented.

Protocol logging

File transfer is often an automated process. For example, an application might send all of the day's transaction files to a remote computer during the night. In this case the application would also keep a record of the files that were successfully transmitted and those that weren't.

The APAX protocol logging feature is ideal for this kind of application. It provides the opportunity to log information about each received or transmitted file and whether the transfer succeeded.

To support logging, the protocol component generates an OnProtocolLog event at the start and end of each file transfer. The event passes a parameter that identifies the current logging action.

The logging routine isn't limited to just writing status information. It also can take care of file-related start-up and cleanup activities. One example of this is to delete partially received files. You would probably want to do this for all protocols except Zmodem, which can resume failed transfers from the point of the error without having to retransmit the entire file.

Multiple file processing

Several of the protocols provided by APAX can transmit and receive batches of files. The protocol engine determines the next file to send with DOS filemask processing, using the mask assigned to the SendFileName property.

For non-batch protocols like Xmodem the file mask should not contain wildcards. Such protocols are capable of transmitting only a single file at a time, and if the mask matches more than one file only the first matching file is transmitted.

Accept file processing

When receiving files, there may be times when you don't want the incoming file. Consider, for example, an open BBS where a first time caller is attempting to upload a 10MB file at 2400 baud. Since this would tie up the BBS for more than 11 hours you probably would want to refuse it immediately. If the caller is using a protocol that transmits the file size in advance, you can detect that it's bigger than you want and refuse the upload.

As another example, suppose that a BBS has a well-publicized rule that it accepts only LZH uploads and it detects that an incoming file has a ZIP extension. If a caller is using a protocol that transmits the file name in advance, you can refuse an upload immediately. The OnProtocolAccept event can be used to build such behavior into your application.

Note that Zmodem, alone among the APAX protocols, has built-in functionality for certain kinds of accept file functions. For example, it can reject an upload that would overwrite an existing file, or accept it only if the upload's time stamp is newer than the existing file. These options are described fully in the Zmodem section later in this chapter.

Once the protocol knows the name of an incoming file, but before it starts receiving data, it generates an OnProtocolAccept event. An application can respond to the event by setting the Accept parameter to True to accept the file, or False to refuse it. By default, all files are accepted.

For all protocols except Zmodem, the first rejected file terminates the entire batch transfer. Zmodem has provisions for skipping files, and the transmitter picks up again with the next file after the rejected one.

The OnProtocolAccept event also provides an opportunity to rename an incoming file if its current name isn't acceptable. For example, if a file name conflicts with an existing file, you can accept the file but change its name.

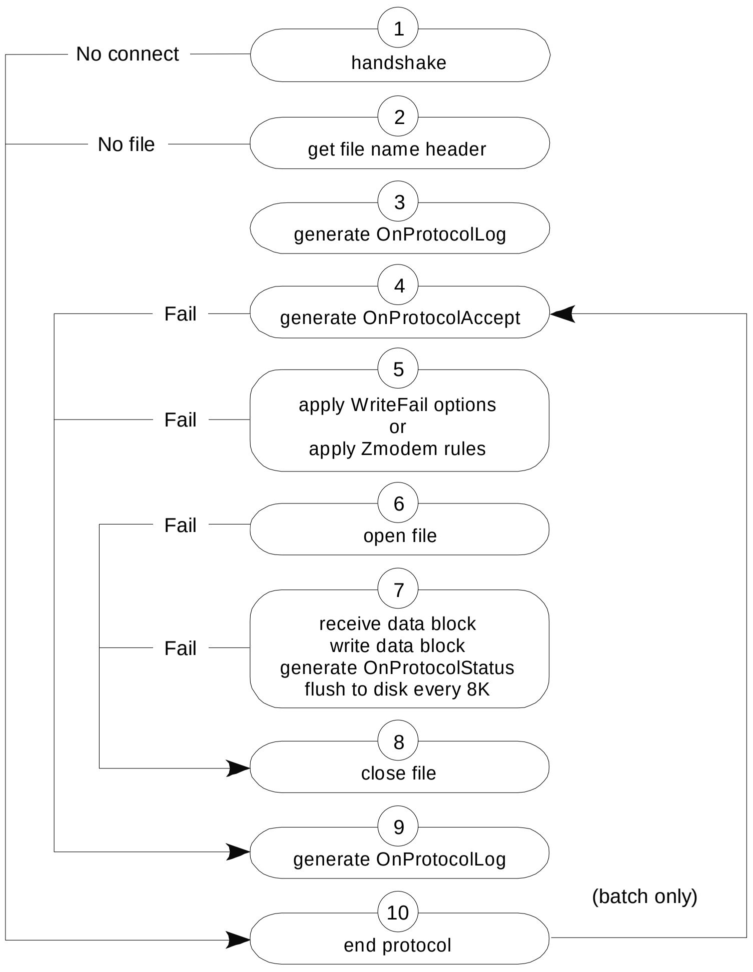
Note that all protocols have built-in options for handling incoming file name collisions. See the "WriteFailAction property" on page 292 for a complete description. You don't need to write an OnProtocolAccept event handler if these constants provide the needed behavior.

Internal logic

The protocol component has so far been described as a black box—you initialize it and call StartTransmit or StartReceive to perform the protocol magic. Now it is time to look inside the box, at how the protocol engine works. With this additional information, you will be able to use the protocols more effectively and take better advantage of the events generated by the protocol window.

Receiving Files

When receiving files, the protocol window employs the following logic. The following diagram generally applies to all protocols.



The protocol first attempts to handshake with the remote machine. A handshake consists of a valid response to an initial character sequence sent by the transmitter. If the handshake is unsuccessful after a specified number of retries, the protocol ends.

If the handshake is successful, the transmitter is asked for the name of the next file to transmit. It responds by sending a block containing the name.

The protocol generates the OnProtocolLog event to give the application an opportunity to record the file name or take any other special action needed at the start of the transfer. Note that the logging routine receives the file name before the OnProtocolAccept event handler has had a chance to modify it.

Next the protocol generates the OnProtocolAccept event. If the message handler sets Accept to False, the file is skipped and control is transferred to step 9. Otherwise, the built-in WriteFail options or Zmodem's built-in file management rules are applied. If the protocol fails at this point, control is transferred to step 9.

The received file is created using the name from the file name header, perhaps as modified by the OnProtocolAccept event handler. Step 6 also allocates work buffers and initializes several internal variables used to manage the receive buffer. If the open fails, control is transferred to step 8, which disposes of buffers and closes the file.

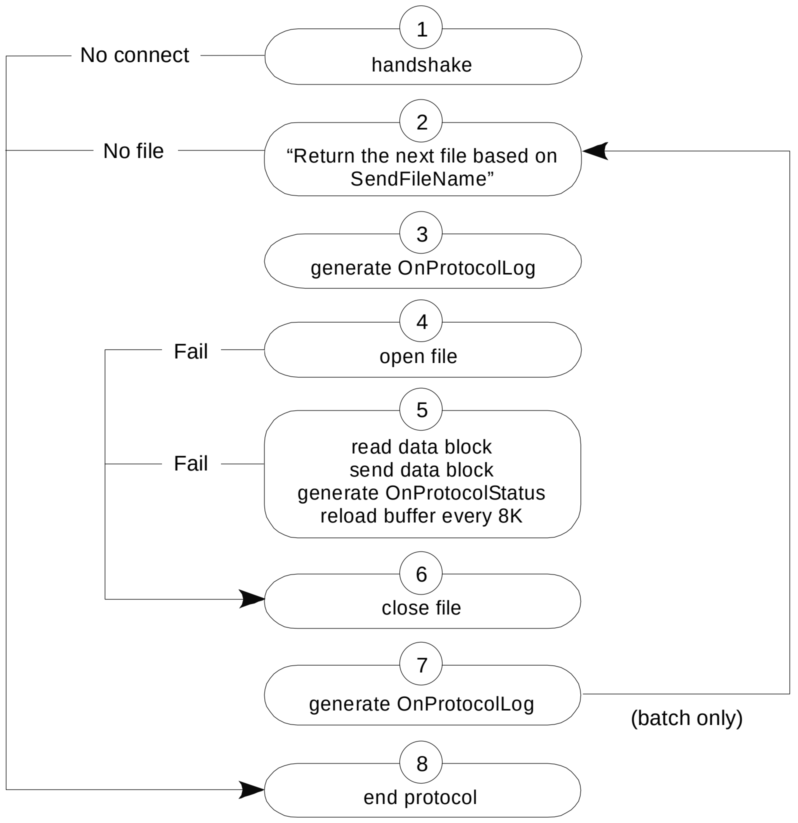
The actual transfer of data comes next in step 7. The internal operations of this step vary tremendously among the protocols, so it is condensed in this diagram. The OnProtocolStatus event is generated at least once for each block received. If unrecoverable errors occur for any reason (user abort, broken connection, disk full, etc.), control is transferred to step 8.

After the file transfer is complete, the file is closed in step 8. Then the OnProtocolLog event is generated with information regarding whether the file was received OK, rejected, or failed.

In a batch protocol, control returns to the top of the loop to get another file header. If one is received, the whole process is repeated. Otherwise, the protocol is ended by coordinating with the transmitter and cleaning up.

Transmitting Files

When transmitting files, the protocol window employs the following logic. This diagram generally applies to all protocols.



The protocol first attempts to handshake with the remote machine. A handshake consists of sending an initial character sequence and waiting for a valid response. If the handshake is unsuccessful after a specified number of retries, the protocol ends.

If the handshake is successful, the component returns the next file based on the SendFileName property.

The protocol generates the OnProtocolLog event to give the application an opportunity to record the outgoing file name or take any other special action needed at the start of the transfer.

The outgoing file is opened in step 4. This step also allocates work buffers and initializes variables used to manage the send buffer. If any of these steps fail, control is transferred to step 6 to clean up.

The actual transfer of data comes next in step 5. The file is read in 8K byte blocks and sent using the block size native to the protocol. The OnProtocolStatus event is generated at least once for every block sent. If unrecoverable errors occur, control is immediately transferred to step 6.

After the file transfer is complete, the file is closed and buffers are disposed in step 6. Then the OnProtocolLog event is generated with information regarding whether the file was transferred OK, rejected, or failed.

In a batch protocol, control returns to the top of the loop to get another file to send. If one is available, the whole process is repeated. Otherwise, the protocol is ended by coordinating with the receiver and cleaning up.

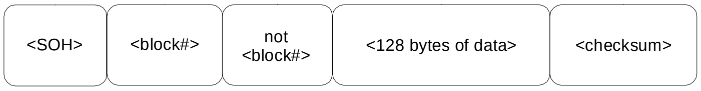
Xmodem

Xmodem is the oldest protocol supported by APAX. It was developed and first implemented by Ward Christensen in 1977 and placed in the public domain. Since then, it has become an extremely popular protocol and continues in use today (although at a diminished frequency).

Xmodem is also the simplest, and perhaps the slowest, protocol supported by APAX. Xmodem uses blocks of only 128 bytes and requires an acknowledgment of each block. It uses only a simple checksum for data integrity.

What follows is a simplified description of the Xmodem protocol, although it describes far more than is required to actually use the protocol in APAX.

Xmodem blocks have the following format:



The <SOH> character marks the start of the block. Next comes a one byte block number followed by a ones complement of the block number. The block number starts at one and goes up to 255 where it rolls over to zero and starts the cycle again. Following the block numbers are 128 bytes of data and a one-byte checksum. The checksum is calculated by adding together all the data bytes and ignoring any carries that result.

A typical Xmodem protocol transfer looks something like this:

Transmitter		**Receiver**
	`<---`	`<NAK>`
`<SOH><1><254><128 data bytes><chk>`	`--->`	
	`<---`	`<ACK>`
`<SOH><2><253><128 data bytes><chk>`	`--->`	
	`<---`	`<ACK>`
`<EOT>`	`--->`	
	`<---`	`<ACK>`

The receiver always starts the protocol by issuing a <NAK>, also called the handshake character. It waits 10 seconds for the transmitter to send a block of data. If it doesn't get a block within 10 seconds, it sends another <NAK>. This continues for 10 retries, after which the receiver gives up.

If the receiver does get a block, it compares the checksum it calculates to the received checksum. If the checksums differ, the receiver sends a <NAK> and the transmitter resends the block. If the checksums match, the receiver accepts the block by sending an <ACK>. This continues until the complete file is transmitted. The transmitter signifies this by sending an <EOT>, which the receiver acknowledges with an <ACK>.

Either side can cancel the protocol at any time by sending three <CAN> characters (^X). However, during an Xmodem receive the receiver cannot tell whether the <CAN> characters are real data or a cancel request. The sequence is recognized as a cancel request only when it comes between blocks. Hence, more than three <CAN> characters are sometimes required to cancel the receiver. Sufficient characters are required to complete the current block, then three more <CAN> characters to cancel the protocol.

From this description several things become clear. First, this protocol does not transfer any information about the file being transmitted. Hence, the receiver must assign a name to the incoming file.

The receiver also doesn't know the exact size of the file, even after it is completely received. The received file size is always a multiple of the block size. This Xmodem implementation fills the last partial block of a transfer with characters of value BlockFillChar, whose default is ^Z.

Xmodem often exhibits a start-up delay. The transmitter always waits for a <NAK> from the receiver as its start signal. If the receiving program was started first, the transmitter probably missed the first <NAK> and must wait for the receiver to time out and send another <NAK>.

Xmodem offers no escaping of binary control characters. Escaping means that characters can be transformed before being transmitted to prevent certain binary data characters, such as <XON>, from being interpreted as data link control characters. As a result, you can't use software flow control in an Xmodem transfer (since the flow control software would misinterpret <XON> or <XOFF> characters in the data stream as flow control requests) and Xmodem itself can't tell the difference between <CAN> characters used for protocol cancellation and for data.

The only merit of the basic Xmodem protocol is that it is so widespread that it's probably supported by any communications program you can find, thus providing a lowest common denominator between systems.

Xmodem extensions

Xmodem has been tweaked and improved through the years. Some of these variations have become standards of their own and are supported by APAX. These Xmodem extensions are treated as separate protocols enabled by assigning a different value to the Protocol property.

The first of these improvements is called Xmodem CRC, which substitutes a 16 bit CRC (cyclic redundancy check) for the original checksum. This offers a much higher level of data integrity. When given the opportunity, you should always choose Xmodem CRC over plain Xmodem.

The receiver indicates that it wants to use Xmodem CRC by sending the character ‘C’ instead of <NAK> to start the protocol. If the transmitter doesn’t respond to the ‘C’ within three attempts, the receiver assumes the transmitter isn’t capable of using Xmodem CRC. The receiver automatically drops back to using checksums by sending a <NAK>.

Another popular extension is called Xmodem 1K. This derivative builds on Xmodem CRC by using 1024 byte blocks instead of 128 byte blocks. When Xmodem 1K is active, each block starts with an <STX> character instead of an <SOH>. Xmodem 1K also uses a 16 bit CRC as the block check.

