VCL components for advanced communications

Async Professional™ 4

DEVELOPER'S GUIDE

A "how-to" guide for getting the most from Async Professional

TURBOPOWER®
Software Company

Async Professional 4™
Developer's Guide

TurboPower Software Company
Colorado Springs, CO

www.turbopower.com

First Edition January 1998
Second Edition November 1999
Third Edition August 2001

License Agreement

Table of Contents

Chapter 1: Introduction

Whether you are new to Async Professional 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 APRO for your serial communications needs.

Since the first version of APRO back in 1991, TurboPower has invested several thousand man-hours designing, developing, testing, refining, and supporting it. We are very proud of the awards APRO has won, and the good reviews it has received.

Async Professional 4 builds on this award-winning foundation to add the features you need for computing beyond the desktop. With APRO your programs can send alphanumeric pages to anyone in the world, even across the Internet. You can retrieve files from FTP servers anywhere. You can even add Dial-Up Networking capabilities to your programs with easy-to-use new VCL controls. We also added an advanced terminal control, automated scripting for complex communications tasks, and even controls for building multi-user fax server solutions. We think you will like the new features we added, because many of the ideas for them came from customers like you!

In fact, comments received from earlier users of APRO served as the genesis for the manual you are reading now. We are very pleased to provide this *Developer's Guide* as a complement to the comprehensive information in our *Reference Guide*.

After the Chapter 1 introductory material, Chapter 2 of this guide gives a brief overview of communications concepts, and describes how the major APRO components fit into the overall communications picture. Chapter 3 provides a more advanced discussion of some of the principles of serial communications and the general issues of serial communications under Windows. Chapter 4 includes overviews and trouble-shooting guides for some of the more problematic areas of serial communication. Chapter 5 has almost 50 tutorial topics, describing how to do most common tasks. Chapter 6 is a guide to the larger demonstration programs we include with APRO.

We wrote the *Developer's Guide* to give you a head start using the Async Professional components. We also hope you will use it to explore some of the library's new areas and functionality. With the *Developer's Guide* in hand, we expect that you will discover new and exciting ways to implement APRO technology in many of your creations.

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 Async Professional.

System Requirements

To use Async Professional, you must have the following hardware and software:

1. A computer capable of running MicroSoft Windows 95/98/ME or Windows NT/2000. A minimum of 16MB of RAM is recommended.
2. Delphi 3 or later or C++Builder 3 or later.
3. A hard disk with at least 50MB of free space is strongly recommended. To install all Async Professional files and compile the example programs requires about 50MB of free disk space for all supported compilers.
4. To rebuild the Async Professional fax printer drivers, you will need Delphi 1.02, and Microsoft Visual C++ 4.0 or greater.

Installation

Async Professional can be installed directly from the CD-ROM (or diskettes) or you can copy the complete set of installation files to your hard disk and run the installation from there. Before installing the product, please scan through the README.HLP file to find any late news that may affect installation.

The setup program

Insert the TurboPower Product Suite CD and follow the instructions presented by the SETUP program.

If you are installing from diskettes, run SETUP. EXE to start the installation process. Since SETUP is a Windows application, you must start Windows first. From the taskbar select Start and then Run and type X:\SETUP. Replace X:\ with the appropriate drive letter or the name of a directory if you copied the distribution files to your hard disk.

SETUP installs Async Professional in the C:\APRO directory by default. You can specify a different directory, if desired.

SETUP creates a new program group named Async Professional (you can specify a different name, if desired) and puts the following program icons in it:

Async Professional help

The Async Professional help file.

Last minute news about Async Professional

A text file that describes changes to the documentation and new features added after the manual was printed. Please read this file before using Async Professional.

Installing for multiple compiler versions

Async Professional supports Borland Delphi 3.0 or greater and Borland C++Builder 3.0 or greater. However, the compiled file format (DCU format) is different for each Delphi version and the header files (HPP format) are different for each version of C++Builder. Since the complete source code for Async Professional is included, it is a simple matter to recompile the library when switching among versions. You can have Delphi automatically do this for you by deleting all the Async Professional DCU files when you switch environments. The only files that differ among the Delphi environments are the compiled DCU files.

Alternatively, if you switch versions frequently, it might be more convenient to keep separate copies of Async Professional installed for each Delphi environment. There is no option in the SETUP program to do this automatically, but you can simply run the SETUP program as many times as necessary and specify different directories each time. If you install multiple copies of Async Professional, make sure to point each Delphi version to its unique directory when adding the components to the palette.

If you are installing support for multiple versions of C++Builder, the headers for the highest selected C++Builder version will be placed in the APRO source folder, and the headers for other selected C++Builder versions will be placed in subdirectories. For example, if you choose to install for C++Builder 3 and 4, the headers for C++Builder 3 will be placed in the APRO\HPP3 folder.

If you need to make changes to any of the included resource files, be aware that the *.RC files should be compiled to *.R16 (for 16-bit), and to *.R32 (for 32-bit), rather than to the default *.RES. This prevents naming conflicts. The 16-bit resources are only required when rebuilding the Windows 95/98/ME printer driver.

Component installation

Installing Packages

To avoid version conflicts with applications using different versions of the Async Professional packages, each version of APRO comes with packages using slightly different names. The APRO packages have the following form:

```
  ANnn_YVv.DPL (or .BPL for Delphi 4 and 5 and C++Builder 3 and 4)
```

Nnn is the version number of APRO. Y indicates whether the package is a run-time package (R) or a design-time package (D). Vv is the version of VCL supported. For example, the run-time package name for APRO 4.00 for Delphi 4 is A400_R40.DPL.

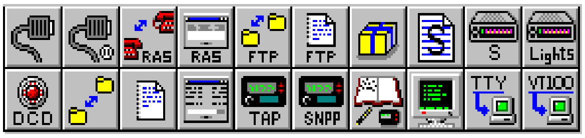
Async Professional help and the design-time package are automatically installed into the VCL if Async Professional is installed using the setup program. If you are upgrading by using a patch or the installation program simply failed to handle this properly, you will need to use the “Install Packages” menu option to install the APRO design-time package: (APNnn_DVv.DPL, or APNnn_DVv.BPL, where Nnn is the APRO version number and Vv is the VCL version number.

In order for the run-time packages to be “seen” by the VCL (and the design-time package) you need to copy them to the Windows system directory (\System for Win95/98/ME or \System32 for NT/2000) or add the Async Professional directory to your path. Another option is to copy the APRO DPL or BPL files to the Delphi\BIN directory.

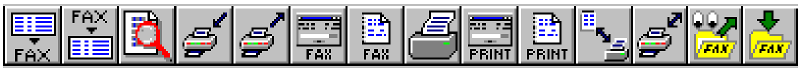
Be sure to alter the library path so that it includes the path to the APRO source files (Tools | Environment Options—Library Page) or add the APRO path to your system Path environment. This allows the compiler to find the APRO source files when required.

The component palette is updated with three tabs: “APro”, “APro Fax”, and “APro TAPI.”

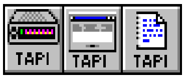
The “APro” tab provides access to the standard comport, advanced terminal, and protocol components:



The “APro Fax” tab provides access to the fax components:



The “APro TAPI” tab provides access to the TAPI components:



These same icons are used throughout the Developer’s Guide and Reference Guide when referring to the Async Professional components.

When you do the component installation, C++Builder automatically generates C++ header files for each source unit in the Async Professional library. The header file name is the unit name plus the extension HPP.

Installing integrated help

The Async Professional help system is typically installed into Delphi and C++Builder by the SETUP program, but steps for manual installation are provided if the need arises.

Installing for Delphi 4 and greater and C++Builder

Use the Help | Customize options from the IDE to install APRO32.HLP. Please refer to your compiler’s documentation for details.

Installing for Delphi 3.0

To manually install Async Professional help into Delphi 3, perform the following steps to install APRO32.HLP:

Edit the Delphi3.cnt file (in the Delphi Help directory) and add the following line to the “index” section:

```
:Index Async Professional Reference =Apro32.hlp
```

The first time you attempt to access Async Professional help, Delphi/Windows won't be able to locate the help file and will ask if you want to find the file yourself. Answer yes to this question and browse for the Async Professional help file (APRO32.HLP), which should be in the \ASYNCPRO directory. This step will only be required the first time you access Async Professional’s help.

Technical Support

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

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. So you will have another way to find answers to your questions even when we're not available.

Other support options are described in the product support brochure included with Async Professional. You can also read about support options at www.turbopower.com/support.

Chapter 2: Communications Basics

Communications is a very difficult field of programming. Several factors contribute to this perception. Firstly, many areas of communications programming lack sufficiently thorough (and understandable) documentation. Secondly, 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 Async Professional is to insulate you from these difficulties. It's very possible for an average programmer to work some communications functionality into an application with a minimum of communications knowledge by using a high level library like Async Professional. 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 major APRO components fit into the picture. If you have been programming in the field of communications for some time, you may be tempted to skip this chapter—but there's always the possibility that you may pick up a few tips here and there regarding how the different Async Professional components fit into the overall equation. If you have a need for some more detailed information, please refer to "Chapter 3: Advanced Communication Principles" on page 17.

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 these days—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 components available in Async Professional, and their purpose in life. Be sure to refer to the applicable sections of the *Reference Guide* to get more information about the various APRO components.

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, text, or whatever—it's always treated as a stream of bytes by the low-level hardware and drivers. The method of communication may differ slightly—it 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.

The TApdComPort and TApdWinsockPort are low-level components designed to handle the actual sending and receiving of data. The TApdComPort is designed to interface with a standard RS-232 or RS-485 serial port via the Windows communications drivers. The TApdWinsockPort is derived from the TApdComPort, retains all of its functionality, and adds the ability to communicate over a network connection using Winsock services.

These low-level components provide an interface to the rest of the world for your program. You'll find that most other components in Async Professional rely on these components in some way. An additional low-level component, the TApdDataPacket component, is designed to help you identify and receive specific data that you expect to come in through the comport. This component is very helpful when you are looking for specific data in the incoming data stream.

Terminals and Emulators

Terminals display data, emulators decide what the data should be. They are always used together in Async Professional; 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 PC keystrokes into their terminal equivalents. With many terminals, pressing some keys 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. Async Professional comes with two emulator components: the teletype emulator, TAdTTYEmulator, and the VT100 emulator, TAdVT100Emulator.

The terminal, on the other hand, has it easy. It merely displays the text that the emulator provides. The TAdTerminal component works with any emulator component: you can link them together either at design time or at run time. If you choose not to use an emulator component, the TAdTerminal will use an internal, hidden, TAdTTYEmulator component anyway. This means that you can get simple terminal functionality just by dropping a TAdTerminal onto your form, but, if you wish a more elaborate emulation, you can drop an emulator component onto the form as well and link them together.

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 that define the communication between an application and the modem, and your application must be able to adapt to those differences.