A larger block size can greatly speed up the protocol because it reduces the number of times the transmitter must wait for an acknowledgment. However, it can actually reduce throughput over noisy lines because more data must be retransmitted when errors are encountered. The implementation of Xmodem 1K in APAX drops back to 128 byte blocks whenever it receives more than 5 <NAK> characters in a row. Once it drops back to 128 byte blocks, it never tries to step back up to 1024 byte blocks.

The final Xmodem extension supported by APAX is Xmodem 1KG. This streaming protocol is requested when the receiver sends ‘G’ as the initial handshake character instead of <NAK>. Streaming in this context means that the transmitter continuously transmits blocks without waiting for acknowledgments. In fact, all blocks are assumed to be correct and the receiver never sends acknowledgments. If the receiver does encounter a bad block, it aborts the entire transfer by sending a <NAK>.

You shouldn’t even consider using this streaming protocol unless you are using error correcting modems with their error control features turned on. In fact, you might want to have your application refuse to use Xmodem 1KG if error correcting modems aren’t detected. The advantage of a streaming protocol like Xmodem 1KG is its very high throughput. There is no acknowledgment delay, so the protocol is extremely efficient. Zmodem has this same property, but can also retry and recover from errors.

Ymodem

Ymodem is a derivative of Xmodem that is different enough to be called a unique protocol. What follows is a simplified explanation of Ymodem that provides more than enough information to use it with APAX.

Ymodem is essentially Xmodem 1K with batch facilities added, which means that a single protocol session can transfer as many files as you care to transmit. Another important enhancement is that the transmitter can provide the receiver with the incoming file name, size, and timestamp.

Ymodem achieves this by adding block zero to the Xmodem 1K protocol. Block zero is transferred first and contains file information in the following format:

<SOH>	<0>	<255>	<name> <0> <len> <0> <date> <0>...<0>	<CRChi> <CRClo>

The <name> field is the only required field. It supplies the name of the file in lower case letters. Path information can be included but the protocol requires that all '\' characters be converted to '/'.

The <len> field specifies the file length as an ASCII string. This field allows the receiver to truncate the received file to discard the filler characters at the end of the last block.

The <date> field is the date and time stamp of the file. It is transmitted as the number of seconds since January 1, 1970 GMT, expressed in ASCII octal digits (a Unix convention).

APAX takes care of properly formatting this block. You don't need to do anything but specify the name of the file to transmit. When receiving Ymodem files, APAX uses the information in this block, if present, to adjust the size of the received file and its modification date.

Here's a sample Ymodem session:

Transmitter		**Receiver**
	`<---`	`'C'`
`<SOH><0><255><file info><crc>`	`--->`	
	`<---`	`<ACK>`
	`<---`	`'C'`
`<STX><1><254><1024 data bytes><crc>`	`--->`	

Transmitter		Receiver
	<---	<ACK>
<STX><2><253><1024 data bytes><crc>	--->	
	<---	<ACK>
<EOT>	--->	
	<---	'C'
<SOH><0><255><file info><crc>	--->	
	<---	<ACK>
	<---	'C'
<STX><1><254><1024 data bytes><crc>	--->	
	<---	<ACK>
<EOT>	--->	
<SOH><0><255><128 zeros><crc>	--->	
	<---	<ACK>

As with the Xmodem protocols, the Ymodem protocol starts when the receiver sends a handshake character (‘C’) to the transmitter. The transmitter responds with a properly formatted block zero. The receiver acknowledges this with an <ACK> and then starts a normal Xmodem CRC protocol by issuing another ‘C’ handshake character.

Ymodem extensions

The Ymodem specification permits Ymodem to use a combination of 128 and 1024 byte blocks. Most Ymodem protocols start with 1024 byte blocks and drop back to 128 byte blocks only if repeated errors are detected. Once the block size is reduced to 128 bytes, it is never stepped back up to 1024.

Like Xmodem, Ymodem also offers a streaming extension called Ymodem G. This is similar in performance (and drawbacks) to Xmodem 1KG, but like Ymodem itself offers the advantages of batch transfers and file information.

Zmodem

Of all the protocols supported by APAX, Zmodem offers the best overall mix of speed, features, and tolerance for errors. The Zmodem protocol has many options and clearly was meant to have lots of room for growth. The APAX implementation of Zmodem does not cover the entire protocol specification but it does implement the features most likely to be required by your application. It should generally be your protocol of choice.

Zmodem was developed for the public domain by Chuck Forsberg under contract to Telenet. The original purpose was to provide a durable protocol with strong error recovery features and good performance over a variety of network types (switched, satellite, etc.). It has generally achieved these design goals.

What follows is a simplified explanation of Zmodem that provides more than enough information to use it with APAX.

Zmodem borrows some concepts from Xmodem, Ymodem, and Kermit but is really a completely new protocol. Instead of adopting the simple block structure of Xmodem and Ymodem, Zmodem employs headers, data subpackets, and frames. A header contains a header identifier, a type byte, four information bytes, and some block check bytes. A data subpacket contains up to 1024 data bytes, a data subpacket type identifier, and some block check bytes. A frame consists of one header and zero or more data subpackets.

Due to the complexity and variety of the Zmodem header and data subpacket formats, they are not all detailed here. Instead, here is a high level look at a sample Zmodem file transfer:

Sender		**Receiver**	**Explanation**
`'rz'<cr>`	`--->`		Start marker for automated transfers
`ZrQinit`	`--->`		Request for receiver's information
	`<---`	`ZrInit`	Receiver answers with its options
`ZFile`	`--->`		Transmitter sends file information
	`<---`	`ZrPos`	Receiver sets the starting filepos
`ZData`	`--->`		Transmitter says file data to follow
`data subpacket`	`--->`		Transmitter sends a data subpacket
`...`			Continues until all data sent
`ZEof`	`--->`		Transmitter indicates end-of-file
	`<---`	`ZrInit`	Receiver says ready for next file

Sender		Receiver	Explanation
ZFin	--->		Transmitter indicates no more files
	<---	ZFin	Receiver acknowledges
'OO'	--->		Transmitter signs off

The ZXxx tags are the header types that the two computers trade back and forth as they decide what is to be done. In most cases all data in the file is sent in one ZData frame (the ZData header followed by as many data subpackets as required). The receiver doesn't have to acknowledge any of the blocks unless the transmitter specifically asks for it. The Zmodem protocol as implemented by APAX never asks for an acknowledgment; however, it respects such requests from the transmitter.

Typically, once a file transfer is underway, the receiver interrupts the transmitter only if it receives a bad block as determined by comparing block check values. An error is reported by sending a ZrPos header, telling the transmitter where in the file to start retransmitting.

The protocol can be canceled at any time if either side sends five <CAN> characters (^X).

Control character escaping

Zmodem escapes certain control characters. Escaping means that characters are transformed before being transmitted to prevent certain binary data characters, such as <XON> and <CAN>, from being interpreted as data link control characters.

Escaping isn't something you need to enable or disable because it's always on. It is mentioned here because escaping is what permits you to use software flow control with Zmodem. That isn't possible with the Xmodem/Ymodem family.

Zmodem always escapes the following characters:

```
<DLE>   Data link escape character (10h, ^P)
<XON>   XOn character (11h, ^Q)
<XOFF>  XOff character (13h, ^S)
<CAN>   Zmodem escape character (18h, ^X)
<DLE*>  Data link escape character with high bit set (90h)
<XON*>  XOn character with high bit set (91h)
<XOFF*> XOff character with high bit set (93h)
```

Zmodem escapes all control characters when requested to by the remote protocol.

Protocol options

While the Zmodem specification describes all sorts of features, not all Zmodem implementations are expected to support all of the features. One of the first things that happens in a Zmodem protocol is that the receiver tells the transmitter what features it supports. The transmitter might modify its standard behavior to accommodate the receiver's support (or lack of support) for a particular option.

Since this process is handled automatically, you generally don't need to worry about it. For your information, the protocol options that APAX Zmodem protocol engine provides and doesn't provide are listed here.

APAX supports the following Zmodem protocol options:

- True full duplex for data and control channels
- Receiving data during disk I/O
- Sending a break signal
- Using 32 bit CRCs
- Escaping all control characters

APAX does not support the following protocol options:

- Encryption
- LZ data compression
- Escaping the 8th bit
- End-of-line conversion for Unix newline characters
- Sparse files

Transfer resume

The Zmodem specification describes an option called "recover/resume." This option is requested by the transmitter when it wants to resume a previously interrupted file transfer. When the receiver sees the request for this option, it compares the incoming file name with the files in its destination directory. If the incoming file already exists and is smaller than the one being transmitted, the receiver assumes that the transmitter wants to transfer only the remaining portion of the file.

When this condition exists, the receiver opens the existing file and moves the file pointer to the end of the file. It then tells the transmitter to move its file pointer to the same point in its copy of the file. The transmitter starts sending data from that point, which resumes the transfer from where it was interrupted.

This option can also be used to append new data to a remote copy of a file.

In either case, you use this option as follows:

```
APAXPort1.SendFileName = "BIGFILE"
APAXPort1.ZmodemRecover = True
APAXPort1.StartTransmit
```

File management options

Zmodem has a variety of file management options built into it. These are simple rules that tell Zmodem whether or not to accept a file. Here are the possible options:

Option Code	Explanation
zfoWriteNewerLonger	Transfer if new, newer, or longer
zfoWriteCrc	Not supported, interpreted same as zfWriteNewer
zfoWriteAppend	Transfer if new, append if existing
zfoWriteClobber	Transfer always
zfoWriteNewer	Transfer if new or newer
zfoWriteDifferent	Transfer if new or different dates or sizes
zfoWriteProtect	Transfer only if new

The zfoWriteCrc option, which requests that a file be transferred only if its CRC is different from the remote copy's, is not supported. When this option is requested, it is treated the same as the zfoWriteNewer option.

The file management options are always requested by the transmitter. To use them, assign a value to the ZmodemFileOptions property before calling StartTransmit. The default behavior is zfoWriteNewer. For example, to transmit all files regardless of whether such files already exist on the remote machine, make the following assignment:

```
APAXPort1.ZmodemFileOptions = zfoWriteClobber
```

Even though the transmitter sets the file management options, APAX allows the receiver to change them. For example, suppose the transmitter has requested zfoWriteClobber but you want to accept only newer files. In this case you would set the ZmodemOptionOverride property to True before calling StartReceive:

```
APAXPort1.ZmodemOptionOverride = True
APAXPort1.ZmodemFileOptions = zfoWriteNewer
```

Setting this property to True tells Zmodem to ignore the file management options requested by the transmitter and to use zfoWriteNewer instead.

Another file management property called ZmodemSkipNoFile is available. Set this property to True to force the receiver to skip any incoming file that doesn't already exist in the destination directory.

Whatever file management rules are in effect, the receiver applies them and either accepts each file or rejects it. If the file is accepted, the file transfer proceeds normally. If the file is rejected, the receiver sends a ZSkip frame to the transmitter, which stops sending the current file and moves on to the next one in its list.

Don't forget that you can implement your own file management rules with an OnProtocolAccept event handler.

Automatic block size control

The Zmodem protocol decreases or increases the number of bytes transmitted per block in response to retransmission requests, usually due to poor line conditions or random line errors. The rationale is that small blocks can transmit more frequently without errors, since there's less time for a small block to be hit by line noise. And, even if a small block is corrupted by noise, it is faster to retransmit than a large block.

If the transmitter receives an unsolicited request from the receiver to resend data, it reduces the block size from 1024 to 512. If the transmitter receives another request to resend, it reduces the block size from 512 to 256. It never reduces the block size below 256 bytes. Conversely, the transmitter raises the block size immediately back to 1024 bytes when it sends four blocks in a row without receiving any requests to resend.

Similar logic is employed with 8K Zmodem, which uses 8192 byte blocks by default. The block size is halved for each retransmission request received, down to a minimum of 256 bytes. The block size is increased to 8K bytes in a single step after four blocks are transmitted without any requests to resend.

Block size control is automatic and cannot be disabled. While this behavior is not documented in the public domain Zmodem specification, it is the process followed by the popular DSZ program and is acceptable to any common Zmodem implementation.

Large block support

The APAXPort's implementation of Zmodem also includes support for 8K byte blocks. This behavior is outside the public domain specification and was added largely for programmers who need to transfer files to or from several popular BBS and FIDONet mailer programs. Since large blocks are not supported by all common Zmodem implementations, their use is not automatic—you must specifically enable them before starting a file transfer by setting the Zmodem8K property to True.

Kermit

The Kermit protocol was developed to allow file transfers in environments that other protocols can't handle. Such environments include links that only pass 7 data bits, links that can't handle control characters, computer systems that can't handle large blocks, and diverse other links such as those between a PC and a mainframe.

Kermit is a public domain protocol that was developed at Columbia University. (The name refers to Kermit the Frog, from The Muppet Show.) What follows is a simplified explanation of Kermit that provides more than enough information to use it with APAX. For additional details, get the Kermit protocol specification from Columbia University, Kermit Distribution, Department OP, 612 West 115th Street, New York, NY 10025.

Character quoting

Character quoting means pretty much the same thing that escaping means in Zmodem. The character is replaced by a quote character and a modified form of the original character. The quote character tells the receiver how to convert the modified character back to its original value. Quoting ensures that certain binary characters are never put into the data stream where they could be misinterpreted by a modem or another part of the serial link.

Although Zmodem transforms only a few critical characters such as <XON> and <XOFF>, Kermit quotes nearly all characters. This is one of the features that permits Kermit to run in nearly any environment. When quoting is finished, a Kermit data packet consists almost entirely of printable ASCII characters. The only exceptions are an <SOH> character at the start of each packet and a <CR> at the end.

Kermit quotes control characters by replacing them with a quote character and a modified version of the control character. For example, ^A becomes '#A' where '#' is the quote character. The process of converting ^A to 'A' is called "Ctl" and it works like this:

```
Ctl(x) = x xor 40h
```

This operation is its own inverse, that is, Ctl(Ctl(x)) = x.

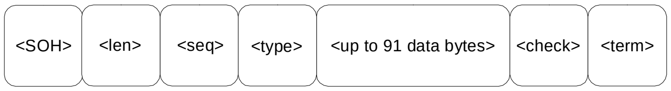
Kermit also quotes characters with their eighth bit set, which allows it to transmit 8 bit data over 7 bit data links. The quote character in this case is '&' and the quoted data character is obtained simply by stripping the high bit. For example, the quoted version of character $C1 ('A' with its high bit set) is '&A'.

Binary numbers in Kermit packet headers and in repeated character strings are also transformed to assure that they are printable characters. This is achieved by adding 32 to each number before it is transmitted and subtracting 32 after it is received. In Kermit parlance, these operations are known as “ToChar” and “UnChar.”

Kermit has a simple built-in data compression mechanism called run length encoding. When it sees a long string of repeated characters, it compresses the string into a quote character, a length byte, and the repeated character. Obviously, there must be at least 4 repeated characters before there is any compression. The quote character for run length encoding is ‘~’. Hence, the string ‘AAAAA’ becomes ‘~%A’, where ‘%’ is equivalent to a binary 5 after the “ToChar” operation.

Kermit packets

The following diagram shows the general format of a Kermit packet:



The <SOH> character, also called the mark field, indicates the start of a Kermit packet.

The length byte specifies the number of bytes that follow. Since it must be transmitted as a printable 7 bit character the binary maximum value is 94, which means that the maximum length of a normal Kermit packet is 96 bytes including the <SOH> and the <length> field.

The <seq> byte is a packet sequence number in the range of 0 to 63. After 63 it cycles back to 0.

The <type> byte describes the various Kermit packet types, which are analogous to the Zmodem frame types.

The data field contains up to 91 bytes including all quote characters. The number of actual data bytes could be considerably less, particularly if binary data is being transmitted.

The standard Kermit <check> field is a single-byte checksum. Kermit offers two optional block check methods called two-byte checksum and three-byte CRC.

The <term> character is the packet terminator which equals carriage return (ASCII 13) by default. You will probably never need to change the terminator.

A typical Kermit protocol transfer looks like this:

Transmitter		**Receiver**	**Explanation**
KSendInit	--->		Transmitter sends its options
	<---	KAck	Receiver answers with its options
KFile	--->		Transmitter sends filename
	<---	KAck	Receiver acknowledges filename
KData	--->		Transmitter sends data packet
	<---	KAck	Receiver acknowledges data packet
KData	--->		Transmitter sends data packet
	<---	KAck	Receiver acknowledges data packet
...			Continues until all data sent
KEndoffile	--->		Transmitter says end of file
	<---	KAck	Receiver acknowledges and closes file
KBreak	--->		Transmitter says end of protocol
	<---	KAck	Receiver acknowledges end of protocol

The KXxx tags are the packet types that the two computers exchange as they decide what is to be done.

Kermit options

Like Zmodem, Kermit offers a variety of options. An implementation of Kermit is not required to support all options. Hence, one of the first things that happens in a Kermit protocol is that the two sides exchange their desired options and use the lowest common denominator of the two sets.

Here are the Kermit options that APAX supports and the default values it uses. The entries in the first column are APAX property names that can be used to adjust each option.

Property	Default	Explanation
KermitMaxLen	80 bytes	Maximum length of the data field
KermitTimeoutSecs	5 seconds	Maximum time-out between characters
KermitPadCount	0 bytes	No pad characters before packets
KermitPadCharacter	' '	Space character used for padding
KermitTerminator	<CR>	Terminator is a carriage return
KermitCtlPrefix	'#'	Control character prefix is '#'
KermitHighbitPrefix	'Y'	8-bit quoting honored, not required
BlockCheckMethod	'1'	Use a 1 byte checksum
KermitRepeatPrefix	'~'	Repeat prefix is '~'
KermitMaxWindows	0	No sliding windows

KermitMaxLen is the maximum number of bytes you want Kermit to include in one packet. The normal maximum value is 94; the default value is 80 as suggested by the Kermit Protocol Manual. If KermitMaxLen exceeds 94, the Kermit "long packets" feature is enabled. (see page 241.) The absolute maximum value is 1024.

KermitTimeoutSecs is the amount of time, in seconds, that a Kermit transmitter will wait for an acknowledgment or a Kermit receiver will wait for the next byte to be received. If more than KermitTimeoutSecs seconds elapse without receiving anything, Kermit assumes an error occurred and resends.

KermitPadCount and KermitPadCharacter describe padding that can be added at the front of all Kermit packets. The only reason for padding is if the remote machine needs a delay between sending a packet and receiving a response. In this case, you can specify enough padding characters to generate the required delay. Generally, though, padding is unnecessary. The Kermit protocol as implemented by APAX automatically honors a remote's request for padding.

KermitTerminator is the character that follows the check field in a packet. Although all Kermit packets have a terminator, it is used only by systems that need an end-of-line character before they can start processing input.

KermitCtlPrefix is the control character prefix that Kermit uses when performing "Ctl" quoting to transform control characters into printable ASCII characters. Generally you won't need to change this prefix.

KermitHighbitPrefix specifies how Kermit transforms high-bit characters into characters without the high-bit set. Generally you won't need to change this setting. See the property description for more information.

BlockCheckMethod specifies the type of block checking Kermit should perform. '1' corresponds to the bcmChecksum value of the BlockCheckMethod type, and it means that Kermit should use a single-byte checksum. All Kermit implementations are guaranteed to support this form of block checking. bcmChecksum2 means that Kermit should use a two-byte checksum, which offers only slightly more protection than the single-byte checksum. bcmCrcK means that Kermit should use a three-byte CRC. This is the preferred block check method because it offers the highest level of error detection. Unfortunately, not all Kermit implementations support the non-default block check methods. If the remote computer doesn't support the block check method you request, both sides drop back to the single-byte checksum.

KermitRepeatPrefix is the repeated-character prefix that Kermit uses when compressing long strings of repeated characters. Generally you won't need to change this prefix.

KermitMaxWindows is the number of sliding windows requested. Setting this to a value between 1 and 27 (the maximum allowed) enables sliding windows support.

The two sides of a Kermit protocol automatically negotiate which options to use, so no intervention is required by your program. If you wish to change the default options, use these properties.

APAX does not provide Kermit server functions and does not support file attribute packets.

Long packets

APAX includes support for long packets, which is an extension to standard Kermit that permits data packets of up to 1024 bytes. Long packets can substantially improve protocol throughput on clean connections that have small turnaround delays. Long packet support is turned off by default and must be enabled by setting KermitMaxLen to a value between 95 and 1024. Most other Kermit implementations also disable this option by default.

Although the specification allows for packets up to 9024 bytes, APAX limits long packets to 1024 bytes. Packets longer than 1024 bytes do not appreciably increase throughput, but they dramatically increase retransmission time when a line error occurs.

The specification also recommends the use of the higher-order checksums with long packets, but does not require it. APAX defaults to 2 byte checksums when long packet support is enabled, but drops back to 1 byte checksums if requested to do so by the remote Kermit.

Sliding windows control

APAX includes supports for the Kermit extension known as Sliding Windows Control (SWC), also called "SuperKermit." SWC provides a "send ahead" facility that dramatically improves throughput when turnaround delays tend to be large, as when using satellite links.

Send ahead means that the transmitter sends many blocks without waiting for an acknowledgment for each block. The transmitter collects acknowledgments when they eventually arrive and marks the previously transmitted blocks as acknowledged. This reduces turnaround delay (the time it takes the receiver to send an acknowledgment) to zero.

SWC operates by keeping a circular table of transmitted packets. The maximum number of packets in this table is called the window size, which is a number between 0 (no sliding window support) and 31. If the transmitter and receiver specify different window sizes, the smaller of the two is used. APAX's Kermit actually limits the maximum number of windows to 27 to avoid encountering a bug in the popular program MSKERMIT.

Sliding window support is off by default. It is enabled by setting the KermitMaxWindows property to a non-zero value.

On the sender's side, each transmitted packet is added to the table. When an acknowledgment is eventually received for a packet, its entry in the table is freed. If the table fills, the transmitter does not send more packets until it receives acknowledgments for one or more existing packets.

On the receiver's side, each received packet is added to the table and remains there until the table is full. Then the oldest packet is written to disk. When errors are detected, the receiver sends a <NAK> for each missed packet, starting past the last known good packet and continuing up to the most recently received packet.

It is possible to enable long packets and SWC simultaneously, but memory consumption rises dramatically from the single 80 byte buffer normally used by Kermit. In the worst case you could have 27 windows of 1024 bytes each, adding up to 27648 bytes.

ASCII

The term “ASCII protocol” is a bit of a misnomer, because in an ASCII transfer neither side of the link is following well documented rules. An ASCII protocol is really just a convenient way of transmitting a text file.

A typical use for the ASCII protocol is when you need to transfer a text file to a remote machine that doesn’t have any protocols available. One way of accomplishing this is to run an APAX program that supports a terminal window and the ASCII protocol. You connect to the remote machine, navigate to the it’s editor, and open up a new text file. Then you start an ASCII protocol transmit of the file you need to transfer. The remote machine’s editor sees this as keystroke input to the editor. You finish the transfer by saving the editor’s file.

The ASCII protocol provides options for tailoring such transfers to the remote machine’s speed, which might necessitate delays between transmitted characters and lines. For example, when transmitting a file into a remote computer’s editor, you might need to use delays to avoid overflowing the editor’s keystroke buffer.

It is difficult for the receiver to know when an ASCII transfer is over because there is no agreed upon method for indicating termination. The ASCII protocol terminates on any of three conditions: when it receives a ^Z character, when it times out waiting for more data, or when the user aborts the protocol and the application calls CancelProtocol. When any of these conditions is detected, the file is saved and the protocol ends.

End-of-line translations

Computer systems sometimes use different character sequences to terminate each line of a text file. Most PC software stores both a carriage return <CR> and a line feed <LF> at the end of each line. Other systems store only <LF> or only <CR> at the end of each line.

The ASCII protocol provides a number of options for translating from one end-of-line sequence to another, both when transmitting and when receiving. When performing these translations, the <CR> and <LF> characters are treated separately, based on the values assigned to the AsciiCRTranslation and AsciiLFTranslation properties. The following enumerated values are used to control the behavior:

Value	**Explanation**
aetNone	The character is not to be modified (the default).
aetStrip	The character is to be stripped from the data stream.
aetAddCRBefore	A <CR> is to be inserted before each <LF>. This can be specified only for the AsciiLFTranslation property.
aetAddLFAfter	An <LF> is to be added after each <CR>. This can be specified only for the AsciiCRTranslation property.

Protocol References

Following is a list of the APAXPort control's properties, methods, and events that pertain to file transfer protocols. This is only a subset of the functionality of the APAXPort functionality. Additional properties, methods, and events are introduced in other chapters.

Properties

- AbortNoCarrier
- AsciiCharDelay
- AsciiCRTranslation
- AsciiEOFTimeout
- AsciiEOLChar
- AsciiLFTranslation
- AsciiLineDelay
- AsciiSuppressCtrlZ
- Batch
- BlockCheckMethod
- BlockErrors
- BlockLength
- BlockNumber
- BytesRemaining
- BytesTransferred
- ElapsedTicks
- FileDate
- FileLength
- FinishWait
- HandshakeRetry
- HandshakeWait
- HonorDirectory
- IncludeDirectory
- InitialPosition
- InProgress
- KermitCtlPrefix
- KermitHighbitPrefix
- KermitLongBlocks
- KermitMaxLen
- KermitMaxWindows
- KermitPadCharacter
- KermitPadCount
- KermitRepeatPrefix
- KermitSWCTurnDelay
- KermitTerminator
- KermitTimeoutSecs
- KermitWindowsTotal
- KermitWindowsUsed
- Protocol
- ProtocolStatus
- ProtocolStatusDisplay
- ReceiveDirectory
- ReceiveFileName
- RTSLowForWrite
- SendFileName
- StatusInterval
- TotalErrors
- TransmitTimeout
- UpcaseFileNames
- WriteFailAction
- XYmodemBlockWait
- Zmodem8K
- ZmodemFileOptions
- ZmodemFinishRetry
- ZmodemOptionOverride
- ZmodemRecover
- ZmodemSkipNoFile

Methods

- CancelProtocol
- EstimateTransferSecs
- StartReceive
- StartTransmit

Events

- OnProtocolAccept
- OnProtocolFinish
- OnProtocolLog
- OnProtocolStatus

Reference Section

AbortNoCarrier property

Description

Determines whether the protocol is canceled automatically when the DCD modem signal drops. Read/write.

Data type

`Boolean`

Syntax

`expression``.`**`AbortNoCarrier`**`[= `*`value`*`]`

`expression`` must reference an `**`APAXPort.`**

Remarks

Default: False

Using the AbortNoCarrier property is better than checking DCD and calling CancelProtocol in your own code. When you do this, the protocol engine sends a cancel sequence to the remote computer. If hardware flow control is enabled and the modem has lowered the DSR or CTS signals as well as DCD, the protocol waits several seconds before deciding it can't send the cancel command, leading to an unnecessary delay for the application. The AbortNoCarrier property prevents the protocol engine from sending the cancel sequence, so the protocol stops immediately.

See also

CancelProtocol

AsciiCharDelay property

Description

Determines the number of milliseconds to delay between characters during an ASCII file transfer. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`AsciiCharDelay`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Default: 0

The default delay of zero should be retained whenever possible to maximize performance. However, if ASCII data is being fed directly into an application such as a text editor, it might be necessary to insert delays to allow the application time to process the data.

Example

The following example sets the inter-character delay to 2 milliseconds and the inter-line delay to 50 milliseconds:

```
MyPort.AsciiCharDelay = 2
MyPort.AsciiLineDelay = 50
```

See also

AsciiLineDelay

AsciiCRTranslation property

Description

Determines the end-of-line translation mode for carriage returns. Read/write.

Data type

`TAsciiEOLTranslation`

Syntax

`expression``.`**`AsciiCRTranslation`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Settings

Valid settings for AsciiCRTranslation are:

Constant	Description
aetNone	The default, do not modify the character
aetStrip	Strip the character from the data stream
aetAddLFAfter	Add a <LF> after each <CR>

9

Remarks

Default: aetNone

Setting AsciiCRTranslation to aetAddCRBefore does not apply to AsciiCRTranslation, so it is treated as aetNone.

Example

The following example causes all <LF> characters to be stripped while <CR> characters are transmitted:

```
MyPort.Protocol = ptAscii
MyPort.AsciiCRTranslation = aetNone
MyPort.AsciiLFTranslation = aetStrip
MyPort.StartTransmit()
```

See also

AsciiEOLChar, AsciiLFTranslation

AsciiEOFTimeout property

Description

Determines the number of ticks before an ASCII transfer is automatically terminated. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`AsciiEOFTimeout`**`[=` *`value`*`]`

`expression`` must reference an `**`APAXPort.`**

Remarks

Default: 364 (20 seconds)

Because most text files are terminated by a ^Z character (ASCII 26), the ASCII protocol closes the file and ends the protocol when it finds a ^Z. If the received file isn't terminated by a ^Z, the ASCII protocol determines the file was completely received after a specified number of ticks elapse without receiving any new data. The default of 20 seconds can be changed by assigning a new tick value to this property.

AsciiEOLChar property

Description

Determines the character that triggers an inter-line delay. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`AsciiEOLChar`**`[= `*`value`*`]`

`expression`` must reference an `**`APAXPort.`**

Remarks

Default: 13 (<CR> or ^M)

After an ASCII file transmit sends the character specified by this property, it pauses for the number of milliseconds specified by the AsciiLineDelay property.

Note that this character is not involved in on-the-fly translation of end-of-line characters read from or written to an ASCII file; that translation is controlled by the AsciiCRTranslation and AsciiLFTranslation properties.

The default end-of-line character is <CR> or ^M. If you are transmitting Unix files, which use <LF> or ^J for the end-of-line marker, you should set AsciiEOLChar to 10 (<LF> or ^J).