Async Professional has several components that are designed to simplify the process of adapting to and communicating with different modems. The base modem component is TApdSModem. It combines the most important modem functions into a simple, easy-to-use component.

TAPI, which stands for Telephony Application Programming Interface, 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 well.

APRO's TAPI support is contained within the TApdTapiDevice component.

Protocols

Protocols come in many flavors. The word protocol simply refers to a standard way of doing something. A login/ logout 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 TApdDataPacket component. 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.

Async Professional provides the TApdProtocol component, and its associated status and logging components, to assist in these complex situations. These components greatly simplify 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.

Faxing

Sending a fax essentially consists of binary transmission of data over the phone lines, using a specialized transfer protocol. In addition to sending and receiving fax images, there are other issues that need to be handled as part of the overall faxing process, such as document conversion and viewing.

Only certain modems are capable of sending or receiving fax images. These modems are classified by the language used to set their fax features and perform fax functions. Async Professional supports Class 1, Class 1.0, Class 2 and Class 2.0 fax modems.

Async Professional contains several stand-alone components that help you deal with the details of sending and receiving faxes: TApdSendFax and TApdReceiveFax components, the TApdFaxConverter and TApdFaxUnpacker components, and the TApdFaxViewer and TApdFaxPrinter components.

APRO also includes three components that make it easy to create a distributed fax server system: TApdFaxServer, TApdFaxServerManager and TApdFaxClient.

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 Async Professional user is typically insulated from many of these details—both by Async Professional's encapsulation of the various communication APIs and by the layers of drivers that exist in Windows. With this in mind, it is very possible to write a successful Async Professional 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 Async Professional.

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 peripheral devices. You can attach modems to call other computers. You can attach printers and plotters. You can attach input devices such as data acquisition equipment and laboratory instruments. 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 that someone might be using 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. Should you call it a modem? printer? instrument? remote device? attached device? that thing at the other end of the line? 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. So, 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 UART, short for Universal Asynchronous Receiver/Transmitter. 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)” later in this section.

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 Async Professional 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 Async Professional 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 other limits 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 x386 machine should be able to achieve 9600 baud. Faster x486 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 protocol in Async Professional 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 shown in Table 3.1.

Table 3.1: *Possible parity values*

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.

Here are 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 pOdd and receives a character with pEven, this is a parity error. 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 have it 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 Async Professional, 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 31.

The brain of the serial communications facilities on IBM PCs, PS/2s, and compatibles is a chip called a Universal Asynchronous Receiver/Transmitter (UART). 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 communicates 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 and Com2 and the de facto standard for Com3 and Com4 on IBM PCs and compatibles; and the IBM PS/2 standard for Com1 through Com8. Tables 3.2 and 3.3 show the addresses and interrupts for each standard.

Table 3.2: *IBM PC standard addresses and interrupts*

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

Table 3.3: *IBM PC/2 standard addresses and interrupts*

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 (see 3RDPARTY.HLP for more information). Async Professional 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)

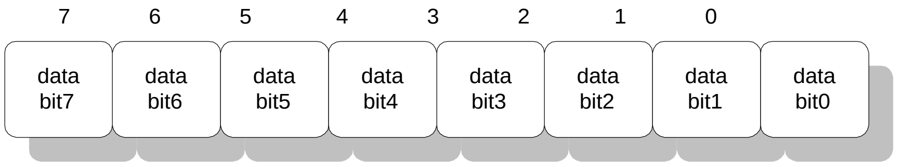


Figure 3.1: Register 0 bit definitions.

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)

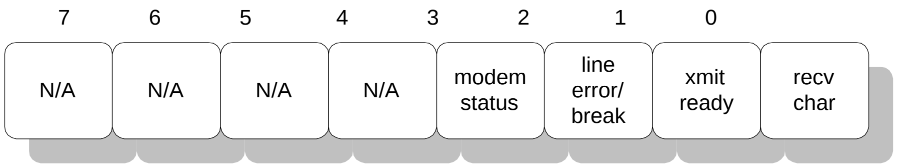


Figure 3.2: Interrupt enable bit definitions.

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)

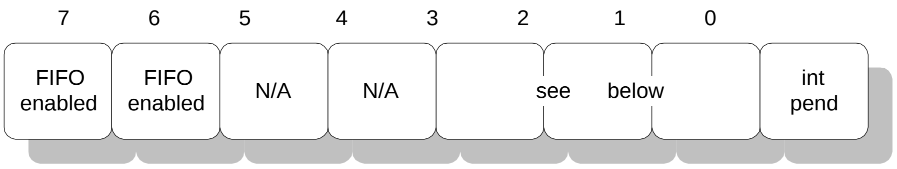


Figure 3.3: Interrupt identification bits.

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.

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. Table 3.4 shows the priority used by the UART and the corresponding bit masks.

Table 3.4: *UART and corresponding bit mask priority*

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 shown in Figure 3.4.

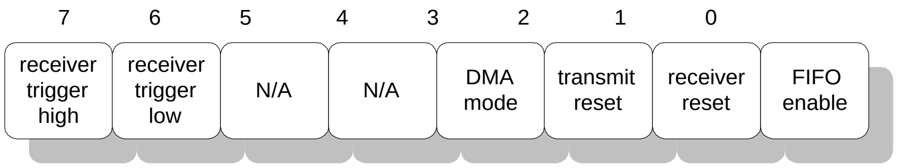


Figure 3.4: FIFO control bit definitions.

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. Table 3.5 shows the possible trigger levels and the corresponding bit values.

Table 3.5: *Possible trigger levels and 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)

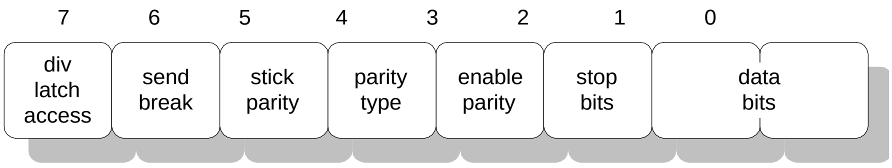


Figure 3.5: Line control bit definitions.

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. Table 3.6 shows how these bits are interpreted.

Table 3.6: *Bit interpretation*

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 are shown in Table 3.7.

Table 3.7: *Parity type*

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)

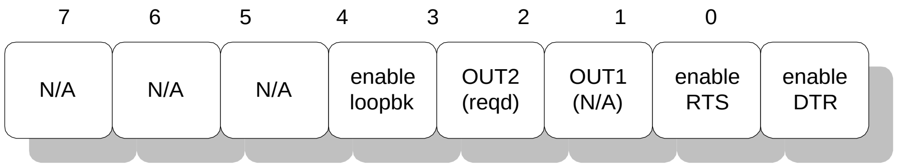


Figure 3.6: Modem control register bit definitions.

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)

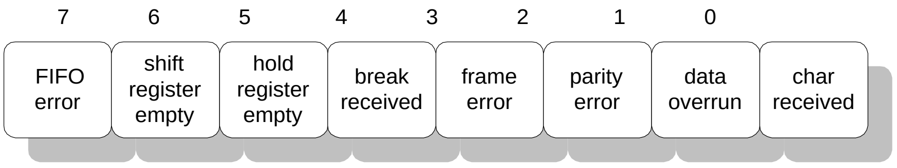


Figure 3.7: Line status bit definitions.

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)

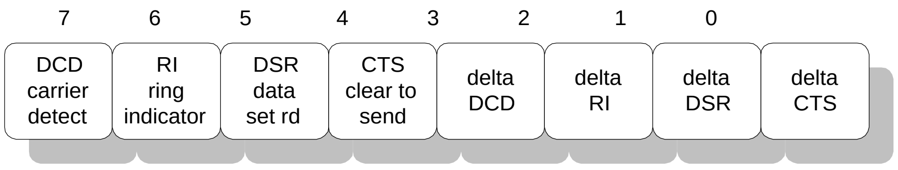


Figure 3.8: Modem control register bit definitions.

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 of these signals are also used by non-modem devices 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 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. Async Professional 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. Notably, when is the flow controlled between the local PC and the local modem and when is the flow controlled between the local PC and the remote PC? These issues are covered in "Modem flow control" on page 35.

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.

Async Professional 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 long to wait for sufficient output buffer space before requesting to transmit some characters. That is, you must handle the possibility that characters won't be transmitted immediately because they are blocked by flow control. You can easily account for this situation by setting an appropriate Async Professional trigger, which will generate an event when a specified amount of output buffer space becomes available.

This covers everything you need to know to use hardware flow control with Async Professional. 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 (shown in Figure 3.9).

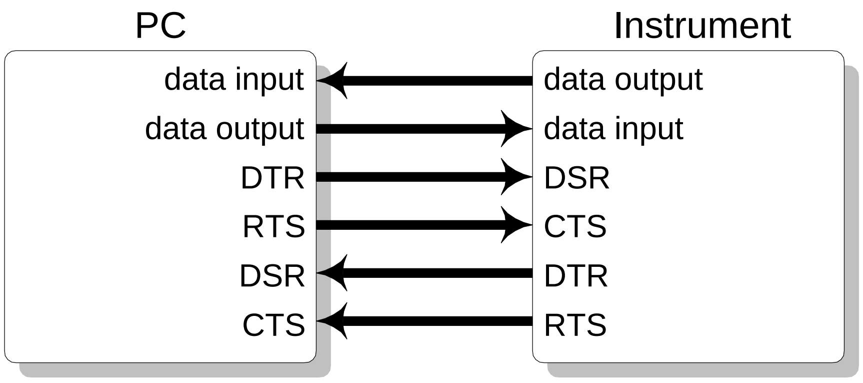


Figure 3.9: Data transmission example.

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. Which raises the question: what do these names mean and what do they have to do with flow control? The complete names are:

- 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 Async Professional 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—it 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 assigning special meaning to two characters—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. All you need to do is enable it.

You should set the buffer full level with more of a safety margin than is necessary for hardware flow control. That's because the remote device may continue to send characters for a brief period after the XOff is sent. This is due to link propagation delays and the response time of the remote device (how long it takes to recognize the XOff and stop transmitting). The same issue applies to the buffer resume level. To keep things going as fast as possible, you shouldn't ever 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 empty.

Conservative flow control levels are 75% of buffer size for the cutoff point and 25% of buffer size for the resume point. Don't be too conservative by setting the cutoff and resume points too close together. That would result in "flow control thrashing" where the transmission of many flow control characters would reduce effective throughput.

Async Professional 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 it an XOn is received. With receive flow control, an XOff is sent when the input buffer hits the buffer full level and an XOn is sent when the input buffer drops below the buffer resume level.

Modem flow control

This discussion focuses on how flow control relates to various modem configurations. Figure 3.10 is used to discuss modem flow control.