See also

AsciiLineDelay

AsciiLFTranslation property

Description

Determines the end-of-line translation mode for line feeds. Read/write.

Data type

`TAsciiEOLTranslation`

Syntax

`expression``.`**`AsciiLFTranslation`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Settings

Valid settings for AsciiLFTranslation are:

Constant	Description
aetNone	The default, do not modify the character
aetStrip	Strip the character from the data stream
aetAddCRBefore	Insert a <CR> before each <LF>
aetAddLFAfter	Not applicable to AsciiLFTranslation

Remarks

Default: aetNone

Setting AsciiLFTranslation to aetAddLFAfter does not apply to AsciiLFTranslation, so it is treated as aetNone.

See also

AsciiCRTranslation, AsciiEOLChar

AsciiLineDelay property

Description

Determines the number of milliseconds to delay between lines during an ASCII file transfer. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`AsciiLineDelay`**`[=` *`value`*`]`

`expression`` must reference an `**`APAXPort.`**

Remarks

Default: 0

The default delay of zero should be retained whenever possible to maximize performance. However, if ASCII data is being fed directly into an application such as a text editor, it might be necessary to insert delays to allow the application time to process the data.

See also

AsciiCharDelay

AsciiSuppressCtrlZ property

Description

Determines whether an ASCII protocol stops transmitting when it encounters the first ^Z in the file. Read/write.

Data type

`Boolean`

Syntax

`expression``.`**`AsciiSuppressCtrlZ`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Default: False

If this property is False, the ASCII protocol transmits all characters in the file, including ^Z characters. If it is True, it stops before transmitting the first ^Z that it encounters. Generally you should leave this property set to False because the receiver might use ^Z as an end-of-protocol indicator, as APAX does.

Batch property

Description

Determines whether the current protocol supports batch transfers. Read-only.

Data type

`Boolean`

Syntax

`expression``.`**`Batch`**

`expression` `must reference an` **`APAXPort.`**

Remarks

Batch transfers allow sending more than one file in the same protocol session.

This property is most useful within an OnProtocolStatus event handler.

BlockCheckMethod property

Description

Determines the error checking method used by the protocol. Read/write.

Data type

`TBlockCheckMethod`

Syntax

`expression``.`**`BlockCheckMethod`**`[= `*`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Settings

Valid settings for BlockCheckMethod are:

Constant	Description
bcmNone	No error checking
bcmChecksum	Single byte checksum
bcmChecksum2	Two byte checksum used by Kermit
bcmCRC16	16-bit CRC
bcmCRC32	32-bit CRC
bcmCRCK	Three byte CRC used by Kermit

Remarks

The default error checking method depends on the protocol. The Xmodem1K, Xmodem1KG, Ymodem, YmodemG, and ASCII protocols provide either no error checking or a single error checking mode, so they ignore assignments to BlockCheckMethod. Assigning bcmCrc16 to BlockCheckMethod converts an Xmodem protocol into an XmodemCrc protocol. Conversely, assigning bcmCheckSum to BlockCheckMethod converts an XmodemCrc protocol to an Xmodem protocol.

The Zmodem protocol accepts only the bcmCrc16 and bcmCrc32 types. The Kermit protocol accepts only the bcmChecksum, bcmCheckSum2, and bcmCrcK types.

No error is generated if an unaccepted type is assigned, but the assignment is ignored. You should be sure to set the desired protocol before setting a non-default BlockCheckMethod.

See also

Protocol

BlockErrors property

Description

The number of errors that have occurred while transferring the current block. Read-only.

Data type

`Integer`

Syntax

`expression``.`**`BlockErrors`**

`expression` `must reference an` **`APAXPort.`**

Remarks

This property is most useful within an OnProtocolStatus event handler.

See also

TotalErrors

BlockLength property

Description

The number of bytes currently being transferred per block. Read-only.

`Integer`

Syntax

`expression``.`**`BlockLength`**

`expression` `must reference an` **`APAXPort.`**

Remarks

For some protocols this value remains fixed (e.g., Xmodem always uses 128 byte blocks); for others it can vary during the transfer process (e.g., Zmodem can vary between 8192 bytes and 256 bytes depending on options and line conditions).

This property is most useful within an OnProtocolStatus event handler.

BlockNumber property

Description

The number of blocks transferred so far. Read-only.

Data type

`Integer`.

Syntax

`expression``.`**`BlockNumber`**

`expression` `must reference an` **`APAXPort.`**

Remarks

This is obtained by dividing the number of bytes transferred by the current block length, so it will change if the block length changes.

This property is most useful within an OnProtocolStatus event handler.

BytesRemaining property

Description

The number of bytes still to be transferred in the current file. Read-only.

Data type

`Integer`

Syntax

`expression``.`**`BytesRemaining`**

`expression` `must reference an` **`APAXPort.`**

Remarks

This is computed as the FileLength minus the value of BytesTransferred. When the file size isn't known, BytesRemaining returns zero.

This property is most useful within an OnProtocolStatus event handler.

See also

BytesTransferred, FileLength

BytesTransferred property

Description

The number of bytes transferred so far in the current file. Read-only.

Data type

`Integer`

Syntax

`expression``.`**`BytesTransferred`**

`expression` `must reference an` **`APAXPort.`**

Remarks

When transmitting, this number is sometimes only an estimate. The uncertainty comes from the fact that the protocol window doesn't know when a particular byte has actually been transferred. BytesTransferred is the number of bytes the protocol window has transferred to the output buffer of the communications driver, minus the number of bytes that the driver reports are currently in the buffer.

Unfortunately, this calculation is still imperfect because it's impossible to know how much of the output buffer holds actual file data and how much holds overhead characters needed by the protocol. Each protocol has a few simple rules it uses to estimate this proportion, which in practice yield good estimates.

This property is most useful within an OnProtocolStatus event handler.

See also

BytesRemaining

CancelProtocol method

Description

Cancels the protocol currently in progress.

Syntax

expression.**CancelProtocol()**

expression must reference an **APAXPort.**

Remarks

CancelProtocol cancels the protocol regardless of its current state. If appropriate, a cancel string is sent to the remote computer. The protocol generates an OnProtocolFinish event with the error code ecCancelRequested, then cleans up and terminates.

Example

The following example shows how to cancel a protocol from within a protocol status dialog:

```
MyPort.CancelProtocol
```

See also

InProgress

ElapsedTicks property

Description

The time elapsed since the protocol started. Read-only.

Data type

Integer

Syntax

expression.**ElapsedTicks**

expression must reference an **APAXPort.**

Remarks

In order to provide accurate character per second transfer rates, the protocol engine doesn't start the timer until it receives the first block from the remote computer, or until it sends the first data block. ElapsedTicks is measured in ticks, which occur at roughly 18.2 per second.

This property is most useful within an OnProtocolStatus event handler.

EstimateTransferSecs method

Description

Returns the amount of time to transfer a file. Returns an integer that corresponds to the transfer duration estimation in seconds.

Syntax

expression.**EstimateTransferSecs(*Size*)**

Part	Description	Data Type
expression	An expression that returns an **APAXPort** object	**APAXPort**
Size	Size of file to be transferred (in bytes)	**Integer**

Remarks

You can call EstimateTransferSecs in a status event handler to obtain the approximate number of seconds required to transfer Size bytes of data. Typically, a status routine calls it in two places. In the first place, which should generally be executed only one time when the status routine is first called, it passes the total size of the file to get the total transfer time. In the second place, which should be executed every time the status routine is called, it passes the number of bytes remaining to get the transfer time remaining.

EstimateTransferSecs automatically accounts for the baud rate of the port's connection and various internal details of the active protocol. The estimated transfer time is also affected by two approximate overhead factors that are specific to the type of protocol. See the Overhead and KermitSWCTurnDelay properties for more information about these factors.

To compute the transfer time, EstimateTransferSecs first computes an effective transfer rate using the following formulas:

```
ActualCPS     = ActualBPS/10

Efficiency    = ratio of data bytes to highest possible number of
                bytes, calculated as follows:

                 BlockLength
 ---------------------------------------------

BlockLength + Overhead + ((TurnDelay * ActualCPS)
                  div 1000)

EffectiveCPS = ActualCPS * Efficiency
```

Then the estimated transfer time is Size divided by EffectiveCPS.

Example

The following example calls EstimateTransferSecs in a status routine to get the total and remaining transfer times:

```
Private Sub AxTerminal1_OnProtocolStatus(ByVal Options As Long)
Dim TotalTime as Integer
Dim RemainingTime as Integer
  TotalTime = MyPort.EstimateTransferSecs(FileLength)
  RemainingTime = MyPort.EstimateTransferSecs(BytesRemaining)
End Sub
```

See also

OnProtocolStatus, KermitSWCTurnDelay

FileDate property

Description

Returns the date and time of the file being transferred. Read-only.

9

Data type

`Integer`

Syntax

`expression``.`**`FileDate`**

`expression`` must reference an `**`APAXPort.`**

Remarks

For transmitted files the file timestamp is always known. For received files the timestamp is known only if the protocol supports this feature and the receiver has received this information. FileDate is accurate after FileName returns a non-empty string.

If the timestamp is not known, FileDate returns zero.

This property is most useful within an OnProtocolStatus event handler.

See also

FileLength, ReceiveFileName

FileLength property

Description

Returns the size of the file being transferred. Read-only.

Data type

`Integer`

Syntax

`expression``.`**`FileLength`**

`expression`` must reference an `**`APAXPort.`**

Remarks

For transmitted files the file size is always known. For received files the file size is known only if the protocol supports this feature and the receiver has received this information. FileLength is known after FileName returns a non-empty string. If the file size is not known, FileLength returns zero.

This property is most useful within an OnProtocolStatus event handler.

See also

FileDate

FinishWait property

Description

Determines how long the receiver waits for an end-of-transmission signal before timing out. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`FinishWait`**`[= `*`value`*`]`

`expression`` must reference an `**`APAXPort.`**

Remarks

Default: 364 (20 seconds)

This property applies only to Xmodem, Ymodem, and Zmodem protocols.

At the end of an Xmodem or Ymodem file transfer the transmitter sends an <EOT> to the receiver to signal the end of the file and then waits FinishWait ticks (20 seconds by default) for a response. Normally this provides ample time. However, when Xmodem1KG and YmodemG are used over links with long propagation times or slow receivers, the default value might not be enough. Use FinishWait to extend the amount of time that the transmitter waits before timing out and reporting an error. Note that FinishWait is specified in ticks.

Similarly, in a Zmodem transfer the transmitter sends a ZFin packet to the receiver to signal the end of the file and then waits FinishWait ticks to receive an acknowledgment before timing out.

See also

ZmodemFinishRetry

HandshakeRetry property

Description

Determines the retry count for protocol handshaking. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`HandshakeRetry`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Default: 10

This property controls how many times each protocol attempts to detect the initial handshake from its remote partner. HandshakeRetry applies to all protocols but ASCII, which does not perform handshaking.

See also

HandshakeWait

HandshakeWait property

Description

Determines the wait between retries for protocol handshaking. Read/write.

Data type

`Integer`

Syntax

`expression.HandshakeWait[= value]`

`expression must reference an APAXPort.`

Remarks

Default: 182

This property is the number of ticks a protocol waits when a handshake attempt fails before it tries again. HandshakeWait applies to all protocols but ASCII, which does not perform handshaking.

See also

HandshakeRetry

HonorDirectory property

Description

Determines whether a protocol honors the directory name of a file being received. Read/write.

Data type

`Boolean`

Syntax

`expression.HonorDirectory[= value]`

`expression must reference an APAXPort.`

Remarks

Default: False

If HonorDirectory is set to True, a received file is stored in the directory specified by the transmitter, unless that directory does not already exist, in which case it is stored in the current directory or the ReceiveDirectory. If HonorDirectory is set to False, the transmitter's directory is ignored.

See also

IncludeDirectory

IncludeDirectory property

Description

Determines whether the complete path name is transmitted. Read/write.

Data type

`Boolean`

Syntax

`expression``.`**`IncludeDirectory`**`[= `*`value`*`]`

`expression`` must reference an `**`APAXPort.`**

Remarks

Default: False

If IncludeDirectory is set to True, the protocol sends the drive and directory along with the file name of each file it transmits. The receiver might use or ignore this information. If IncludeDirectory is False, only the file name is transmitted, even if the file is not found in the current directory.

See also

HonorDirectory

InitialPosition property

Description

The initial file offset for a resumed transfer. Read-only.

Data type

`Integer`

Syntax

`expression``.`**`InitialPosition`**

`expression` `must reference an` **`APAXPort.`**

Remarks

This property applies only to the Zmodem protocol, which support resumed file transfers. For a transfer from scratch, InitialPosition returns zero. For a resumed transfer, InitialPosition returns the offset where the transfer was resumed. This offset should be subtracted from BytesTransferred to obtain the actual number of bytes transferred during the resumed session.

This property is most useful within an OnProtocolStatus event handler.

Example

The following example shows how to compute the character per second transfer rate in a protocol status routine. The constant values are used to convert ticks to seconds. Note that the same expression is valid whether or not the transfer has been resumed:

```
CPS =
 (91*(MyPort.BytesTransferred-MyPort.InitialPosition))
  div (5 * MyPort.ElapsedTicks)
```

See also

BytesTransferred

InProgress property

Description

Returns True if a file transfer is currently in progress. Read-only.

Data type

`Boolean`

Syntax

`expression``.`**`InProgress`**

`expression` `must reference an` **`APAXPort.`**

Remarks

This property is important because APAX protocols run in the background. A call to StartTransmit or StartReceive returns immediately to your code. You can use either a polling loop that checks InProgress or an OnProtocolFinish event handler to detect when the protocol has finished.

See also

OnProtocolFinish

KermitCtlPrefix property

Description

Determines the character Kermit uses to quote control characters. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`KermitCtlPrefix`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Default: 35 ('#')

Character quoting is similar to escaping in Zmodem. The character is replaced by a quote character and a modified form of the original character. The quote character tells the receiver how to convert the modified character back to its original value. Quoting ensures that certain binary characters are never put into the data stream where they could be misinterpreted by a modem or another part of the serial link.

Although Zmodem transforms only a few critical characters such as <XON> and <XOFF>, Kermit quotes nearly all characters. This is one of the features that permits Kermit to run in nearly any environment. When quoting is finished, a Kermit data packet consists almost entirely of printable ASCII characters. The only exceptions are an <SOH> character at the start of each packet and a <CR> at the end.

Kermit quotes control characters by replacing them with a quote character and a modified version of the control character. For example, ^A becomes '#A' where '#' is the quote character. The process of converting ^A to 'A' is called "Ctl" and it works like this:

```
Ctl(x) = x xor 40h
```

This operation is its own inverse, that is, Ctl(Ctl(x)) = x.

Kermit also quotes characters with their eighth bit set, which allows it to transmit 8 bit data over 7 bit data links. The quote character in this case is '&' and the quoted data character is obtained simply by stripping the high bit. For example, the quoted version of character $C1 ('A' with its high bit set) is '&A'.

Binary numbers in Kermit packet headers and in repeated character strings are also transformed to assure that they are printable characters. This is achieved by adding 32 to each number before it is transmitted and subtracting 32 after it is received. In Kermit parlance, these operations are known as "ToChar" and "UnChar."

Kermit has a simple built-in data compression mechanism called run length encoding. When it sees a long string of repeated characters, it compresses the string into a quote character, a length byte, and the repeated character. Obviously, there must be at least 4 repeated characters before there is any compression. The quote character for run length encoding is '~'. Hence, the string 'AAAAA' becomes '~%A', where '%' is equivalent to a binary 5 after the "ToChar" operation.

See also

KermitHighbitPrefix, KermitRepeatPrefix

KermitHighbitPrefix property

Description

Determines the technique Kermit uses to quote characters that have their eighth bit set. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`KermitHighBitPrefix`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Default: 89 (‘Y’)

The value specified by this property is not always transmitted literally as a quote character. If it equals ‘Y’, the default, it means that the protocol won’t use high bit quoting unless the remote requires it, in which case it uses the prefix character requested by the remote.

If KermitHighbitPrefix equals 38 (‘&’) or is in the ASCII range 33-62 or 96-126, it indicates that the protocol requires high bit quoting and that its value is the character used for the prefix.

If KermitHighbitPrefix equals 78 (‘N’) or any other value not listed here, the protocol won’t use high bit quoting at all, even if the remote requests it.

See also

KermitCtlPrefix, KermitRepeatPrefix

KermitLongBlocks property

Description

Returns True if Kermit long packets are in use. Read-only.

Data type

`Boolean`

Syntax

`*expression*.**KermitLongBlocks**`

`*expression* must reference an **APAXPort.**`

See also

KermitMaxLen

KermitMaxLen property

Description

Determines the maximum number of bytes in one Kermit packet. Read/write.

Data type

`Integer`

Syntax

`*expression*.**KermitMaxLen**[= *value*]`

`*expression* must reference an **APAXPort.**`

Remarks

Default: 80

The normal maximum value is 94, but the default value of 80 is suggested by the Kermit Protocol Manual. If KermitMaxLen is set to a value in the range of 95 to 1024, long packets are enabled with the specified packet size. As with other Kermit settings, however, long packets will be used only if the remote partner also supports it.

See also

KermitMaxWindows

KermitMaxWindows property

Description

Determines whether Kermit sliding windows control is enabled. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`KermitMaxWindows`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Default: 0

If KermitMaxWindows is set to a value between 1 and 27, sliding windows are enabled with the specified window count. This allows a Kermit transmitter to send additional packets without waiting for an acknowledgment from the receiver, thus improving throughput. As with other Kermit settings, however, sliding windows control will be used only if the remote partner also supports it.

See also

KermitWindowsTotal, KermitWindowsUsed

KermitPadCharacter property

Description

Determines the character that Kermit uses to pad the beginning of each packet. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`KermitPadCharacter`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Default: 32 (<Space>)

See also

KermitTerminator

KermitPadCount property

Description

Determines the number of pad characters that Kermit transmits at the beginning of each packet. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`KermitPadCount`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Default: 0

See also

KermitPadCharacter

KermitRepeatPrefix property

Description

Determines the prefix that Kermit uses when compressing strings of repeated characters. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`KermitRepeatPrefix`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Default: 126 (‘~’)

When Kermit sees four or more equal and adjacent characters, it compresses the sequence into a quote character (KermitRepeatPrefix), a length byte, and the repeated character. The default quote character rarely needs to be changed.

See also

KermitCtlPrefix, KermitHighbitPrefix

KermitSWCTurnDelay property

Description

Determines the turnaround delay used by EstimateTransferSecs when a Kermit sliding windows protocol is in use. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`KermitSWCTurnDelay`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Default: 0

This property is the time in milliseconds for a data block to transit from the sender to the receiver, for the receiver to send an acknowledgment, and for the acknowledgment to arrive back at the sender. It is used by the EstimateTransferSecs method to estimate the time to transfer a given amount of data.

When Kermit sliding windows control is enabled, the transmitter does not generally wait for acknowledgment of a packet before sending the next one. Hence, an appropriate default is zero milliseconds.

EstimateTransferSecs uses the value of the TurnDelay property for Kermit transfers when sliding windows control is not enabled, and the KermitSWCTurnDelay property when it is enabled.

KermitTerminator property

Description

Determines the character used to terminate a Kermit data packet. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`KermitTerminator`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Default: 13 (<CR> or ^M)

This character is used only by Kermit hosts that cannot start processing a data line until a terminating character is received.

See also

KermitPadCharacter

KermitTimeoutSecs property

Description

Determines how long Kermit waits for the next expected byte. Read/write.

Data type

`Integer`

Syntax

expression.**KermitTimeoutSecs** [= *value*]

expression must reference an **APAXPort.**

Remarks

Default: 5

If a Kermit transmitter waits more than KermitTimeoutSecs for an acknowledgment, it resends the last packet. If a Kermit receiver waits more than KermitTimeoutSecs for the next byte, it sends an error packet to the transmitter.

See also

TransmitTimeout

KermitWindowsTotal property

Description

Returns the total number of Kermit sliding windows negotiated for the current transfer. Read-only.

Data type

`Integer`

Syntax

expression.**KermitWindowsTotal**

expression must reference an **APAXPort.**

Remarks

If sliding windows control is disabled, KermitWindowsTotal returns 0.

See also

KermitMaxWindows, KermitWindowsUsed

KermitWindowsUsed property

Description

Returns the number of Kermit sliding windows that currently contain data. Read-only.

Data type

`Integer`

Syntax

```
expression.KermitWindowsUsed

expression must reference an APAXPort.
```

Remarks

If sliding windows control is disabled, KermitWindowsUsed returns 0.

See also

KermitMaxWindows, KermitWindowsTotal

OnProtocolAccept event

Description

Defines an event handler that is called as soon as the name of an incoming file is known.

Syntax

```
Private Sub expression_OnProtocolAccept(
  Accept As Boolean, FName As String)
```

Part	Description	Data Type
expression	References the **APAXPort** object that fired the event	**APAXPort**
Accept	Allows for acceptance or rejection of the file transfer	**Boolean**
FName	Defines the name of the file to be received	**String**

Remarks

This event handler provides an opportunity for the receiver to reject or rename the incoming file. If an OnProtocolAccept handler is not installed, all files are accepted (subject to the setting of the WriteFailAction property).

The event handler should set Accept to True to accept the file, False to reject it. FName is the name of the file to be received. The event handler can change the name if, for example, it would overwrite an existing file.

See also

OnProtocolFinish, OnProtocolLog, OnProtocolStatus, WriteFailAction

OnProtocolFinish event

Description

Defines an event handler that is called when a protocol transfer ends.

Syntax

```
Private Sub Expression_OnProtocolFinish(ByVal ErrorCode As Long)
```

Part	Description	Data Type
Expression	References the **APAXPort** object that fired the event	**APAXPort**
ErrorCode	Indicates success or failure of the protocol	**Long**

Remarks

This event is generated whether the protocol ends successfully or not. If it ends successfully, ErrorCode is zero. Otherwise, ErrorCode is a number indicating the type of error.

An application could use this handler to display a completion dialog box (needed only if a protocol status event handler is not also in use) or to enable the scheduling of another file transfer.

See also

InProgress, OnProtocolAccept, OnProtocolLog, OnProtocolStatus

OnProtocolLog event

Description

Defines an event handler that is called at well defined points during a protocol transfer.

Syntax

Private Sub *Expression*_**OnProtocolLog(ByVal** *Log* **As Long)**

Part	Description	Data Type
Expression	References the **APAXPort** object that fired the event	**APAXPort**
Log	Indicates the current state of the protocol	**Long**

Settings

The possible values for Log are:

Constant	Description
lfReceiveStart	File receive is starting
lfReceiveOK	File was received successfully
lfReceiveFail	File receive failed
lfReceiveSkip	File was skipped (rejected by receiver)
lfTransmitStart	File transmit is starting
lfTransmitOK	File was transmitted successfully
lfTransmitFail	File transmit failed
lfTransmitSkip	File not found on the sender

Remarks

The primary purpose of this event is to give applications a chance to log statistical information about file transfers such as the transfer time and whether they succeeded or failed. Applications can also use this event for start-up and cleanup activities such as deleting partial files after unsuccessful downloads.

No other information is passed along with the event. Use protocol status properties such as SendFileName and ElapsedTicks to get additional information about the state of the transfer.

OnProtocolStatus event

Description

Defines an event handler that is called regularly during a file transfer.

Syntax

`Private Sub` *`Expression`*`_`**`OnProtocolStatus(ByVal`** *`Options`* **`As Long)`**

Part	**Description**	**Data Type**
Expression	References the **APAXPort** object that fired the event	**APAXPort**
Options	Indicates the current state of the protocol	**Long**

Remarks

This event is generated for each block transmitted or received, after the completion of each major operation (e.g., renaming a file, detecting an error, ending the transfer), and at intervals of StatusInterval ticks (by default 18 ticks, or about 1 second). The program can use it to update a status display that informs the user about the protocol progress.

A number of the properties of this component can be read to establish the status of the transfer. Options is set to apFirstCall (1) on the first call to the handler, apLastCall (2) on the last call to the handler, and zero on all other calls.

A predefined status window is supplied with APAX. For a standard protocol status window you can simply set the ProtocolStatusDisplay to True. If you do so, there is no need to supply your own OnProtocolStatus event handler.

See also

ProtocolStatus, ProtocolStatusDisplay, StatusInterval

Protocol property

Description

Determines the type of file transfer protocol. Read/write.

Data type

`TProtocolType`

Syntax

`expression``.`**`Protocol`**`[= `*`value`*`]`

`expression`` must reference an `**`APAXPort.`**

Settings

Valid settings for Protocol are:

Constant	Description
`ptNoProtocol`	`No protocol`
`ptXmodem`	`Xmodem protocol`
`ptXmodemCRC`	`XmodemCRC protocol`
`ptXmodem1K`	`Xmodem1K protocol`
`ptXmodem1KG`	`Xmodem1KG protocol`
`ptYmodem`	`Ymodem protocol`
`ptYmodemG`	`YmodemG protocol`
`ptZmodem`	`Zmodem protocol`
`ptKermit`	`Kermit protocol`
`ptAscii`	`Ascii protocol`

Remarks

Default: ptZmodem

APAX encapsulates all of the file transfer protocols that it supports into a single component. To select a particular type of protocol, you must assign the desired type to the Protocol property. You should generally assign to Protocol before assigning other properties, since various defaults are assigned whenever you change Protocol, and some properties are valid only when Protocol has a particular value.

Assigning a new value to Protocol first deallocates any protocol-specific memory used by the prior protocol, then allocates and initializes any structures required by the current protocol.

You should generally not assign ptNoProtocol to Protocol, but it can be used to de-allocate previous protocol memory while temporarily not allocating new protocol memory.

See also

BlockCheckMethod

ProtocolStatus property

Description

Returns a code that indicates the current state of the protocol. Read-only.

Data type

`Integer`

Syntax

`expression``.`**`ProtocolStatus`**

`expression`` must reference an `**`APAXPort.`**

Remarks

This property is most useful within an OnProtocolStatus event handler.

ProtocolStatus returns a code that indicates the current state of the protocol. The following table shows all of the possible values. The usual status value is psOK, which means that the protocol is operating normally. Other status values indicate recoverable error conditions, protocol resume conditions, protocol start-up states, and internal protocol states. Fatal protocol errors are not represented by protocol states. However, it is possible that a final status message might be sent after a fatal error occurs.

Settings

Status Code	Value	Explanation
psOK	0	Protocol is ok
psProtocolHandshake	1	Protocol handshaking in progress
psInvalidDate	2	Bad date/time stamp received and ignored
psFileRejected	3	Incoming file was rejected
psFileRenamed	4	Incoming file was renamed
psSkipFile	5	Incoming file was skipped
psFileDoesntExist	6	Incoming file doesn't exist locally, skipped
psCantWriteFile	7	Incoming file skipped due to Zmodem options
psTimeout	8	Timed out waiting for something
psBlockCheckError	9	Bad checksum or CRC
psLongPacket	10	Block too long
psDuplicateBlock	11	Duplicate block received and ignored
psProtocolError	12	Error in protocol
psCancelRequested	13	Cancel requested
psEndFile	14	At end of file
psSequenceError	16	Block was out of sequence
psAbortNoCarrier	17	Aborting on carrier loss
psGotCrcE	18	Received Zmodem CrcE packet
psGotCrcG	19	Received Zmodem CrcG packet
psGotCrcW	20	Received Zmodem CrcW packet
psGotCrcQ	21	Received Zmodem CrcQ packet

See also

TotalErrors

ProtocolStatusDisplay property

Description

An instance of a protocol status window. Read/write.

Data type

`Boolean`

Syntax

`expression.`**`ProtocolStatusDisplay`**`[= value]`

`expression` `must reference an` **`APAXPort.`**

Remarks

If ProtocolStatusDisplay is False, as it is by default, the protocol does not provide an automatic status window. You can install an OnProtocolStatus event handler to display a custom status dialog in this case.

If you set this property to True, the status window will be displayed and updated automatically.

ReceiveDirectory property

Description

Determines the directory where received files are stored. Read/write.

Data type

`String`

Syntax

`expression.`**`ReceiveDirectory`**`[= value]`

`expression` `must reference an` **`APAXPort.`**

Remarks

If the value specifies only a drive (e.g., 'D:'), files are stored in the current directory on that drive. If the property is set to an empty string, as it is by default, received files are stored in the current directory.

See also

ReceiveFileName

ReceiveFileName property

Description

Determines the name of the file currently being received. Read/write.

Data type

`String`

Syntax

`*expression*.**ReceiveFileName**[= *value*]`

`*expression* must reference an **APAXPort.**`

Remarks

This should be considered a read-only property for all protocols except Xmodem and ASCII, which do not transfer a filename along with the file data. For these two protocols you must assign a value to FileName before calling StartReceive. For the remaining protocols supported by APAX, you can read the value of SendFileName within a protocol status routine to obtain the file name transferred by the protocol.

If FileName does not include drive or path information, the incoming file is stored in the current directory or the directory specified by ReceiveDirectory. If FileName includes drive and/or path information and HonorDirectory is True, the incoming file is stored in that directory regardless of whether a value was assigned to ReceiveDirectory.

Example

The following example stores a file received via Xmodem to C:\DOWNLOAD\RECEIVE.TMP:

```
  MyPort.Protocol = ptXmodem
  MyPort.FileName = "C:\DOWNLOAD\RECEIVE.TMP"
  MyPort.StartReceive()
```

See also

ReceiveDirectory, HonorDirectory

RTSLowForWrite property

Description

Determines whether protocols force RTS low while writing received data to disk. Read/write.