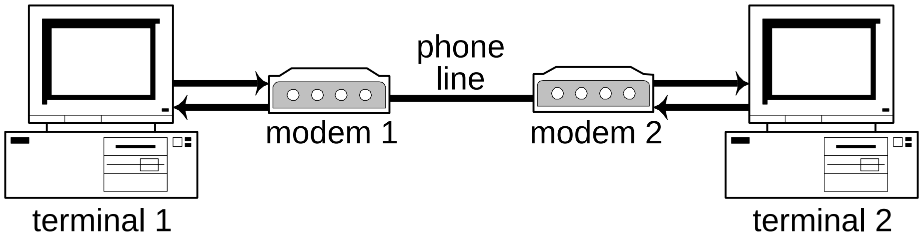


Figure 3.10: Modem flow control.

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 DTEs, data terminal equipment, and the modems are DCEs, data communications equipment.)

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—the flow control is between Terminal1 and Terminal2.

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 ^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, ^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: MNP (for Microcom Network Protocol) and V.42 LAPM (Link Access Procedure for Modems, a standard supported by CCITT). 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.

In short, 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.

You can also use software flow control in these cases. Unlike an unmanaged link, however, the software flow control is between the terminal and modem instead of between the two terminals. But, as we just saw above, 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 some type of flow control—either software or hardware. 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

Serial communications programming under DOS could often be an ordeal, requiring the programmer to work directly with the serial port hardware. In many ways serial communications under Windows is much easier because the Windows communications driver handles the hardware details. Application programs 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.

The 32-bit Windows communications routines send serial port control and I/O requests to a standard communications driver named VCOMM. VCOMM does not directly control the serial port, but relies on "port drivers" to do that. Windows provides built-in port drivers for standard serial port hardware.

The 32-bit Windows communications routines are also thread aware and support multiple concurrent threads. Async Professional takes advantage of threads to minimize CPU usage, while at the same time minimizing response time to communications events.

When a TApdComPort component is opened, it creates three threads—a communications thread and a timer/dispatcher thread to handle incoming data, and an output thread to send data in the background.

The communications thread uses the WIN32 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, WIN32 "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 trigger events that are due. If a trigger event is required, the dispatcher formats and generates an appropriate OnTriggerXxx 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 (i.e., SendMessage). SendMessage assures that the thread that created the window is active before the message is delivered. Therefore OnTriggerXxx, OnModemXxx, OnProtocolXxx, and all other Async Professional 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 (several 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.

Some events can be missed without serious penalty. If other events (OnTriggerAvail, for example) are missed, the application probably won't function properly.

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.

Configuring windows

The configuration of serial ports under Windows is controlled by various INI file settings. These settings can be changed by manually editing the appropriate INI file or by running Control Panel and selecting the Ports icon.

WIN.INI contains the following entries for communications:

```
[Ports]
COM1:=9600,n,8,1,x
COM2:=9600,n,8,1,x
...
```

Entries of this format set the initial line parameters used when a port is opened. These settings are generally unimportant because Async Professional immediately forces the ports it opens to the line parameters specified by the component.

SYSTEM.INI contains the following entries for communications:

```
Com1AutoAssign=-1
Com2AutoAssign=0
Com3AutoAssign=2
...
```

These entries tell Windows how to handle device contention, which occurs when two virtual machines attempt to use the same port. When ComXAutoAssign is set to -1, Windows pops up a dialog box whenever a virtual machine tries to open a port owned by another virtual machine. When set to 0, Windows allows each virtual machine to access the serial port as it wishes. Obviously, this will yield strange and incorrect results if two virtual machines are actually using the same serial port. Values greater than zero indicate a time-out, in seconds, that Windows uses to consider a port available. A value of 2, which is the default, means a virtual machine owns the port for up to 2 seconds beyond its last port access. After that time Windows considers it free for use by other virtual machines. Values greater than 1000 are ignored.

```
...
COM3Base=02E8
COM3Irq=3
...
```

These entries specify the base address and IRQ of a serial port. If they are missing, Windows uses the industry standard addresses and IRQs (see “Universal Asynchronous Receiver/Transmitter (UART)” on page 22 for more information).

```
COMIrqSharing=Off
```

This entry controls Windows IRQ sharing logic for serial ports. Windows sets this to Off unless you are using a Microchannel-bus or EISA-bus machine. You should set this to On only if you are certain that your serial port hardware can share an IRQ.

```
COM1FIFO=1
COM2FIFO=1
...
```

These entries tell Windows whether to enable the receive FIFO (first in, first out) buffer available on 16550 UARTs. Off or 0 means disable the FIFO, On or 1 means enable the FIFO. If the UART doesn’t have a FIFO buffer Windows ignores this setting. Windows always sets the FIFO trigger level to 14 when the FIFO is enabled.

The FIFO buffer can improve throughput and reduce errors due to UART overruns. It should always be used if available.

```
COMBoostTime=2
```

This entry applies to all serial ports. It is the number of milliseconds by which the time slice of the virtual machine is extended after processing a serial port interrupt. The default value is 2. The objective of the boost time is to keep the current virtual machine active to handle the next serial port interrupt. If another interrupt does not occur within the specified number of milliseconds the time-slice expires and the virtual machine is suspended until the next character arrives.

Windows documentation advises caution when changing this value, since too large a value may prevent other virtual machines from receiving adequate execution time.

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 substantially lower than those achievable under DOS.

Because there are so many variables that come into play (machine speed, INI file settings, ill-behaved Windows applications, replaceable port drivers, 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 DOS boxes that are performing background processes. If you are redistributing your application to other users, warn them of lower data throughput when multiple background DOS boxes are active during communications.
- Use a 16550 UART, which has a 16 byte FIFO buffer, and set COMxFIFO=On in SYSTEM.INI.
- Set the COMBoostTime value in SYSTEM.INI above 2.
- Replace COMM.DRV with a higher performance communications driver. See the README.TXT file for a list of replacement drivers.
- 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. See the README.HLP file for a list of intelligent serial port boards.

Event Management

Windows is a message-driven environment and Async Professional is designed to fit into the Windows message system. Async Professional 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 comport component 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 using Winsock.

Once the dispatcher thread 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.

Async Professional uses the term *trigger* for any event that can cause the dispatcher to generate a message. There are four types of triggers:

- Data available trigger: received data is available.
- Data match trigger: a particular character or character string was received.
- Status trigger: a line or modem status event occurred.
- Timer trigger: a timer expired.

Triggers are associated with a particular port component. The TApdComPort component contains a variety of functions for managing triggers. Triggers can be added, activated, modified, and deactivated.

The TApdComPort component registers a message handler with the dispatcher so that it receives each message. The component then categorizes the messages and generates VCL events for a handler in your application.

The triggers and their associated event handlers are described fully in the documentation for the TApdComPort component: Chapter 2 in the Reference Guide.

Device Independence

Async Professional provides for device independence by identifying a set of required core routines and building all other capabilities on top of those. Windows already defines such a set of core routines in the basic Windows API. In fact, Windows already provides for device independence by using installable device drivers. The producer of a new communications device provides a device driver to translate the standard Windows communications API calls into functions specific to the device.

This is huge step forward for applications programmers since it places the burden of supporting odd devices onto the device producer. Unfortunately, the producers don't always do a perfect job. Some are slow to produce Windows drivers and their drivers differ in subtle ways from the standard drivers. For these reasons Async Professional includes its own device independence facility. Hopefully the need to use it will be rare.

Since Windows already defines a standard communications API, Async Professional simply adopts this API for the core routines. Three device layers are supplied in the following units:

- AwWin32: The dlWin32 device layer supports the standard WIN32 API (VCOMM and all vendor supplied replacement port drivers).
- AwWnsock: The dlWinsock device layer supports network and Internet communications using Winsock.
- TAPI: Located in the AwWin32 units is a descendant device layer designed specifically to support TAPI. This device layer differs from its ancestors only by the fact that TAPI services are employed to open and close the serial port, and are automatically activated when the comport component's TapiMode property is set to tmOn.

The dlWin32 and dlWinsock device layers are the only ones available to your application directly. The TAPI device layer is automatically selected when the comport's TapiMode property is tmOn. A TApdWinsockPort can select either the dlWin32 device layer to emulate the TApdComPort or the dlWinsock device layer to connect via Winsock.

Async Professional also provides for custom device layers. Custom device layers are implemented as classes derived either from TApdBaseDispatcher or—if the custom device is similar to one of the devices directly supported by Async Professional—from one of its descendants. TApdBaseDispatcher is declared in AWUSER.PAS. To activate your custom device layer, you'll have to derive a new port component from TApdCustomComPort, overriding the ActivateDeviceLayer method to return an instance of the newly defined dispatcher class.

A custom device layer derived directly from TApdBaseDispatcher must override the following methods in order to implement support for a particular hardware device:

```
function OpenCom(
  ComName: PChar; InQueue, OutQueue : Cardinal) : Integer;
  virtual; abstract;
function CloseCom : Integer; virtual; abstract;
function EscapeComFunction(
  Func : Integer) : LongInt; virtual; abstract;
function FlushCom(Queue : Integer) : Integer; virtual; abstract;
function GetComError(
  var Stat : TComStat) : Integer; virtual; abstract;
function GetComEventMask(
  EvtMask : Integer) : Cardinal; virtual; abstract;
function GetComState(var DCB: TDCB): Integer; virtual; abstract;
function ReadCom(
  Buf : PChar; Size: Integer) : Integer; virtual; abstract;
function SetComState(
  var DCB : TDCB) : Integer; virtual; abstract;
function WriteCom(
  Buf : PChar; Size: Integer) : Integer; virtual; abstract;
function SetupCom(
  InSize, OutSize : Integer) : Boolean; virtual; abstract;

function WaitComEvent(var EvtMask : DWORD;
  lpOverlapped : POverlapped) : Boolean; virtual; abstract;
```

Unfortunately, describing how to write a new device layer is beyond the scope of this manual. If you find the need to do so, please study the supplied device layers in the Async Professional source code.

Chapter 4: Overviews and Troubleshooting Sessions

Communications programming is such an intricate undertaking—and Async Professional 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 APRO users: picking the right modem for your next project and issues regarding voice modems and TAPI voice support. A third overview offers a general discussion of the new Fax Server Components and how they inter-relate and a fourth overview offers a collection of tips and techniques for debugging Windows communications programs and diagnosing common hardware difficulties.

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

4

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. Here's 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. Sure, 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—the more you learn about 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, the telephone service called DSL in the United States labels the device use 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 APRO's knowledge). Just keep in mind that it's a complex process, and things can easily go wrong.

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 "smarts" of the modem 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. Most currently do not work in alternate operating systems such as Linux.
- The software drivers that support these modems are proprietary, and cannot be used directly without license. For this reason, APRO 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 website also has additional information about modems, as well as useful links to other modem websites.

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 it's not an 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.

Don’t “chase the latest technology”

Modem makers often race to hit the market first with a new feature in order to gain market share. It often takes a little while to get 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. APRO currently supports Class 1, Class 1.0, Class 2, and Class 2.0 faxing.