Data type

`Boolean`

Syntax

`expression``.`**`RTSLowForWrite`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Default: False

When RTSLowForWrite is set to True, hardware flow control is used to prevent the transmitter from sending additional data while the receiver writes data to disk. As soon as the disk write is finished, RTS is raised again. This feature might be required if other Windows applications are being run at the same time as a protocol transfer or if the disk driver leaves interrupts disabled for an excessive time.

In order for this option to be effective, disk write caching must be disabled.

If the protocol is transferring files using a modem, it might also be necessary to configure the modem to react correctly to the RTS signal.

SendFileName property

Description

Determines the file mask to use when transmitting files. Read/write.

Data type

`String`

Syntax

`expression``.`**`SendFileName`**`[= `*`value`*`]`

`expression`` must reference an `**`APAXPort.`**

Remarks

SendFileName can specify a single file or can contain DOS wildcards to transmit multiple files using a batch protocol such as Zmodem. If it does not specify a drive and directory, files are read from the current directory.

Only a single mask can be used for each transfer.

Example

The following example transmits all files with a ZIP extension in the C:\UPLOAD directory:

```
MyPort.FileMask = "C:\UPLOAD\*.ZIP"
MyPort.StartTransmit()
```

StartReceive method

Description

Tells the protocol to start receiving files.

Syntax

```
expression.StartReceive()
```

```
expression must reference an APAXPort.
```

Remarks

The steps leading up to calling StartReceive look something like this:

1. Set the Protocol property.
2. Set other properties to customize the protocol.
3. Write suitable handlers for protocol events.
4. Call the StartReceive method.

StartReceive returns immediately and receives files in the background, occasionally generating events to keep the application apprised of progress. When the protocol is finished, either successfully or with a fatal error, it generates an OnProtocolFinish event and its InProgress property is set to False.

See also

Protocol, StartTransmit

StartTransmit method

Description

Tells the protocol to start transmitting files.

Syntax

`expression.StartTransmit()`

`expression must reference an APAXPort.`

Remarks

The steps leading up to calling StartTransmit look something like this:

1. Set the Protocol property.
2. Set other properties to customize the protocol.
3. Write suitable handlers for protocol events.
4. Set the FileMask property to return a list of files to transmit.
5. Call the StartTransmit method.

StartTransmit returns immediately and transmits files in the background, occasionally generating events to keep the application apprised of progress. When the protocol is finished, either successfully or with a fatal error, it generates an OnProtocolFinish event and its InProgress property is set to False.

See also

FileMask, Protocol, StartReceive

StatusInterval property

Description

The maximum number of clock ticks between OnProtocolStatus events. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`StatusInterval`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Default: 18

The OnProtocolStatus event is generated for each block transmitted or received, after the completion of each major operation (e.g., renaming a file, detecting an error, ending the transfer), and at intervals of StatusInterval ticks.

This property also determines how frequently the status display window is updated.

See also

OnProtocolStatus, ProtocolStatusDisplay

TotalErrors property

Description

The number of errors encountered since the current file transfer was started. Read-only.

Data type

`Integer`

Syntax

`expression``.`**`TotalErrors`**

`expression` must reference an **`APAXPort.`**

Remarks

This error count is reset whenever a new file is started. This property is most useful within an OnProtocolStatus event handler. See "Protocol Status" on page 388 for more information.

TransmitTimeout property

Description

Determines the maximum time a sender will wait for the receiver to release flow control. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`TransmitTimeout`**`[= `*`value`*`]`

`expression`` must reference an `**`APAXPort.`**

Remarks

Default: 1092

If the receiver blocks flow control for longer than TransmitTimeout ticks (60 seconds by default), the protocol is aborted.

9

UpcaseFileNames property

Description

Determines whether the protocol converts file names to upper case. Read/write.

Data type

`Boolean`

Syntax

`expression``.`**`UpcaseFileNames`**`[= `*`value`*`]`

`expression`` must reference an `**`APAXPort.`**

Remarks

Default: True

Applications provide file names to protocols by setting the SendFileName property. File names can also be received as part of the protocol transfer. Because the DOS/16-bit Windows file system stores all file names in upper case, the protocol component converts file and path names to uppercase.

Windows preserves the specified case in file names, although they don't normally use case to distinguish between file names. For example, the file name "MixCase.Txt" is stored by the file system with the upper and lower case characters preserved, however, it can be accessed by any combination of upper and lower case (e.g., "MIXCASE.TXT" or "mIXCAse.tXt"). If you want to display the preserved case in status and log routines, set UpcaseFileNames to False.

WriteFailAction property

Description

Determines the receiver's behavior when the destination file already exists. Read/write.

Data type

`TWriteFailAction`

Syntax

`expression``.`**`WriteFailAction`**`[= `*`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Settings

The valid settings for WriteFailAction are:

Constant	Description
`wfWriteFail`	`Fail the receive attempt`
`wfWriteRename`	`Rename the incoming file`
`wfWriteAnyway`	`Overwrite the existing file`

Remarks

Default: wfWriteRename

When wfWriteRename is selected and the destination file already exists, the first character in the incoming file name is replaced with '$' (e.g., 'SAMPLE.DOC' becomes '$AMPLE.DOC'). If that renamed file already exists, it is overwritten without warning.

The logic that handles these overwrite options is executed after the OnProtocolAccept event has been generated. If you write an event handler that deals with possible overwrites, be sure to set WriteFailAction to wfWriteAnyway before starting a transfer.

XYmodemBlockWait property

Description

Determines the number of ticks Xmodem and Ymodem wait between blocks for a response from the remote machine. Read/write.

Data type

`Integer`

Syntax

expression.**XYmodemBlockWait**[= *value*]

expression must reference an **APAXPort.**

Remarks

Default: 91

If the wait exceeds XYmodemBlockWait ticks, a sending protocol retransmits the block and a receiving protocol aborts the transfer. The default wait is about 5 seconds.

See also

TransmitTimeout

Zmodem8K property

Description

Determines whether 8K blocks are enabled. Read/write.

Data type

`Boolean`

Syntax

expression.**Zmodem8K**[= *value*]

expression must reference an **APAXPort.**

Remarks

Default: False

See “Large Block Support” on page 411 for more information.

ZmodemFileOptions property

Description

Determines the Zmodem file management options to use. Read/write.

Data type

TZmodemFileOptions

Syntax

expression.**ZmodemFileOptions**[= *value*]

expression must reference an **APAXPort.**

Settings

Valid settings for ZmodemFileOptions are:

Constant	Description
zfoNoOption	No option
zfoWriteNewerLonger	Transfer if new, newer or longer
zfoWriteCRC	Not supported, treated same as WriteNewer
zfoWriteAppend	Transfer if new, append if exists
zfoWriteClobber	Transfer regardless
zfoWriteNewer	Transfer if new or newer
zfoWriteDifferent	Transfer if new or different dates or lengths
zfoWriteProtect	Transfer only if new

Remarks

Default: zfoWriteNewer

Regardless of the value of this property, new incoming files are accepted unless the ZmodemSkipNoFile property is set to False.

The logic that handles these file management options is executed after the OnProtocolAccept event has been generated. If you write an event handler that deals with possible overwrites, be sure to set ZmodemFileOptions to zfoWriteClobber before starting to receive.

See also

ZmodemOptionOverride, ZmodemSkipNoFile

ZmodemFinishRetry property

Description

Specifies the number of times to retry the final handshake of a Zmodem protocol session. Read/write.

Data type

`Integer`

Syntax

expression.**ZmodemFinishRetry**[= *value*]

expression must reference an **APAXPort.**

Remarks

Default: 0

A Zmodem transmitter signals that it has no more files to transmit by sending a ZFin frame. The receiver acknowledges this by sending its own ZFin frame. The transmitter then sends 'OO' as the final frame of the transfer.

The Zmodem specification indicates that this portion of the protocol isn't critical (since all files have already been completely received) and that a time-out while waiting for the response should be ignored. However, this strategy doesn't work well with DSZ, a Zmodem implementation by Omen Technology, Inc.

DSZ retries after a ZFin time-out, which can sometimes cause unneeded packet transfers when the handshake time-out is 10 seconds or less. To handle this situation, APAX mimics DSZ when ZmodemFinishRetry is set a non-zero value. It waits the number of ticks set in the FinishWait property for a response.

ZmodemFinishRetry is the number of times to resend the ZFin in response to a time-out. When ZmodemFinishRetry is zero the ZFin is sent only once. If no response is received the protocol finishes without an error.

See also

FinishWait

ZmodemOptionOverride property

Description

Determines whether a remote sender's options are ignored. Read/write.

Data type

`Boolean`

Syntax

`expression``.`**`ZmodemOptionOverride`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Default: False

If ZmodemOptionOverride is set to True, a receiving protocol component ignores the sender's options and uses its own settings for ZmodemFileOptions and ZmodemSkipNoFile. Otherwise, it uses the sender's options.

See also

ZmodemFileOptions

ZmodemRecover property

Description

Determines whether Zmodem performs file recovery. Read/write.

Data type

`Boolean`

Syntax

`expression``.`**`ZmodemRecover`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Default: False

Zmodem is capable of resuming interrupted file transfers if the receiver kept the partial file when a previous transfer was interrupted. The transmitter requests this action by setting ZmodemRecover to True. The request is transmitted to the receiver along with the file name to be recovered. If the receiver has this file, it sends back the current file size. The transmitter then adjusts its file offset and starts sending data from that point. If the receiver doesn't already have this file, a normal file transfer takes place.

See "Transfer Resume" on page 408 for more information.

See also

InitialPosition

ZmodemSkipNoFile property

Description

Determines whether a Zmodem receiver should skip all files that don't already exist. Read/write.

Data type

`Boolean`

Syntax

`expression``.`**`ZmodemSkipNoFile`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Default: False

See also

ZmodemFileOptions

Chapter 10: Data Trigger Management

The purpose of the data trigger mechanism is to provide a simple solution to the common task of looking for a particular sequence of bytes in the incoming data. Data triggers collect data that has certain properties and pass the data as a complete unit to the client application.

The Data Trigger Mechanism

The data trigger mechanism provides automatic data delivery and notification from the incoming data stream, based on simple properties set in the APAXPort control.

Data triggers automatically collect data from the incoming data stream based on the value specified in the DataTriggerString property, and deliver the data when a match is found. Data triggers also do their own buffering. This means that data will always be available for processing when the OnDataTrigger event fires.

Data triggers are typically used when the data you are looking for has a fixed length or starts and/or ends with a known string of data. These conditions can be set for each individual data trigger as they are added at run time.

Data ownership

There is no limit on the number of data triggers for a serial port. However, any incoming character can be part of only one data trigger. The first enabled data trigger that has its start condition met takes ownership of all incoming data until the trigger is complete. If a data trigger times out, the data it has collected up to that point is made available to any other enabled data triggers associated with the serial port.

Start and end conditions

The data trigger mechanism employed by the APAXPort control allows for the use of traditional DOS wildcard characters ('?' and '*'). If a question mark ('?') appears in a data trigger string, this instructs APAX to accept any single character in place of the question mark. If an asterisk ('*') appears in a data trigger string, this instructs APAX to accept any number of characters until the character immediately following the asterisk is received. In the event that you need to look for either of these two characters, they can be escaped by preceding the character with a '\'. Any character following a '\' character is treated as a literal. For example, the following string instructs APAX to search for a string beginning with "A" followed by any number of characters until the letter 'C' is found:

```
"A*C"
```

The next example string instructs APAX to search for an 'A' followed by a '*' followed by a 'C':

```
"A\*C"
```

With that said, we can discuss data trigger start conditions and end conditions. If a string contains no wildcard characters, the start condition is equal to the entire string and the end condition is also equal to the entire string. However, a data trigger string that contains wildcards has different start and end conditions. In this case, the start condition is equal to all of the characters in the string preceding the first occurrence of a wildcard character. The end condition is defined by all of the characters in the string that follow the last occurrence of a wildcard character.

You can also specify a packet size that simply defines how many characters to look for, and/or a time-out parameter that terminates the data trigger if the end condition is not satisfied within the specified time frame. If multiple end conditions are defined (end string match, specified number or characters, or time-out value), the first condition met will cause the OnDataTrigger event to fire.

Data trigger persistence

Once a data trigger is matched, an OnDataTrigger event will be fired. One of the parameters passed to an OnDataTrigger event handler, ReEnable, is a Boolean passed by reference. Within your event handler, you can set this value to True to re-enable the trigger, or you can set this value to False to disable the trigger.

Data Trigger References

Following is a list of the APAXPort control's properties, methods, and events that pertain to data triggers. This is only a subset of the functionality of the APAXPort functionality. Additional properties, methods, and events are introduced in other chapters.

Properties

DataTriggerString

Methods

AddDataTrigger	EnableDataTrigger
DisableDataTrigger	RemoveDataTrigger

Events

OnDataTrigger

Reference Section

AddDataTrigger method

Description

Adds a data trigger and returns the index of the new data trigger. Returns an Integer.

Syntax

```
expression.AddDataTrigger(
  TriggerString, PacketSize, Timeout, IncludeStrings, IgnoreCase)
```

Part	Description	Data Type
expression	An expression that returns an **APAXPort** object	**APAXPort**
TriggerString	Defines the string to match	**String**
PacketSize	Defines the size of the packet	**Integer**
Timeout	Specifies how long to wait for the string	**Integer**
IncludeStrings	Determines whether or not to include the start and stop strings in the trigger string	**Boolean**
IgnoreCase	Determines whether case is considered in the match	**Boolean**

Remarks

Use this property to add and enable a data trigger. The TriggerString parameter should contain the string to search for and may or may not include wildcards ('?' and '*'). The PacketSize parameter can be used to specify a particular count of bytes received. If PacketSize is zero, no counting of bytes occurs. Similarly, a non-zero value for Timeout instructs APAX to set an internal timer. This value is specified in clock ticks. There are approximately 18 milliseconds per clock tick. If this amount of time elapses before a valid end condition is met, the data trigger will time out and an OnDataTrigger event will fire with the Timeout parameter set to True. To bypass any time restrictions on a data trigger, set the Timeout parameter to zero.

IncludeStrings is a Boolean parameter that instructs APAX whether or not to include the start and end strings in the Data parameter passed to the OnDataTrigger event. This parameter only pertains to strings that have wildcards ('?','*'). Refer to the discussion at the beginning of this chapter for further information regarding start and end conditions. Set IncludeStrings to True to include the start and end strings in the Data parameter of the OnDataTrigger event, or set IncludeStrings to False to exclude the start and end strings in the Data parameter of the OnDataTrigger event.

The IgnoreCase parameter simply instructs APAX whether or not to respect case sensitivity when looking for a match. If IgnoreCase is True, the data trigger will only find a match when the strings match exactly.

The return value of this method contains the index of the data trigger just added. This value should be retained so that you can enable, disable, or remove the data trigger based on the index value.

See also

DisableDataTrigger, EnableDataTrigger, RemoveDataTrigger, DataTriggerString, OnDataTrigger

DataTriggerString property

Description

Contains a delimited list of all defined data triggers. Read/write.

Data type

`String`

Syntax

`expression``.`**`DataTriggerString`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Use this property to read or define multiple strings to match. This stream is delimited by the pipe character ('|').

Example

The following example defines 3 distinct strings to search for, specifically Login, Logout, and Error:

```
  APAX1.DataTriggerString = "Login|Logout|Error"
```

See also

AddDataTrigger, DisableDataTrigger, EnableDataTrigger, RemoveDataTrigger, OnDataTrigger

DisableDataTrigger method

Description

Disables the data trigger specified by the Index parameter.

Syntax

expression.**DisableDataTrigger**(***Index***)

Part	**Description**	**Data Type**
expression	An expression that returns an **APAXPort** object	**APAXPort**
Index	Defines the index of the data trigger	**Integer**

Remarks

Each data trigger is uniquely identified by an index. This index is returned to you from the AddDataTrigger method. Calling this method disables the data trigger corresponding to the index value. Triggers that are disabled will not search for strings or fire OnDataTrigger events.

See also

AddDataTrigger, EnableDataTrigger, RemoveDataTrigger, DataTriggerString, OnDataTrigger

EnableDataTrigger method

Description

Enables the data trigger specified by the Index parameter.

Syntax

expression.**EnableDataTrigger(*Index*)**

Part	**Description**	**Data Type**
expression	An expression that returns an **APAXPort** object	**APAXPort**
Index	Defines the index of the data trigger	**Integer**

Remarks

Each data trigger is uniquely identified by an index. This index is returned to you from the AddDataTrigger method. Calling this method enables the data trigger corresponding to the index value. A data trigger must be enabled to search for strings and fire OnDataTrigger events.

See also

AddDataTrigger, DisableDataTrigger, RemoveDataTrigger, DataTriggerString, OnDataTrigger

OnDataTrigger event

Description

Defines an event that is fired when one of the defined data triggers finds a match.

Syntax

```
Private Sub expression_OnDataTrigger(ByVal Index As Long,
  ByVal Timeout As Boolean, ByVal Data As Variant,
  ByVal Size As Long, ReEnable As Boolean)
```

Part	**Description**	**Data Type**
expression	References the APAXPort object that fired the event	**APAXPort**
Index	Specifies the index of the matched data trigger	**Long**
Timeout	Indicates whether the data trigger timed out	**Boolean**
Data	Contains the matched data	**Variant**
Size	Specifies the length or size of the Data parameter	**Long**
ReEnable	Allows you to re-enable the data trigger	**Boolean**

Remarks

This event is fired automatically when a data trigger has found a match or timed out. The Index parameter defines the index of the data trigger that found a match. Timeout will be True for any data trigger that timed out prior to finding a match and False otherwise. The Data parameter refers to the actual data acquired from the data trigger and the Size indicates the length (in bytes) of the Data parameter.

Within your event handler, you can set the ReEnable property to True to once again enable the data trigger, or set it to False to disable the data trigger.

See also

AddDataTrigger, DisableDataTrigger, EnableDataTrigger, RemoveDataTrigger, DataTriggerString

RemoveDataTrigger method

Description

Deletes the data trigger specified by the Index parameter.

Syntax

`expression``.`**`RemoveDataTrigger(`*`Index`*`)`**

Part	**Description**	**Data Type**
expression	An expression that returns an APAXPort object	***APAXPort***
Index	Defines the index of the data trigger	***Integer***

Remarks

Each data trigger is uniquely identified by an index. This index is returned to you from the AddDataTrigger method. Calling this method permanently removes the data trigger corresponding to the index value.

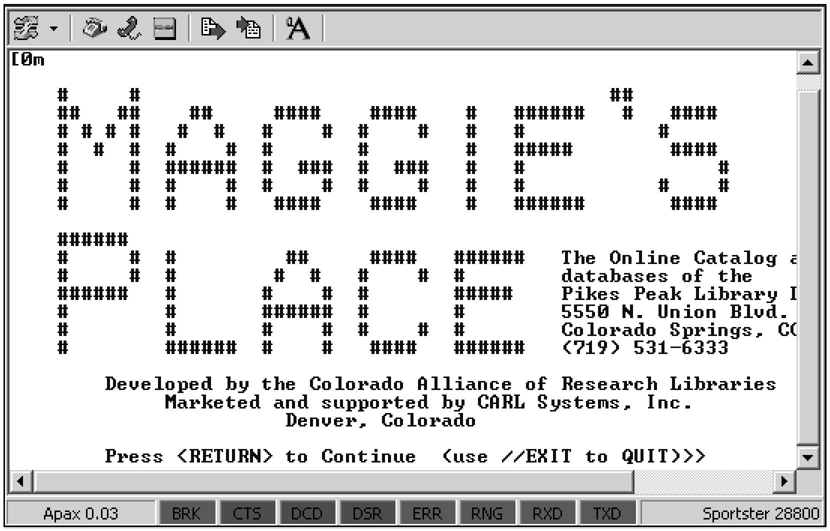
See also

AddDataTrigger, DisableDataTrigger, EnableDataTrigger, DataTriggerString, OnDataTrigger

Chapter 11: Visual Elements

The APAXPort control has two customizable visual elements: the toolbar and the status bar. The toolbar is located at the top of the control and the status bar is located at the bottom of the control. Both of these elements can be hidden or made visible to the user. These elements are discussed in detail in this chapter.

The toolbar consists of a series of buttons which can be hidden or displayed on an individual basis. Each of the buttons on the toolbar has default functionality built in to the OnClick event handler. This default functionality can be used as is without writing a single line of code, or you can completely override this functionality by supplying your own OnXxxButtonClick event handler in which you set the Default parameter to False. Additionally, you can supply your own OnXxxButtonClick event handler and set the Default parameter to True. When the Default parameter is set to True, any code that is supplied in your event handler will be executed prior to the default code execution. The reference section that describes the toolbar immediately follows the reference section on the status bar.



```
[0m

    #     #                                       ##
    ##   ##    ##      ####    ####    #   ######   #   ####
    # # # #   #  #   #    #  #    #   #   #           #
    #  #  #  #    #  #       #        #   #####        ####
    #     #  ######  #  ###  #  ###   #   #                #
    #     #  #    #  #    #  #    #   #   #           #    #
    #     #  #    #   ####    ####    #   ######       ####

    ######
    #     #  #         ##     ####   ######   The Online Catalog a
    #     #  #        #  #   #    #  #        databases of the
    ######   #       #    #  #       #####    Pikes Peak Library I
    #        #       ######  #       #        5550 N. Union Blvd.
    #        #       #    #  #    #  #        Colorado Springs, C
    #        ######  #    #   ####   ######   (719) 531-6333

        Developed by the Colorado Alliance of Research Libraries
             Marketed and supported by CARL Systems, Inc.
                       Denver, Colorado

        Press <RETURN> to Continue  (use //EXIT to QUIT)>>
```

The status bar is comprised of three panels. The first panel simply displays the current version of APAX. The second panel contains the status lights and is discussed in more detail later in this chapter. The third and final panel of the status bar simply displays the selected communications device. The selected

11

communications device will either be a serial port (COM1), a TAPI device, or Winsock.The APAXPort control allows you to add a status light display, similar to the LEDs found on external modems, to your communications programs. APAX monitors the status of the associated serial port and changes the state of the display lights correspondingly. The goal of the component is to give communications programs a status light display similar to the LEDs found on external modems.

APAX is capable of monitoring the serial port's line signals (DCD, DTR, CTS, and RI), line breaks and errors, and whether data is currently being received or transmitted.

Following is a list of all status lights and the port condition they monitor:

Status Light	**Line condition**
BRKLight	Lit for BreakOffTimeout ticks when a line break occurs
CTSLight	Lit when CTS signal high
DCDLight	Lit when DCD signal high
DSRLight	Lit when DSR signal high
ERRLight	Lit for ErrorOffTimeout ticks when a line error occurs
RNGLight	Lit for RingOffTimeout ticks after RI signal goes high
RXDLight	Lit when data is being received
TXDLight	Lit when data is being transmitted

The mechanism employed by APAX for displaying status lights is very simple to use, and only requires two steps. First, the panel on which the lights reside must be made visible by setting the ShowStatusBar property to True. Second, the ShowLights property must be set to True.

The status light functionality of the APAXPort control simply displays two bitmaps, or two different colors, depending on whether the light is "lit" or "unlit." The corresponding modem status signal determines which of the two states is displayed.

Status Bar References

The status bar is displayed by setting the ShowStatusBar property to True. The status bar consists of a Caption panel, status light panels (BRK, CTS, DCD, ERR, RNG, RXD, and TXD) and an end panel that displays the current DeviceType. The status lights are displayed by setting the ShowLights property to True.

Properties

Caption	LightsNotLitColor	ShowStatusBar
CaptionAlignment	LightWidth	Version
CaptionWidth	ShowLightCaptions	
LightsLitColor	ShowLights	

11

Reference Section

Caption property

Description

Determines the caption displayed in the lower left section of the status bar. Read/write.

Data type

`String`

Syntax

`expression.``Caption [= `*`value`*`]`

`expression`` must reference an `**`APAXPort.`**

Remarks

The caption is displayed only if the ShowStatusBar property is set to True.

See also

CaptionAlignment, CaptionWidth, ShowStatusBar

CaptionAlignment property

Description

Determines the horizontal alignment of the status bar caption. Read/write.

Data type

`TAlignment`

Syntax

`expression.``CaptionAlignment [= `*`value`*`]`

`expression`` must reference an `**`APAXPort.`**

Settings

Constant	Description
taLeftJustify	Caption is left justified
taRightJustify	Caption is right justified
taCenter	Caption is centered

Remarks

The caption is displayed and aligned only if the ShowStatusBar property is set to True.

See also

Caption, CaptionWidth

CaptionWidth property

Description

Determines the width of the status bar caption area. Read/write.

Data type

`Integer`

Syntax

`*expression*.**CaptionWidth**[= *value*]`

`*expression* must reference an **APAXPort.**`

Remarks

The status lights and the selected device panels are not automatically resized when the CaptionWidth property is changed. If the CaptionWidth is increased, the status lights and selected device panels are shifted to the right. If the CaptionWidth is decreased, the status lights and selected device panels are shifted to the left.

See also

Caption, CaptionAlignment

LightsLitColor property

Description

Determines the color of the status lights when they are in the set (asserted or True) state. Read/write.

Data type

`TColor`

Syntax

`expression``.`**`LightsLitColor`**`[= `*`value`*`]`

`expression`` must reference an `**`APAXPort.`**

Remarks

The actual state (lit or unlit) of the individual lights (BRK, CTS, DCD, DSR, ERR, RNG, RXD, and TXD) is determined automatically by the modem state. The LightsLitColor property merely defines the color of the light when the corresponding modem state signal is asserted.

See also

LightsNotLitColor

LightsNotLitColor property

Description

Determines the color of the status lights when they are in the cleared (False) state. Read/write.

Data type

`TColor`

Syntax

`expression``.`**`LightsNotLitColor`**`[= `*`value`*`]`

`expression`` must reference an `**`APAXPort.`**

Remarks

The actual state (lit or unlit) of the individual lights (BRK, CTS, DCD, DSR, ERR, RNG, RXD, and TXD) is determined automatically by the modem state. The LightsNotLitColor property merely defines the color of the light when the corresponding modem state signal is lowered.

See also

LightsLitColor

LightWidth property

Description

Determines the width of the status light displays. Read/write.

Data type

`Integer`

Syntax

`expression``.`**`LightWidth`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

The width of individual status lights cannot be set directly. Instead, this property setting applies to all status lights.

Status lights are displayed only if the ShowStatusBar property is set to True.

See also

ShowLightCaptions, ShowLights

ShowLightCaptions property

Description

Determines whether captions are superimposed on the status light displays. Read/write.

Data type

`Boolean`

Syntax

`expression.`**`ShowLightCaptions`**`[= value]`

`expression must reference an `**`APAXPort.`**

Remarks

By default, the status lights have their corresponding caption superimposed on the individual lights (for example, "TXD", "RXD", "RNG", etc.) Set the ShowLightCaptions property to False to override this default behavior.

See also

ShowLights, LightWidth

ShowLights property

Description

Determines whether or not the status light indicators are visible. Read/write.

Data type

`Boolean`

Syntax

`expression.`**`ShowLights`**`[= value]`

`expression must reference an `**`APAXPort.`**

Remarks

Individual status lights cannot be singled out and hidden or made visible. Setting the ShowLights property displays all status lights and conversely, setting ShowLights to True displays all status lights.

See also

LightWidth, ShowLightCaptions

ShowStatusBar property

Description

Determines whether the status bar is visible. Read/write.

Data type

`Boolean`

Syntax

`expression``.`**`ShowStatusBar`**`[= `*`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

The status bar is comprised of the caption panel, status light indicators, and the selected device panel. Setting the ShowStatusBar property to False will hide all of these constituent components.

See also

ShowToolBar, Visible

Version property

Description

Indicates the current version of APAX. Read-only.

Data type

`String`

Syntax

`expression``.`**`Version`**

`expression` `must reference an` **`APAXPort.`**

Remarks

This property is particularly useful when dealing with technical support.

Toolbar References

The toolbar is displayed by setting the ShowToolBar property to True. Buttons are arranged on the toolbar in groups according to their functionality, and each group can be displayed or not. Each button generates an event that can be used to override or enhance the default functionality.

Properties

ShowConnectButtons	ShowProtocolButtons	ShowToolBar
ShowDeviceSelButton	ShowTerminalButtons	

Events

OnCloseButtonClick	OnFontButtonClick	OnSendButtonClick
OnConnectButtonClick	OnListenButtonClick	
OnDeviceButtonClick	OnReceiveButtonClick	

Reference Section

OnCloseButtonClick event

Description

Defines an event that is fired when the user clicks the Close button on the toolbar.

Syntax

`Private Sub` *`expression`*`_`**`OnCloseButtonClick(`***`Default`*** `as Boolean)`**

Part	**Description**	**Data Type**
expression	References the APAXPort object that fired the event	**APAXPort**
Default	Determines whether APAX implements default handling	**Boolean**

Remarks

This event provides you the opportunity to override the default APAX Close button behavior. The Default parameter can be set to False in your event handler to completely circumvent the default APAX behavior. Setting the Default value to True in your event handler allows the code in your event handler to execute, followed by the execution of the default APAX code.

See also

OnCloseButtonClick, OnDeviceButtonClick, OnFontButtonClick, OnListenButtonClick, OnReceiveButtonClick, OnSendButtonClick

OnConnectButtonClick event

Description

Defines an event that is fired when the user clicks the Connect button on the toolbar.

Syntax

`Private Sub` *`expression_`***`OnConnectButtonClick(`***`Default`* **`as Boolean)`**

Part	Description	Data Type
expression	References the APAXPort object that fired the event	**APAXPort**
Default	Determines whether APAX implements default handling	**Boolean**

Remarks

This event provides you the opportunity to override the default APAX Connect button behavior. The Default parameter can be set to False in your event handler to completely circumvent the default APAX behavior. Setting the Default value to True in your event handler allows the code in your event handler to execute, followed by the execution of the default APAX code.