The fax class defines the communication between APRO and the modem. Class 2 and 2.0 modems handle more of the work themselves, with only minimal communication required between APRO and the modem. This is usually an advantage, because the modem handles most of the work (similar to the advantages mentioned above in the software modem discussion). A potential disadvantage with Class 2 & 2.0 faxing is that APRO is “kept in the dark” during large portions of the fax session. Obviously, this isn’t a problem if the fax session goes well—but if there are problems during the session, APRO has limited ability to detect, correct, and log the situation.

With Class 1 and Class 1.0 modems, APRO is intimately involved with nearly all aspects of the fax session. As stated earlier, this loads the system more but if things go wrong, there are more options available to correct the situation.

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. This only makes sense. If you’re going to be buying several hundred modems, make sure you’re getting modems that will work well in your situation.

Overview: Using the Fax Server Components

This topic shows what the TApdFaxServer, TApdFaxServerManager and TApdFaxClient components do and how they interrelate.

Async Professional has always had components to send, receive, and manipulate faxes: TApdSendFax and TApdReceiveFax. These components are discussed fully in other sections of the Async Professional documentation. The new Fax Server Components: TApdFaxServer, TApdFaxServerManager and TApdFaxClient represent a considerable design-shift from other APRO faxing components.

To clarify the inter-dependencies of the Fax Server Components, the following discussion breaks the Fax Server paradigm up into manageable pieces: device configuration, fax reception, job creation and scheduling and fax transmission. It will be helpful if you have a general understanding of the purpose of and process behind a fax server before we get into details.

What is a fax server?

A fax server is a group of functions that send, receive and manipulate faxes. It should also handle scheduling faxes, and sending faxes to multiple recipients. Async Professional offers you a choice: you can use the separate TApdSendFax and TApdReceiveFax components, work up some form of queuing system and a way to send the same fax to different people at different times or you can let the Fax Server Components do much of this for you.

The fax job

The key to the Fax Server Components is the concept of the fax job. The Fax Server Components use fax jobs to handle scheduling faxes and multiple recipients. Fax jobs are basically regular APF files with additional headers. There is a job header, which contains information that applies to all recipients of the fax; such as who is sending the fax, the friendly name of the fax, and pointers to cover page text and the actual APF data. Following the job header is a recipient header for each recipient. The recipient header has information specific to each recipient; such as the phone number, header line information, and scheduling information. The fax job file also can hold the text of a cover page document, allowing you to customize the cover page for each recipient using replaceable tags in the text.

Who does what?

The TApdFaxServer, TApdFaxServerManager, and TApdFaxClient work together. Each component has it’s own purpose, and they all need to be working properly for the process to function as designed.

The TApdFaxServer component is the only component out of the three that communicates with the physical faxmodem. This component is responsible for receiving and sending faxes.

The TApdFaxServerManager component manages the TApdFaxServer for sending faxes. The TApdFaxServer component asks the TApdFaxServerManager component for fax jobs that are ready to be sent, and the TApdFaxServerManager component returns the job to handle. This component does not communicate with the modem; it just gives the TApdFaxServer faxes to send.

The TApdFaxClient component's primary task is to create fax jobs. The fax job format is a modification of the regular Async Professional APF format, and the TApdFaxClient handles making those modifications.

Device configuration

The TApdFaxServer component makes configuring a device to use for faxing a straight forward process. Drop a TApdComPort on the form, and set the ComNumber property to the serial port number of the faxmodem you want to use. If you want to use TAPI, drop a TApdTapiDevice component on the form also. Finally, drop a TApdFaxServer component on the form, and set the ComPort property to the TApdComPort component we just dropped on the form, and the TapiDevice property to the TApdTapiDevice component. That's all there is to it, when the TApdFaxServer needs access to the port, it will open it. Once the port is no longer needed, the port will be closed.

OK, it's a bit more complicated than this, but not much more. If the TApdComPort.TapiMode property is tmOn and the TApdTapiDevice.SelectedDevice is pointing to a valid TAPI device, the port will be opened through TAPI. If the SelectedDevice is not valid or an empty string, the port will be opened through the TApdComPort. If the TApdComPort.TapiMode is tmOff, then the TApdTapiDevice is ignored.

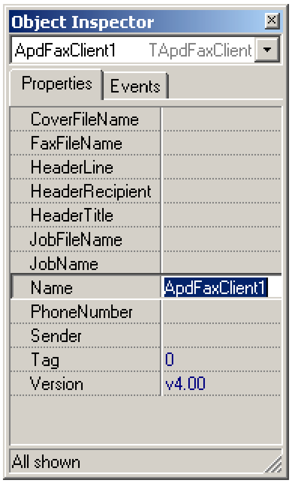
Fax reception

The TApdFaxServer component handles receiving faxes and the process is surprisingly easy. The TApdFaxServer.Monitoring property controls whether or not incoming faxes should be received. Set this property to True to receive incoming faxes; set it to False to stop receiving faxes. When incoming faxes are received, they will be named according to the FaxNameMode property, and saved in the directory specified by the DestinationDirectory property. The OnFaxServerFinish event will fire for each fax that is received successfully; the OnFaxServerFatalFinish event will fire for each fax that is not successfully received.

Job creation and scheduling

Fax jobs, in our context, are files that contain fax data to send (similar to our APF format) and have additional headers that have information about where the job goes and when to send it. The TApdFaxClient creates these jobs. Since there is a lot of information required to schedule a fax, this step is rather detailed.

Most of the properties of the TApdFaxClient component are included in the APJ file that it will create. Figure 4.1 shows the Object Inspector.



CoverFileName	
FaxFileName	
HeaderLine	
HeaderRecipient	
HeaderTitle	
JobFileName	
JobName	
Name	ApdFaxClient1
PhoneNumber	
Sender	
Tag	0
Version	v4.00

Figure 4.1: The Object Inspector.

Enter the name of the text cover file in the CoverFileName property, and the name of the APF fax file that you want to send in the FaxFileName property. When the job is created, these two files will be loaded and embedded in the APJ file.

The JobFileName property is the name of the APJ that will be created. JobName is a short description of the fax job. Sender is the name of the person or machine that is submitting the fax.

The other properties are specific to the recipient of the fax. HeaderLine, HeaderRecipient, and HeaderTitle all relate to the line of text that is at the top of the fax when it is sent. PhoneNumber is the number of the fax machine to which this fax will be sent.

There is one property that is not accessible at design time, or through the Object Inspector: the scheduled date and time that the fax should be sent. Set the TApdFaxClient.SchedDT property to the TDateTime that the fax should start. If the property is not set, then it is assumed that you want to send the job immediately.

Once all of this information is entered, call the TApdFaxClient.MakeFaxJob method, and the job file will be created.

To add additional recipients to the fax job, so the identical cover page and fax file is sent to different machines, set the PhoneNumber, HeaderTitle, HeaderLine, HeaderRecipient, and SchedDT properties to what you need for the new recipient; and call the AddRecipient method.

After all this is done, just copy the APJ to the directory being monitored by the TApdFaxServerManager, and it will get picked up and faxed. If you have a single recipient for the fax job, you can save a step by specifying the monitored directory for the FaxJobFileName property, and you won't have to copy it there later.

Fax transmission

As we've already mentioned, submitting a fax job for transmission consists of creating the fax job and putting it where the TApdFaxServerManager component can see it. The process is just a little more complicated than that, but that is the general idea.

Set up the TApdComPort and TApdTapiDevice components, as you did earlier for fax reception (if you are already set up to receive, you don't need to do anything else with these components).

The TApdFaxServer component sends faxes when the TApdFaxServerManager component gives it a fax to send. To find a fax to send, the TApdFaxServerManager monitors a specific directory. Drop a TApdFaxServerManager component on the form, and set the MonitoredDir property to the directory to look at. The TApdFaxServerManager component is now configured to provide fax jobs to the TApdFaxServer component.

The TApdFaxServer component will ask the TApdFaxServerManager component for fax jobs on an interval defined by the SendQueryInterval property (in seconds). Since the TApdFaxServerManager component does a lot of disk access when asked for a job, it's a good idea to keep SendQueryInterval relatively large, 30 seconds should be the minimum. When SendQueryInterval is a non-zero value, the TApdFaxServer component starts requesting fax jobs from the TApdFaxServerManager component, and sends them when they are ready.

Advanced topics

What we've talked about here is a fairly simple fax server. You can easily expand on this by having multiple TApdFaxClient components distributed across a workgroup, each one submitting fax jobs to a TApdFaxServerManager component. Or, you can have several TApdFaxServer components, each connected to a separate fax modem, all looking at a single TApdFaxServerManager component. Or, you can have multiple TApdFaxServer components looking at several TApdFaxServerManager components. One feature that is requested fairly often concerns the storage of the fax job files (APJs). Copying APJ files across the network from a client to the directory being monitored by the TApdFaxServerManager is often cumbersome, posting an APJ as a BLOB field in a database, or even storing the information in a different structure, might be better suited to your environment. The TApdFaxServerManager.OnCustomGetJob method can be used to support an alternate job storage medium. This event is generated when a TApdFaxServer component requests a job (based on the SendQueryInterval timer). With a little work you could look up the next fax in a database, create a temporary fax job file, and then let the TApdFaxServer fax that.

Related examples

FXSRVR.DPR

FXCLIENT.DPR

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.
- Async Professional negotiates for TAPI version 1.4, which is backwards compatible in all later TAPI versions as of this writing. Async Professional implements a few TAPI 2.0 functions, such as retrieving the port number associated with the device. Async Professional 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. Table 4.1 shows the availability of Unimodem on various operating systems.

Table 4.1: *Unimodem availability by operating system*

	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

Related examples

EXFAXOD.DPR - Example of Fax-on-Demand (Faxes sent on separate call)

EXFODR.DPR - Example of Fax-on-Demand (Faxes received on same call)

EXFODS.DPR - Example of Fax-on-Demand (Faxes sent on same call)

EXRECORD.DPR - Example of Voice Recording on a call (simple VoiceMail)

EXVOICE.DPR - Example of DTMF Detection

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 such as one of our example programs. If that 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 the DOS libraries Async Professional or Async Professional for C/C++, you may recall cautions about using debuggers with communications programs. 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. And 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 the Async Professional debugging tools

Async Professional has several built-in features that aid in the debugging process. The simplest is the tracing facility. It provides a character-by-character audit report of all the data transmitted or received by your program. Tracing is particularly useful when your program advances to some point and then starts misbehaving. After a few minutes of study, a trace of such a program run will generally lead you to the problem area. See "Tracing" in Chapter 2 of the Reference Guide for more information.

Async Professional provides another auditing tool called dispatch logging, which works at a much lower level than tracing. Dispatch logging provides an exact chronology (with millisecond timestamps) of all events processed by the internal dispatcher, as well as state changes in associated APRO components. 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" in Chapter 2 of the Reference Guide for more information on this facility.