See also

OnCloseButtonClick, OnDeviceButtonClick, OnFontButtonClick, OnListenButtonClick, OnReceiveButtonClick, OnSendButtonClick

OnDeviceButtonClick event

Description

Defines an event that is fired when the user clicks the Device Select button on the toolbar.

Syntax

`Private Sub` *`expression`*`_`**`OnDeviceButtonClick(`***`Default`* **`as Boolean)`**

Part	Description	Data Type
expression	References the APAXPort object that fired the event	**APAXPort**
Default	Determines whether APAX implements default handling	**Boolean**

Remarks

This event provides you the opportunity to override the default APAX Device Select button behavior. The Default parameter can be set to False in your event handler to completely circumvent the default APAX behavior. Setting the Default value to True in your event handler allows the code in your event handler to execute, followed by the execution of the default APAX code.

See also

OnCloseButtonClick, OnConnectButtonClick, OnFontButtonClick, OnListenButtonClick, OnReceiveButtonClick, OnSendButtonClick

OnFontButtonClick event

Description

Defines an event that is fired when the user clicks the Font button on the toolbar.

Syntax

```
Private Sub expression_OnFontButtonClick(Default as Boolean)
```

Remarks

Part	Description	Data Type
expression	References the APAXPort object that fired the event	**APAXPort**
Default	Determines whether APAX implements default handling	**Boolean**

This event provides you the opportunity to override the default APAX Font button behavior. The Default parameter can be set to False in your event handler to completely circumvent the default APAX behavior. Setting the Default value to True in your event handler allows the code in your event handler to execute, followed by the execution of the default APAX code.

See also

OnCloseButtonClick, OnConnectButtonClick, OnDeviceButtonClick, OnListenButtonClick, OnReceiveButtonClick, OnSendButtonClick

OnListenButtonClick event

Description

Defines an event that is fired when the user clicks the Listen button on the toolbar.

Syntax

```
Private Sub expression_OnListenButtonClick(Default as Boolean)
```

Part	Description	Data Type
expression	References the APAXPort object that fired the event	**APAXPort**
Default	Determines whether APAX implements default handling	**Boolean**

Remarks

This event provides you the opportunity to override the default APAX Listen button behavior. The Default parameter can be set to False in your event handler to completely circumvent the default APAX behavior. Setting the Default value to True in your event handler allows the code in your event handler to execute, followed by the execution of the default APAX code.

See also

OnCloseButtonClick, OnConnectButtonClick, OnDeviceButtonClick, OnFontButtonClick, OnReceiveButtonClick, OnSendButtonClick

OnReceiveButtonClick event

Description

Defines an event that is fired when the user clicks the Receive File button on the toolbar.

Syntax

`Private Sub` *`expression_`***`OnReceiveButtonClick(`***`Default`* **`as Boolean)`**

Part	Description	Data Type
expression	References the APAXPort object that fired the event	**APAXPort**
Default	Determines whether APAX implements default handling	**Boolean**

Remarks

This event provides you the opportunity to override the default APAX Receive File button behavior. The Default parameter can be set to False in your event handler to completely circumvent the default APAX behavior. Setting the Default value to True in your event handler allows the code in your event handler to execute, followed by the execution of the default APAX code.

See also

OnCloseButtonClick, OnConnectButtonClick, OnDeviceButtonClick, OnFontButtonClick, OnListenButtonClick, OnSendButtonClick

OnSendButtonClick event

Description

Defines an event that is fired when the user clicks the Send File button on the toolbar.

Syntax

`Private Sub` *`expression`*`_`**`OnSendButtonClick(`***`Default`*** **`as Boolean)`**

Part	Description	Data Type
expression	References the APAXPort object that fired the event	**APAXPort**
Default	Determines whether APAX implements default handling	**Boolean**

Remarks

This event provides you the opportunity to override the default APAX Send File button behavior. The Default parameter can be set to False in your event handler to completely circumvent the default APAX behavior. Setting the Default value to True in your event handler allows the code in your event handler to execute, followed by the execution of the default APAX code.

See also

OnCloseButtonClick, OnConnectButtonClick, OnDeviceButtonClick, OnFontButtonClick, OnListenButtonClick, OnReceiveButtonClick

ShowConnectButtons property

Description

Determines whether the Close, Listen, and Connect buttons are visible on the toolbar. Read/write.

Data type

`Boolean`

Syntax

`expression``.`**`ShowConnectButtons`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

The Close, Listen, and Connect buttons are treated as an individual unit due to the fact that their functionality is tightly coupled. Individual buttons from this set cannot be singled out and hidden or made visible. Setting the ShowConnectButtons to False hides all three buttons and conversely, setting ShowConnectButtons to True will force all three buttons to their visible state.

This property setting has no effect unless the ShowToolBar property is set to True.

See also

ShowToolBar, ShowDeviceSelButton, ShowProtocolButtons, ShowTerminalButtons

ShowDeviceSelButton property

Description

Determines whether the Device Select button is visible on the toolbar. Read/write.

Data type

`Boolean`

Syntax

`expression``.`**`ShowDeviceSelButton`**`[= `*`value`*`]`

`expression`` must reference an `**`APAXPort.`**

Remarks

Setting the ShowDeviceSelButton property to False hides the Device Select button and conversely, setting the ShowDeviceSelButton property to True forces the Device Select button to its visible state.

This property setting has no effect unless the ShowToolBar property is set to True.

See also

ShowToolBar, ShowConnectButtons, ShowProtocolButtons, ShowTerminalButtons

ShowProtocolButtons property

Description

Determines whether the Send File and Receive File buttons are visible on the toolbar. Read/write.

Data type

`Boolean`

Syntax

*`expression.`***`ShowProtocolButtons`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

The Send File and Receive File buttons are treated as an individual unit due to the fact that their functionality is tightly coupled. Individual buttons from this set cannot be singled out and hidden or made visible. Setting the ShowProtocolButtons to False hides both buttons and conversely, setting ShowProtocolButtons to True will force both buttons to their visible state.

This property setting has no effect unless the ShowToolBar property is set to True.

See also

ShowToolBar, ShowConnectButtons, ShowDeviceSelButton, ShowTerminalButtons

ShowTerminalButtons property

Description

Determines whether the Font button is visible on the toolbar. Read/write.

Data type

`Boolean`

Syntax

*`expression.`***`ShowTerminalButtons`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Setting the ShowTerminalButtons property to False hides the Terminal Font Select button and conversely, setting the ShowTerminalButtons property to True forces the Terminal Font Select button to its visible state.

This property setting has no effect unless the ShowToolBar property is set to True.

See also

ShowToolBar, ShowConnectButtons, ShowDeviceSelButton, ShowProtocolButtons

ShowToolBar property

Description

Determines whether the toolbar is visible. Read/write.

Data type

`Boolean`

Syntax

*`expression.`***`ShowToolBar`**`[=` *`value`*`]`

`expression` `must reference an` **`APAXPort.`**

Remarks

Setting the ShowToolBar property to False hides the toolbar and all of its buttons. Conversely, setting the ShowToolBar property to True shows the toolbar and also shows any constituent tool buttons whose corresponding ShowXxx property is set to True.

See also

ShowStatusBar, Visible

Appendix

The following properties, methods, and events are included in the APAX type library for backwards compatibility with MSCom™ and PDQCom™.

Properties

About	Download	PortOpen
AnswerBack	DSRHolding	RThreshold
Appearance	Echo	RTSEnable
AreaCode	Emulation	ScrollRows
AutoProcess	FontBold	Settings
AutoScroll	FontItalic	SmoothScroll
AutoSize	FontName	SThreshold
AutoZModem	FontSize	TapiName
BackColor	FontUnderline	TapiVersion
BackSpace	ForeColor	Text
Break	Handshaking	Upload
CallHandle	hWnd	XferCarrierAbort
CaptureFilename	InBufferCount	XferDestFilename
CDHolding	InBufferSize	XferDialogHeight
ColorFilter	Input	XferDialogLeft
CommEvent	InputLen	XferDialogTop
CommID	Interval	XferDialogWidth
CommPort	InTimeout	XferFileDate
CountryCode	KeyTranslation	XferFileSize
CTSHolding	LineHandle	XferFileTime
CurrentDevice	LineInput	XferMessage
CursorColumn	Location	XferProtocol
CursorRow	NullDiscard	XferSourceFilename
CursorType	OutBufferCount	XferStatus
DeviceName	OutBufferSize	XferStatusDialog
Devices	Output	XferTransferred
Disp	ParityReplace	

Methods

CloseCommHandle	HangUp	SuspendComm
CRC16	PlaceCall	TranslateNumber
CRC32	SetCallSettings	WaitForCall
EnumTapiDevices	SetDeviceByName	
EnumTapiLocations	SetModemSettings	

Events

CallState	OnComm
LocationChange	OnTapi

Constants

The following constants are included in the APAX type library for backwards compatibility with MSCom and PDQCom.

AutoProcess constants

Constant	Value
PDQ_AUTOPROCESS_NONE	0
PDQ_AUTOPROCESS_SERIAL	1
PDQ_AUTOPROCESS_KEY	2
PDQ_AUTOPROCESS_BOTH	3

AutoScroll constants

Constant	Value
PDQ_AUTOSCROLL_NONE	0
PDQ_AUTOSCROLL_VERTICAL	1
PDQ_AUTOSCROLL_HORIZONTAL	2
PDQ_AUTOSCROLL_BOTH	3
PDQ_AUTOSCROLL_VERTKEY	4

Backspace constants

Constant	Value
PDQ_BACKSPACE_DESTRUCTIVE	0
PDQ_BACKSPACE_NON_DESTRUCTIVE	1

Capture constants

Constant	Value
PDQ_CAPTURE_STANDARD	0
PDQ_CAPTURE_BINARY	1
PDQ_CAPTURE_VISIBLE	2

ColorFilter constants

Constant	Value
PDQ_COLOR_FULL	0
PDQ_COLOR_GREY	1
PDQ_COLOR_MONO	2

Cursor Type constants

Constant	Value
PDQ_CURSOR_VBAR	0
PDQ_CURSOR_BLOCK	1

Emulation constants

Constant	Value
PDQ_EMULATION_NONE	0
PDQ_EMULATION_TTY	1
PDQ_EMULATION_ANSI	2
PDQ_EMULATION_VT52	3
PDQ_EMULATION_VT100	4

Errors constants

Constant	Value
PDQ_ER_BREAK	1001
PDQ_ER_CTSTO	1002
PDQ_ER_DSRTO	1003
PDQ_ER_FRAME	1004
PDQ_ER_INTO	1005
PDQ_ER_OVERRUN	1006
PDQ_ER_CDTO	1007
PDQ_ER_RXOVER	1008
PDQ_ER_RXPARITY	1009
PDQ_ER_TXFULL	1010

Events constants

Constant	Value
PDQ_EV_SEND	1
PDQ_EV_RECEIVE	2
PDQ_EV_CTS	3
PDQ_EV_DSR	4
PDQ_EV_CD	5
PDQ_EV_RING	6
PDQ_EV_EOF	7
PDQ_EV_ZMODEM	8
PDQ_EV_XFER	100

Handshake constants

Constant	Value
PDQ_HANDSHAKING_NONE	0
PDQ_HANDSHAKING_XON	1
PDQ_HANDSHAKING_RTS	2
PDQ_HANDSHAKING_BOTH	3

Key Translation constants

Constant	Value
PDQ_KEY_NONE	0
PDQ_KEY_MANUAL	1
PDQ_KEY_ANSI	2
PDQ_KEY_VT52	3
PDQ_KEY_VT100	4

PDQColorConstants constants

Color	Value	RGB Value (Hex)
PDQ_COLOR_BLACK	0	&H000000
PDQ_COLOR_BLUE	1	&H800000
PDQ_COLOR_GREEN	2	&H008000
PDQ_COLOR_CYAN	3	&H808000
PDQ_COLOR_RED	4	&H000080
PDQ_COLOR_MAGENTA	5	&H800080
PDQ_COLOR_YELLOW	6	&H008080
PDQ_COLOR_WHITE	7	&HC0C0C0

Color	Value	RGB Value (Hex)
PDQ_COLOR_GRAY	8	&H808080
PDQ_COLOR_LIGHTBLUE	9	&HFF0000
PDQ_COLOR_LIGHTGREEN	10	&H00FF00
PDQ_COLOR_LIGHTCYAN	11	&HFFFF00
PDQ_COLOR_LIGHTRED	12	&H0000FF
PDQ_COLOR_LIGHTMAGENTA	13	&HFF00FF
PDQ_COLOR_LIGHTYELLOW	14	&H00FFFF
PDQ_COLOR_BRIGHTWHITE	15	&HFFFFFF

XferProtocol constants

Constant	Value	Protocol
PDQ_XMODEM_CHECKSUM	0	XModem-Checksum
PDQ_XMODEM_CRC	1	XModem-CRC
PDQ_XMODEM_1K	2	XModem-1K
PDQ_YMODEM_BATCH	3	YModem-Batch
PDQ_YMODEM_G	4	YModem-G
PDQ_ZMODEM	5	ZModem
PDQ_KERMIT	7	Kermit

XferStatus constants

Constant	Value
PDQ_XFER_TERM_OK	0
PDQ_XFER_WAITING	1
PDQ_XFER_FILE_READY	2
PDQ_XFER_FILE_START	3
PDQ_XFER_XFERING	4
PDQ_XFER_SKIP	5
PDQ_XFER_ABORT	6
PDQ_XFER_FINISHED	7
PDQ_XFER_LOSTCARRIER	8
PDQ_XFER_TIMEOUT	9
PDQ_XFER_TERM_ERROR	-1

XferStatusDialog constants

Constant	Value
PDQ_XFERDIALOG_NONE	0
PDQ_XFERDIALOG_MODELESS	1
PDQ_XFERDIALOG_MODAL	2

TAPI constants

Constant	Value
PDQ_CALLSTATE_ACCEPTED	4
PDQ_CALLSTATE_BUSY	64
PDQ_CALLSTATE_CONFERENCED	2048
PDQ_CALLSTATE_CONNECTED	256
PDQ_CALLSTATE_DIALING	16
PDQ_CALLSTATE_DIALTONE	8
PDQ_CALLSTATE_DISCONNECTED	16384
PDQ_CALLSTATE_IDLE	1
PDQ_CALLSTATE_OFFERING	2
PDQ_CALLSTATE_ONHOLD	1024
PDQ_CALLSTATE_ONHOLDPENCONF	4096
PDQ_CALLSTATE_ONHOLDPENDTRANSFER	8192
PDQ_CALLSTATE_PROCEEDING	512
PDQ_CALLSTATE_RINGBACK	32
PDQ_CALLSTATE_SPECIALINFO	128
PDQ_CALLSTATE_UNKNOWN	32768

Identifier Index

Subject Index