Getting technical support

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

Technical support is always a tough job and throwing communications problems into the equation makes the task even tougher. 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 some of the Async Professional built-in debugging tools, Tracing and Dispatch Logging, to try to find the problem.

Finally, any Async Professional 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 all the built-in debugging tools 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 project 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 trace file or event 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 Async Professional. They are organized in a question and answer format.

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

Probably 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; 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 things to go wrong.

When it's not working there are a number of places to start looking, depending on your particular situation.

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 "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.
- See the sections below on Com 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. Such configuration issues are discussed in the next few sections.

Resources:

In order to function, 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 shown in Table 4.1.

Table 4.1: *IBM PC serial port resource assignments*

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, and 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). IRQ5 (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.

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 generally these should 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 at 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, but this is not guaranteed; and some software has problems dealing with ports with peculiar settings.

What do I do if something isn't working?

If you're having problems with serial ports with hardware set IRQ and 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 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 then 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 to attach the cables to the main board are often small and are easy to not have seated properly, 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.

Another possibility is that Windows itself has conflicting or erroneous settings for the ports.

How do I fix the Windows settings?

In Windows 9x/ME/2000, 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 box, the "Resources" tab has the settings for the IRQ and Address Range. You can change the IRQ and base address settings for these 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 in Windows 98/ME and 2000.

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 then 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 above sections on "The Modem Isn't Working…" and "The serial ports aren't responding…" for diagnostics.

- If you're using Windows 95 you may need to obtain the updated TAPI (UNIMODEM/V) software from Microsoft; Windows 98 and NT 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", and since 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 brought 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 a WinModem's 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 leOverrun 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 dispatcher 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 acknowledgements. Flow control is required to prevent the sender from overflowing the modem or the receiver. And remember, flow control must be enabled at four places: your software, your modem, the remote software, and the remote modem. See page 31 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 shows some common questions and answers 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 TApdComPort component?

To open a port, the TApdComPort component tries to activate the serial port of the computer identified in the TApdComPort.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, the EOpenComm general exception is raised. You can trap that exception and bring up the default Comport Selection Dialog by setting the ComNumber property to 0 and then opening the port again.

The following example shows one way of handling this:

```
procedure TForm1.OpenBtnClick(Sender : TObject);
begin
  while True do
    try
      ApdComPort1.Open := True;
      Break;
    except
      on EOpenComm do
        ApdComPort1.ComNumber := 0;
    end
end;
```

Why won’t my device 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 TApdComPort.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 resetting your system’s hardware. Use the Control Panel | System applet to resolve the conflict.

Why won’t my device 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.

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 TApdComPort.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, whenever I enter characters into the TAdTerminal component, are they doubled?

Turn off the Echo mode of the modem, or set the TAdTerminal.HalfDuplex 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"#13 (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. The following example demonstrates one method of doing this by using the DelayTicks function:

```
uses
  OoMisc;

procedure TForm1.DialBtnClick(Sender : TObject);
begin
  ApdComPort1.Output := 'AT&F'#13;
  DelayTicks(36, True);
  ApdComPort1.Output := 'ATDT 260 9726'#13;
end;
```

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 faxing or sending files.

Also, most Winmodem, 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 shows some common questions and answers 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. You can get the text of the code by passing it to the ErrorMsg method. "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?

If you are not already using one, drop a TApdProtocolStatus component on the form; 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"#13 to let the receiver know something is coming. If the receiver is not watching for "rz"#13, your transfer will eventually time-out.

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

The TApdProtocol.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 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.

Troubleshooting a Fax Session

This topic shows some common questions and answers for troubleshooting a fax session.

Under normal circumstances, faxing with Async Professional is reliable, but you have been having a hard time getting good results.

A fax connection is very similar to a data connection and a file transfer, but the faxing protocol is more standardized throughout the communications industry. Refer to the “Troubleshooting a Connection Session” and “Troubleshooting a File Transfer” for details not covered here.

Why can I dial a remote fax machine, but am immediately disconnected?

Set the TApdSendFax/TApdReceiveFax.FaxClass to fcClass1. Fax Class 1 is almost universally supported, being the base fax transfer standard. Some of the more popular fax modems and stand-alone fax machines do not support alpha characters in the Station ID, so you can try using numbers only. Other settings that may resolve the problem are:

- Change the ApdSendFax/ApdReceiveFax.ModemInitString to &H3&I2&R2S7=90.
- Add S36=0 to the ModemInitString.
- Lower the BPS to 7200, 4800 or 2400.
- Download the latest drivers for your device from the manufacturer.

Why are the faxes missing sections or have spots on them?

This is a result of poor phone line quality. Place the call again. Each time a call is placed the phone company routes it differently.

Why are my received faxes elongated or shortened when I view or print them?

The TApdFaxViewer.AutoScaleMode determines whether or not the fax being viewed is automatically scaled. If it is asNone, no scaling is done, If it is asDoubleHeight, the fax height is stretched to twice its original size. If it is asHalfWidth, the fax width is compressed to half its original size. The TApdFaxPrinter.PrintScale determines whether the fax is scaled to fit the printed page or if it keeps its original dimensions. If the printed copy of the fax seems out of proportion, set PrintScale to psNone.

Why do I get “Bad response from modem” errors before dialing?

This is, by far, the most common error, especially when TAPI is used to configure the modem. There are two main causes of the problem: baud mismatch, or fax class query confusion. If you have a baud mismatch, TAPI is configuring the modem for a different baud than is required for faxing, and the dispatcher log will show garbled random entries. To correct this problem, set the ComPort.Baud property to 33,600 before opening the port, and then set it back to 19,200 in the OnTapiPortOpen event. You could also send “AT”#13 to the modem, wait for about 1 second for a reply using DelayTicks, then call the StartXxx fax method.

If you are suffering from the fax class query confusion problem, the dispatcher log will show a readcom of “ERROR” after APRO sends the “AT+FCLASS=” command. The solution here is to set the FaxClass property to a supported fax class before starting the fax.

Chapter 5: Tutorials

Async Professional is unlike most other component libraries. To use the components successfully, developers must handle different system configurations, system settings, telephone lines, and brands of modems. This Tutorial section is designed to assist you in getting started with the basic tasks of Async Professional. There are many topics, but Async Professional covers a lot of programming ground.

Whether you are just starting with Async Professional or are developing advanced applications, start with the basic tasks and move forward to the topics that apply to your project. Keep in mind that these topics are designed to give you a head start and give you ideas. They are not intended to be the only solutions to particular needs. You may need to adapt them a bit to fit your situation.

In each topic, you will find the components that are required for the particular task, prerequisite topics, other components that can help you add the finishing touches, detailed descriptions of the steps involved, and a list of related example projects.

Setting Up a Comport

This topic shows how to set up a serial port for serial communications.

The TApdComPort component is the basic building block of the Async Professional component library. This component provides access to the physical serial port/modem/device and, therefore, to the characters entering or leaving the port. Since most of the Async Professional components rely on this component, it is essential that it is configured correctly. Fortunately, the default properties work in most applications.

Keep in mind that this topic, along with all other topics involving the TApdComPort may also apply to the TApdWinsockPort—particularly when the TApdWinsockPort is not operating in Winsock mode.

Required components

TApdComPort

Prerequisite topics

None

Related components

None

What to do

The most important property of this component is the ComNumber property, which associates the component with the desired serial port. The ComNumber property is an Integer value. If the device you wish to use is attached to Com2, ComNumber should be 2; if the device is on Com 4, ComNumber should be 4. If ComNumber is 0, (the default) then a port selection dialog is shown when the port is opened. This allows the end-user to select a port at run time. If you don’t want the user to have this level of control, then assign the appropriate value at design time or run time before the port is opened.

Next, you need to know the line settings of the device with which you want to communicate. Most devices use N81, which is No Parity, 8 Data bits, and 1 Stop Bit. This is the default configuration of the TApdComPort component. If these values are not correct, change the settings of the Parity, DataBits, and StopBits properties respectively to the proper values.

The next step is to control when the port it opened with the Open property. When Open is set to True, the port is initialized and opened. When Open is set to False, the port is closed. The AutoOpen property determines if the port automatically opens when needed by another Async Professional component.

Related examples

EXCOM.DPR

Sending Characters

This topic shows how to send characters out through the serial port.

One of the basic tasks of serial communication is sending data through the serial port. All data that can be transmitted through the port is in the form of a sequence of characters. The easiest way to transmit text from one system to another is to drop a TAdTerminal component on the form. This component handles sending, receiving, and displaying of characters for you. If you need greater control, or if you want to send characters programmatically, then you should use the comport's Output property.

Required components

TApdComPort

Prerequisite topics

"Setting Up a Comport" on page 74.

Related components

TAdTerminal

What to do

Drop a TApdComPort component on the form and set it up for your system (see "Setting Up a Comport" on page 74).

In your code, where you want to send the characters, assign the characters to be sent to the TApdComPort.Output property. All the low-level tasks are automatically handled and the characters are sent.

The following example sends "Hello world" to the remote system as a result of a button click:

```
procedure TForm1.Button1Click(Sender : TObject);
begin
  ApdComPort1.Output := 'Hello world';
end;
```

If you need to transmit non-character data, such as numerical data, that data must be converted to a character before sending. For example, if you need to send the number 6 to a piece of equipment, you would send the character with an ordinal value of 6 (which happens to be the ACK character). For example:

```
...
  ApdComPort1.Output := #6
...
```

Related examples

EXCOM.DPR

Receiving Characters

This topic shows how to receive characters through the serial port.

A typical communications session includes receiving information that is coming in through the serial port. The easiest way to receive incoming characters is to drop a TAdTerminal component on the form and let it handle sending and receiving the characters. However, this method does not give you the level of control that might be needed to programmatically handle incoming data. In those cases, you need to use the TApdComPort.OnTriggerAvail event, which is designed just for this purpose.

Required components

TApdComPort

Prerequisite topics

"Setting Up a Comport" on page 74.

Related components

TAdTerminal

What to do

Whenever characters are received at the serial port, the Windows communications driver notifies Async Professional. When the dispatcher receives the notification, the TApdComPort.OnTriggerAvail event fires. The number of available characters is passed in the Count parameter of the event handler. Inside this event your code should retrieve exactly Count characters with the TApdComPort.GetChar method.

Due to the event-driven nature of the Windows communications drivers, the number of available characters fluctuates during the program's execution. In light of this, your code should retrieve all Count characters into a buffer, then process the buffer. If necessary, the code can check individual characters, or the entire buffer, as characters are retrieved from the dispatcher buffer (using the GetChar method).

The following example retrieves characters from the dispatcher buffer, puts them into a buffer, and changes the caption when “Hello” is received:

```
var
  Buffer : string[255];
  BufferIndex : Integer;

procedure TForm1.ApdComPort1TriggerAvail(
  CP : TObject; Count : Word);
var
  I : Word;
  C : Char;
begin
  for I := 1 to Count do begin
    C := ApdComPort1.GetChar;
    if C = #7 then
      MessageBeep(0)
    else if C in [#32..#126] then begin
      Buffer := Buffer + C;
      if (Pos('Hello', Buffer) > 0) then
        Caption := 'Got Hello';
    end;
  end;
end;
```

Related examples

EXCOM.DPR

Detecting a Specific String in the Data Stream

This topic shows how to detect a string, individual character, or sequence of characters that have been received at the serial port.

Detecting specific strings or characters in the incoming data stream is an integral part of most communications applications. Whether you are automating a logon session, getting data from a monitoring instrument, or creating a custom file transfer protocol, your code needs to detect specific strings or individual characters and handle them. There are several ways to do this: monitor all incoming characters in the OnTriggerAvail event and parse the accumulated characters, use DataTriggers, or use the TApdDataPacket component. The TApdDataPacket component is the easiest to use, since the majority of the work can be done at design time.

Required components

TApdComPort

TApdDataPacket

Prerequisite topics

“Setting Up a Comport” on page 74.

“Sending Characters” on page 76.

Related components

None

What to do

Drop a TApdDataPacket component on the form and right-click it. Select the Edit Properties item to display the TApdDataPacket property editor (you can also edit the properties in the Object Inspector, but this property editor makes things much simpler). Enter the string or character for which the program must look in the “When this string is received” edit box. When that string or character is detected, the OnPacket and/or OnStringPacket events fire, depending on which events are declared. The OnPacket event passes a pointer to the packet data and the OnStringPacket event passes the data in a string. Either event can be used to notify your application that the data was detected.

The string can include both control and alpha-numeric characters. When you enter an alpha-numeric string, the property editor works like any other string editor. When you enter a control character, enter '#' character followed by the decimal value of the character or the '^' character followed by an alpha-character. If you need to mix alpha-numeric and control characters, put the alpha-numeric characters in single quotes. For example:

Searching for an alpha-numeric string

In the property editor, enter the string without quotes. In your code, assign the string inside single quotes to the property.

```
ApdDataPacket1.StartString := 'Hello';
```

Searching for a control character/characters

In the property editor enter “#X”, where X is the decimal value of the character, or “^X” where X is the control-key combination. Do not use quotes. In your code, the same rule applies.

```
ApdDataPacket1.StartString := #2;
ApdDataPacket1.EndString := ^B;
```

Searching for an alpha-numeric string with control characters

In the property editor enter the string within single quotes and the control characters without quotes. Do the same in your code.

```
ApdDataPacket1.StartString := 'Hello'#2;
ApdDataPacket1.StartString := 'Hello'#13'World'#13;
```

Here are the steps for a quick example that detects when “OK” is received:

Drop the following components on the form: TApdComPort, TAdTerminal, TApdDataPacket, and TButton. Right-click the TApdDataPacket component and enter “OK” (without the quotes) in the “Received string” edit control. Double click the TApdDataPacket component to create the OnPacket event. Add the following line:

```
Caption := 'Received OK';
```

Now, double-click the button to create its OnClick event handler. In this example, we'll send the “ATZ” command to the modem (the default initialization command for most modems). Add the following line to the OnClick event handler:

```
ApdComPort1.Output := 'ATZ'#13;
```

Compile and run the application. Select the comport to which your modem is attached and then click the button. The modem is sent the “ATZ” command and should respond with “OK”. If so, the OnPacket event fires and the form's caption changes to “Received OK”.

Related examples

QRYMDM.DPR

EXWPACKT.DPR

Detecting a Packet

This topic shows how to detect packets (i.e., specific sequences of characters).

Many times, the data you're trying to collect during a communications session is in the form of a data packet. A data packet can be defined as any sequence of characters. It can be a specific character/string, be bracketed between two known characters/strings with the data having an unknown length, start with a known character/string and have a known length, or end with a known character/string and have a known length. These types are all addressed in this topic.

Required components

TApdComPort

TApdDataPacket

Prerequisite topics

"Setting Up a Comport" on page 74.

"Sending Characters" on page 76.

"Detecting a Specific String in the Data Stream" on page 80.

Related components

None

What to do

The TApdDataPacket component can handle several forms of packet structures. There are only a few possible structures that would be of any use and they are described below. If your application is trying to monitor, filter, or process incoming data, that data will be in one of the following forms:

1. A specific character/string. For example, the Zmodem auto-receive message would look something like: 'rz'#13. This is the most common usage and can be thought of as an enhanced DataTrigger from previous versions of Async Professional. The packet is defined by its own StartString, so it is easy to detect. Set StartCond to scString and StartString to 'rz'#13. When this string is detected the OnPacket and OnStringPacket events are fired.

2. A bracketed packet. This is when the data characters are bracketed between two known characters (e.g., <STX>‘XXXX’<ETX>). In this case, the packet starts with the STX character (#2) and ends with the ETX character (#3). “XXXX” represents an unknown number of characters. To detect this type of packet, set the StartCond to scString, StartString to “#2”, EndCond to [ecString] and EndString to “#3” (without the quotes). When StartString is detected, all following characters are saved in the TApdDataPacket component’s buffer. When EndString is detected, the OnPacket and OnStringPacket events are fired. The OnPacket event passes a pointer to the bracketed characters while the OnStringPacket passes a string containing the bracketed characters.
3. A known start character or string followed by data of known length. An example might be <STX>”XXXX.” Here, the packet starts with the STX character (#2) and contains a sequence of characters of four characters (represented by ‘X’). To detect this packet, set the StartCond to scString, StartString to “#2” (without the quotes), EndCond to [ecPacketSize] and PacketSize to the length of the data packet. When StartString is detected, the following PacketSize characters are saved in the TApdDataPacket component’s buffer. Once the specified number of characters is received, the OnPacket event, or the OnStringPacket event, or both, fire.
4. A known number of characters followed by a terminating character, e.g., “XXXX”<ETX>. This packet can start at any time in the communications session, and is defined by the terminating character or string. To detect this type of packet, set StartCond to [scAnyData], EndCond to [ecString] and EndString to “#3” (without the quotes). When the EndString character or string is detected, the OnPacket and OnStringPacket events fire. Your code must then extract the expected number of characters from the buffer.

   Since all characters preceding the EndString are placed in the TApdDataPacket component’s buffer, this type of packet can be error-prone, especially if the packet for which you are looking follows other data or information. In this case, you must enable the TApdDataPacket component by setting TApdDataPacket.Enabled property to True just prior to when the data is expected.

Related examples

EXWPACKT.DPR

QRYMDM.DPR

Selecting and Configuring a Modem

This topic shows how to select and configure a modem that is attached to a comport.

Configuring a modem is a very important process in a communications application. The modem has to be properly configured for it to accept the commands and return the appropriate response strings, as well as configuring the protocol negotiations.

Usually, the services provided by TAPI are adequate. TAPI will configure the modem and establish a connection based on the modem's property dialog settings. Under some requirements, TAPI does not provide the flexibility needed. Consider a connection where non-standard line settings are required, such as 7 data bits instead of 8. TAPI, by default, will configure the line settings for 8N1. To use something other than that, the modem properties dialog will need to be displayed, the settings changed, then the connection can be made. The TAdModem component provides an alternate technique which can be implemented without user interaction.

Required components

TApdComPort

TAdModem

Prerequisite topics

"Setting Up a Comport" on page 74.

Related components

TApdTapiDevice

What to do

The TAdModem provides access to a few thousand modem configuration structures, which define the commands and responses to configure an equal number of modems. The modem configuration structures are accessed through the TApdLibModem class, which is discussed in the next tutorial, for our purposes now we just need to know it is there, but we do not need to know any particulars about it.

The TAdModem has properties, events and methods that are similar to the TApdTapiDevice. The SelectedDevice property contains information about the name, manufacturer which file contains the modem configuration structure. The SelectDevice method invokes a modem selection dialog box where a modem can be selected. The

ConfigAndOpen method configures the modem without establishing a connection. The Dial and AutoAnswer methods configure the modem then establish a connection by either dialing or answering. The CancelCall method cancels whatever the TAdModem is doing.

One of the benefits of the TAdModem over the TApdTapiDevice is that the serial port is accessible before the connection is made. When dialing or answering with TAPI, the serial port is valid only after the connection has been made. Since the connection is already established, the line settings have already been negotiated, and that is too late for most circumstances. The serial port is accessible at practically any time with the TAdModem component, so the line settings can be changed before the connection is established.

Consider our non-standard line settings from above. All you need to do is set the TApdComPort properties according to the line settings you need, then use the Dial or AutoAnswer methods of the TAdModem class to establish the connection.

Related examples

EXMDM.DPR

LibModem

The TAdModem uses LibModem to access the modemcap modem database. The modemcap modem database is a TurboPower initiative to provide modem configuration information outside of TAPI. Modemcap consists of XML documents, which contain modem configuration and operational commands and responses. The format is very similar to the modem's .INF files, which TAPI uses for the same purposes.

LibModem (through the TApdLibModem component) provides access to modemcap to add, delete and modify the modem definitions contained in modemcap.

Required components

TApdLibModem

Prerequisite topics

None

Related components

TAdModem

TApdTapiDevice

What to do

The modemcap database consists of modem configuration information, which was extracted from the Windows modem INF database. This creates a very large database, and it is likely that you will not want to distribute the default database. There are essentially three alternatives, either distribute a subset of the database, or distribute a shell and let the user add modem information, or a combination of these.

Accessing modemcap

The modemcap database is accessed through the TApdLibModem. The TAdModem component provides limited modemcap support, primarily for selecting and retrieving the modem details. TApdLibModem provides methods for adding new modems, deleting and editing existing modems, and for creating new modem detail files. To illustrate how to use the TApdLibModem class, we will create a program that will extract modem details from modemcap, put them in a single modem detail file, then select the details from the new, consolidated file. To do this, create a new project and keep reading.

You will need to have an instance of the TApdLibModem class, so add "AdLibMdm" to your uses clause and the following to the form's declaration:

```
type
  TForm1 = class(TForm)
...
  public
    LibModem : TApdLibModem;
  end;
```

Create the form's OnCreate event handler and add the following to create the LibModem instance:

```
...
  LibModem := TApdLibModem.Create(Self);
...
```

Free LibModem when the form is freed also, so create the OnDestroy event handler and free the class from there.

Drop a TListBox component on the form and set the Name property to lbxManufacturers. Drop a TButton component on the form and create the OnClick event handler. In the OnClick method, use the GetModemRecords method of the TApdLibModem to retrieve a collection of all of the modems, and then add the modem manufacturer names to the list box. GetModemRecords returns a TApdLmModemCollection, which is a collection of modem records. Add the following to your form's declaration under the LibModem added earlier:

```
...
    ModemColl : TApdLmModemCollection;
...
```

Create the button's OnClick event handler and add the following. Since we want this list box to contain the modem manufacturers, there is code to prevent duplicates.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I : Integer;
  Manufacturer : string;
begin
  LibModem.LibModemPath := {add the path to modemcap};
  ModemColl := LibModem.GetModemRecords;
  lbxManufacturers.Items.Clear;
  for I := 0 to pred(ModemColl.Count) do begin
    Manufacturer := ModemColl[I].Manufacturer;
    if lbxManufacturers.Items.IndexOf(Manufacturer) = -1 then
      lbxManufacturers.Items.Add(Manufacturer);
  end;
end;
```

You now have something that can be compiled and executed, go ahead and do that now. When you click the button, the list box is filled with all of the manufacturers contained in modemcap.

The next thing to do is display a list of modems from a selected manufacturer. Drop another TListBox on the form and change the name to lbxModels. Populate this list when a manufacturer name is selected in the lbxManufacturers TListBox, so create the OnClick (or OnSelected) event handler of the lbxManufacturers component and add the following:

```
procedure TForm1.lbxManufacturersClick(Sender: TObject);
  { displays all modems from the selected manufacturer }
var
  I : Integer;
begin
  if lbxManufacturers.ItemIndex > -1 then begin
  lbxModels.Items.Clear;
    for I := 0 to pred(ModemColl.Count) do
      if ModemColl[I].Manufacturer =
lbxManufacturers.Items[lbxManufacturers.ItemIndex] then
        lbxModels.Items.Add(ModemColl[I].ModemName);
  end;
end;
```

This event handler will iterate through the collection of modem records and add the names of the manufacturer's modems to the second list box. You can compile and execute the project now and see the list of manufacturers, when you select one of them the second list box displays modems by that manufacturer.

The final step in our little project is to select a modem from the second list box, extract the modem details and add them to a new detail file. To do this, create the OnDblClick event handler for the lbxModels TListBox. In this method, create load the selected modem's details, and then save them to another file.

```
procedure TForm1.lbxModelsDblClick(Sender: TObject);
var
  LmModem : TLmModem;
  ModemName : string;
  SourceDetailFile : string;
  I : Integer;
begin
  if lbxModels.ItemIndex > -1 then begin
    ModemName := lbxModels.Items[lbxModels.ItemIndex];
    for I := 0 to pred(ModemColl.Count) do
      if ModemColl[I].ModemName = ModemName then begin
        SourceDetailFile := ModemColl[I].ModemFile;
        Break;
      end;
    LibModem.GetModem(SourceDetailFile, ModemName, LmModem);
    LibModem.AddModem('mymodems.xml', LmModem);
  end;
end;
```

This project can be used to create a “installed modems” detail file, which can then be used without the rest of the modemcap database.

Related examples

EXMDM.DPR

EXMDMCAP.DPR

EXLIBMDM.DPR

Configuring a TAPI Device

This topic shows how to perform custom configuration for a TAPI device.

One of the best reasons to use TAPI to configure a device is that the operating system maintains the list of commands and responses so the same code-base can communicate seamlessly with a wide variety of modems. What could be simpler than the operating system maintaining the modem's configuration? Most of the time, this is a much better solution than having to maintain a separate list of modems and modem configuration; but what can you do if the configuration supplied from the operating system doesn't work with your requirements?

Required components

TApdTapiDevice

Prerequisite topics

"Selecting and Configuring a Modem" on page 85.

Related components

None

What to do

The normal configuration for a TAPI device is good for normal connections (i.e., 8 data bits, no parity, 1 stop bit, hardware flow control, etc.). Each time the ApdTapiDevice opens the physical serial port, the normal TAPI configuration is forced on the TApdComPort. Let's say you need to connect to a 7E1 host—how can you do that if TAPI is forcing 8N1? Well, you have to change the configuration before TAPI makes the connection.

TAPI uses an opaque structure to contain the configuration for the device. This means that you can't change anything in that structure directly, and it will probably change from one modem to the next. You will need to get the configuration from TAPI through the Modem Properties dialog, let the user change the properties to match your configuration requirements, then save the configuration structure. Each time you need to use the configuration for that modem, load the configuration structure and apply it to the TAPI device. You'll find detailed steps below, but first let's talk a bit about the structures and methods that Async Professional's TApdTapiDevice gives you.

The TTapiConfigRec record is designed to hold the TAPI configuration structure.

The GetDevConfig method gives you the current configuration structure.

The SetDevConfig method forces a new configuration structure on the TAPI device.

The ShowConfigDialogEdit method displays the Modem Properties dialog and gives you the changed configuration.

The first thing to do is select the device that you want to modify by setting the TApdTapiDevice.SelectedDevice property. Next, show the Modem Properties dialog box with the current configuration and get the modified configuration back from TAPI. Finally, save the modified configuration so you can load it in the future. Since TAPI devices can be added or removed, it is a good idea to keep track of the modem's name so you don't accidentally use the wrong configuration structure. The following example shows how to do it:

```
procedure TForm1.ReconfigureTheTapiDevice;
var
  CfgRec : TTapiConfigRec;
  Reg : TRegistry;
begin
  { Initialize the record with current config, show config dialog
    and store the result }
  CfgRec := ApdTapiDevice.ShowConfigDialogEdit(
    ApdTapiDevice.GetDevConfig);

  { Set the device configuration with the result from the dialog }

    ApdTapiDevice.SetDevConfig(CfgRec);

  { Save the new configuration to the registry. This assumes a key
    of 'Software\MyApp' exists, each TAPI device that is
    configured will have it’s own key named the same as the
    device name }
  Reg := TRegistry.Create;
  try
    Reg.OpenKey(
      'Software\MyApp' + ApdTapiDevice.SelectedDevice, True);
    Reg.WriteBinaryData(
      'TapiConfig', CfgRec.Data, CfgRec.DataSize);
  finally
    Reg.Free;
  end;
end;
```

To load the configuration when you need it, do something like this:

```
procedure TForm1.ConfigureTheDevice;
var
  CfgRec : TTapiConfigRec;
  Reg : TRegistry;
begin
  { Clear record }
  FillChar(CfgRec, SizeOf(CfgRec), #0);

  { Make sure we have selected a valid TAPI device }

  if (ApdTapiDevice.SelectedDevice = '') or
    (ApdTapiDevice.TapiDevices.IndexOf(
      ApdTapiDevice.SelectedDevice) = -1) then
    ApdTapiDevice.SelectDevice;

  { Get configuration from registry }
  Reg := TRegistry.Create;
  if Reg.KeyExists(
    'Software\MyApp' + ApdTapiDevice.SelectedDevice) then begin
    try
      Reg.OpenKey('Software\MyApp' +
        ApdTapiDevice.SelectedDevice, False);
      CfgRec.DataSize := Reg.ReadBinaryData('TapiConfig',
        CfgRec.Data, SizeOf(CfgRec.Data));
    finally
      Reg.Free;
    end;
    { Set the device configuration }
    ApdTapiDevice.SetDevConfig(CfgRec);
  end else
    { This modem hasn’t been added, so we need to show the dialog
      and add the custom configuration }
    ReconfigureTheTapiDevice;
end;
```

Related examples

EXTCONFG.DPR

Dialing

This topic shows how to dial a phone number through a modem or similar device.

Sooner or later, you'll probably want to dial a modem and connect to another system. This can be done quite easily with Async Professional, but the exact commands to use will depend on which components are used to control the port.

Required components

TApdComPort

TApdTapiDevice

Prerequisite topics

"Setting Up a Comport" on page 74.

"Sending Characters" on page 76.

"Selecting and Configuring a Modem" on page 85.

Related components

TApdModemDialer

TApdDialerDialog

What to do

TApdComPort

If you are establishing a connection without a TApdTapiDevice component, then you or your code must send commands to the port directly (how you would do it is covered later). The usual AT modem command for dialing is "ATDT" for tone dialing and "ATDP" for pulse dialing. To perform a tone dialing of "260-9726" your code would look something like:

```
ApdComPort1.Output := 'ATDT 260 9726'#13;
```

When the remote answers, and the modems negotiate a mutually supported connection type, the modem will return "CONNECT." You can use a TApdDataPacket component to detect the connect message.

TAdModem and TApdTapiDevice

The TAdModem and TApdTapiDevice components each have a method called Dial that dials a given number. The TAdModem.Dial method will configure the modem, then dial the number specified by the PhoneNumber property. The TApdTapiDevice.Dial method tells TAPI to configure the modem and dial the number specified by the parameter for the method.

```
  AdModem1.Dial ('555 1212');
```

or

```
  ApdTapiDevice1.Dial ('555 1212');
```

The phone numbers used for these methods will dial the given phone number, without regard to the TAPI location settings (dial ‘9’ for an outside line, long distance prefixes, etc.). The TAdModem will need those modifications made manually. With TAPI, the TApdTapiDevice.TranslateAddress method will convert the phone number for you.

When the TAdModem component detects that the modem is connected, the OnModemConnect event is generated. When TAPI detects that the connection is made, the OnTapiConnect and OnTapiPortOpen events will fire.

Related examples

ApdTapiDevice

EXTAPID.DPR

EXSMODEM.DPR

Monitoring the Progress of a Dial Attempt

This topic shows how to keep track of what is happening after a dial command is executed up to the time the connection is established.

Simply dialing is usually not enough to do anything in a communications application; eventually, you'll want to do something once the connection is made. The TApdTapiStatus component gives the user a visual indication of the dialing status, but they provide little feedback to what's happening in your application. Fortunately, the TApdTapiDevice component supplies events that fire throughout the connection process and give you the means of providing feedback to the user.

Required components

TApdComPort

TApdTapiDevice (32-bit)

Prerequisite topics

"Setting Up a Comport" on page 74.

"Selecting and Configuring a Modem" on page 85.

"Dialing" on page 94.

Related components

TApdTapiStatus

What to do

TApdTapiDevice

Unlike the TApdSModem component, the TApdTapiDevice component relies on TAPI to provide the connection information. After you execute the Dial method, the OnTapiConnect event fires to notify your application of a successful connection. If the connection attempt failed, the OnTapiFail event fires. For specific information about the connection attempt, use the OnTapiStatus event. There are several parameters passed in this event, but only a few are used in this particular case. The First and Last parameters are True on the first and the last status event for the current call respectively. The Message parameter provides the category of the status change, Param1 defines the specific change, and Param 2 defines extended information. The following example dials a number and uses the OnTapiStatus event to update a label.

```
procedure TForm1.DialBtnClick(Sender : TObject);
begin
  ApdTapiDevice1.Dial('260 9756');
end;

procedure TForm1.ApdTapiDevice1TapiStatus(
  CP : TObject; First, Last : Boolean; Device, Message, Param1,
  Param2, Param3 : Integer);
begin
  if First then
    Label1.Caption := 'Got first status message'
  else if Last then
    Label1.Caption := 'Got last status message'
  else
    Label1.Caption :=
      ApdTapiDevice1.TapiStatusMsg(Message, Param1, Param2);
end;
```

Related examples

ApdTapiDevice

EXTAPID.DPR

Terminating a Connection

This topic shows how to terminate a connection by hanging up the device/modem.

Now that you know how to connect to another system, you need to know how to disconnect from it (think of the phone bill otherwise). If you are using the TApdComPort component directly, you can just set the Open property to False to terminate the connection, but that is about as elegant as unplugging the phone line if you are using the TApdSModem or TApdTapiDevice components.

Required components

TApdComPort

TAdModem

TApdTapiDevice

Prerequisite topics

“Setting Up a Comport” on page 74.

“Dialing” on page 94.

Related components

None

What to do

The TApdTapiDevice.CancelCall and TAdModem.CancelCall methods terminate either a dialing attempt or an established connection. When CancelCall is executed, a request is sent to TAPI. CancelCall will return once the call has been cancelled, which could be immediately or can take a few seconds, depending on how responsive TAPI is at the time the request was made. The OnTapiPortClose event will fire if the port has been opened (if you had an OnTapiPortOpen event previously for this call). The OnTapiFail event will also fire if a connection attempt has been made. The TApdTapiDevice.Cancelled property will be True in the OnTapiFail event if you called CancelCall. It will be False otherwise. The following example terminates a TAPI call.

```
procedure TForm1.HangupBtnClick(Sender : TObject);
begin
  ApdTapiDevice1.CancelCall;
end;

procedure TForm1.ApdTapiDevice1TapiFail(Sender : TObject);
begin
  if ApdTapiDevice1.Canceled then
    { Cancelled the call }
  else
    { TAPI detected a failure }
  end;
end;

procedure TForm1.ApdTapiDevice1TapiPortClose(Sender : TObject);
begin
  { TAPI has now terminated the connection, do any cleanup
    routines }
end;
```

Related examples

EXTAPID.DPR

Sending Files

This topic shows how to send text or binary files to a remote system via a serial connection.

You can connect to another system, send and receive characters and strings, and terminate that connection, but what if you needed to send files? File transfers are generalized under the term Protocol Transfers. A protocol is simply an established method of communication. In a file transfer the communication is the contents of the file. Included in the protocol are specific strings or characters that handle errors in the transfer, file information and handshaking. For most cases, Zmodem is the protocol of choice as it offers speed, reliability, and convenience.

Required components

TApdComPort

TApdProtocol

Prerequisite topics

"Setting Up a Comport" on page 74.

"Dialing" on page 94.

Related components

TApdProtocolStatus

TApdProtocolLog

What to do

The TApdProtocol component can be used with a connection made with a TApdSModem, TApdTapiDevice, TApdWinsockPort, or a TApdComPort component. The first thing to do is to select which protocol you want to use. For most cases, Zmodem is your best choice, and it is the default in the ProtocolType property. The default property values are also appropriate for most transfers.

The next step is to determine which file or files are to be sent. If you are sending a single file, enter the path and file name in the FileMask property. If you are sending multiple files, enter a DOS file mask that identifies the files. The DOS file mask can contain wildcards (e.g., *.TXT, REPORTS.*, or *.*, etc.).

If the files can not be described with a file mask, or you are using a non-batch protocol such as XModem, use the OnProtocolNextFile event to specify the files. The OnProtocolNextFile event fires immediately following a call to StartTransmit, and after each file is sent. Set the FName parameter to the path and file name of the file to be sent. To signal the end of the files to be transferred, set the FName parameter to an empty string. Don't enter anything in the FileName property as that is relevant only for protocol receiving. Also, the FileMask property is ignored when the OnProtocolNextFile event is used.

To start the transfer, call the StartTransmit method. When the transfer is complete, the OnProtocolFinish event is fired. To see how the transfer is going, drop a TApdProtocolStatus component on the form. It is displayed automatically when StartTransmit is called. The OnProtocolFinish event fires when all files specified in the FileMask property or from the OnProtocolNextFile event have been sent. The ErrorCode parameter is 0 if the transfer session was successful. The OnProtocolFinish event fires only after all files have been sent. Therefore, it should not be used to determine the status of individual files. For that purpose, use the OnProtocolLog event which fires at the start and end of transmission of each file, successful or not. The Log parameter of the OnProtocolLog event provides the state of the transfer.

The following example sends a single file selected from an OpenFile dialog, which is invoked in the SendFileBtnClick event, or sends all files in the "C:\SEND" directory from the SendAllFilesBtnClick event. The status of each file sent is added to a TMemo.Lines string list in the OnProtocolLog event so you can see the progress of the transfer.

```
procedure TForm1.SendFileBtnClick(Sender : TObject);
begin
  OpenDialog1.FileMask := 'All files (*.*)|*.*';
  if OpenDialog1.Execute then begin
    ApdProtocol1.FileMask := OpenDialog1.FileName;
    ApdProtocol1.StartTransmit;
  end;
end;

procedure TForm1.SendAllFilesBtnClick(Sender : TObject);
begin
  ApdProtocol1.FileMask := 'C:\SEND\*.*';
  ApdProtocol1.StartTransmit;
end;
```

```
procedure TForm1ApdProtocol1ProtocolLog(
  CP : TObject; Log : Word);
begin
  case Log of
    lfTransmitStart : Memo1.Lines.Add(
      ApdProtocol1.FileName + ' : started');
    lfTransmitOK    : Memo1.Lines.Add(
      ApdProtocol1.FileName + ' : sent OK');
    lfTransmitFail  : Memo1.Lines.Add(
      ApdProtocol1.FileName + ' : failed');
    lfTransmitSkip  : Memo1.Lines.Add(
      ApdProtocol1.FileName + ' : rejected');
  end;
end;

procedure TForm1.ApdProtocol1ProtocolFinish(
  CP : TObject; ErrorCode : SmallInt);
begin
  ShowMessage('File transfer complete');
end;
```

The following example sends all the files selected from an OpenFile dialog (configured to accept multiple files), updates a status label with the status of the current file, and shows a message once all files have been transferred. The list of files to be sent is maintained in a TMemo and, as each file is transmitted, its name is removed from the list. Once all files are transferred, the protocol transfer is complete.

```
procedure TForm1.SendFilesBtnClick(Sender : TObject);
begin
  OpenDialog1.FileMask := 'All files (*.*)|*.*';
  OpenDialog1.Options := [ofAllowMultiSelect];
  if OpenDialog1.Execute then begin
    Memo1.Lines.Clear;
    Memo1.Lines := OpenDialog1.Files;
    ApdProtocol1.FileMask := Memo1.Lines[0];
    Memo1.Lines.Delete(0);
    ApdProtocol1.StartTransmit;
  end;
end;

procedure TForm1.ApdProtocol1ProtocolNextFile(CP : TObject;
  var FName : TPassString);
```

```
begin
  if Memo1.Lines.Count > 0 then begin
    FName := Memo1.Lines[0];
    Memo1.Lines.Delete(0);
  end;
end;

procedure TForm1.ApdProtocol1ProtocolLog(
  CP : TObject; Log : Word);
begin
  case Log of
    lfTransmitStart : Status.Caption :=
                        'Sending ' + ApdProtocol1.FileName;
    lfTransmitOK    : Status.Caption :=
                        ApdProtocol1.FileName + ' sent OK';
    lfTransmitFail  : Status.Caption :=
                        ApdProtocol1.FileName + ' failed';
  end;
end;

procedure TForm1.ApdProtocol1ProtocolFinish(
  CP : TObject; ErrorCode : SmallInt);
begin
  ShowMessage('File transfers complete');
end;
```

Related examples

EXPROT.DPR

EXZSEND.DPR

Receiving Files

This topic shows how to receive text or binary files from a remote system via a serial connection.

There are many situations where the information you want to get from the remote is in a file. The TApdProtocol component can receive these files as well as send them. Again, Zmodem is the preferred protocol—offering speed, robustness, and convenience. The Zmodem sender also passes the file name of the file being transferred.

Required components

TApdComPort

TApdProtocol

Prerequisite topics

"Setting Up a Comport" on page 74.

"Detecting a Specific String in the Data Stream" on page 80.

Related components

TApdProtocolStatus

TApdProtocolLog

What to do

The TApdProtocol component can be used with a connection made with a TApdSModem, TApdTapiDevice, TApdWinsockPort, or a TApdComPort component. The first thing to do is to select which protocol to use. For most cases, Zmodem is your best choice, and it is the default in the ProtocolType property. The default property values are also appropriate for most transfers.

The Zmodem sender typically sends 'rz'#13 (which stands for "receive zmodem") to initiate the transfer; this makes Zmodem the easiest protocol to implement. When StartReceive is called, the TApdProtocol component waits for the 'rz'#13 string and starts receiving the file or files once that string is detected. The other protocols must be started explicitly at the appropriate time.

For protocols that support batch transfers, e.g., Zmodem, Ymodem, YmodemG, and Kermit, several files can be transmitted in one session. The OnProtocolFinish event fires when all files have been received. The OnProtocolLog event fires at beginning and end of transmission for each individual file.

The following example receives either a single file or a batch of files. As each file is transmitted, a status label is updated. When the file transmission is complete, the file name is added to a TMemo. Once the file or files are transferred, a message is displayed.

```
procedure TForm1.Button1Click(Sender : TObject);
begin
  ApdProtocol1.StartReceive;
end;

procedure TForm1.ApdProtocol1ProtocolLog(
  CP : TObject; Log : Word);
begin
  case Log of
    lfReceiveStart : Status.Caption :=
                       'Receiving ' + ApdProtocol1.FileName;
    lfReceiveOK    : Memo1.Lines.Add(
                       ApdProtocol1.FileName + ' received OK');
    lfReceiveFail  : Memo1.Lines.Add(
                       ApdProtocol1.FileName + ' failed');
  end;
end;

procedure TForm1.ApdProtocol1ProtocolFinish(
  CP : TObject; ErrorCode : Integer);
begin
  ShowMessage('Transfer complete');
end;
```

Related examples

EXZRECV.DPR

Sending and Receiving Faxes on the Same Line

This topic shows how to send a fax while waiting to receive a fax.

A very popular usage of the APRO fax components is to use a single line to receive faxes, and send faxes using the same line. The TApdReceiveFax component traditionally opened the serial port for exclusive access while it waited for an incoming call. When the TApdReceiveFax component was waiting, the port could not be used by the TApdSendFax component. The integration of TAPI with the TApdReceiveFax component greatly simplifies this design.

Required components

TApdComPort

TApdSendFax

TApdReceiveFax

TApdTapiDevice

Prerequisite topics

"Configuring a Device for Faxing" on page 142

"Sending Faxes to One Recipient" on page 143

"Receiving Faxes" on page 150

Related components

TApdFaxStatus

TApdFaxLog

What to do

The TApdReceiveFax and TApdSendFax components are TAPI aware, and can use the services provided by TAPI. TAPI is capable of monitoring a device for incoming calls, while permitting other TAPI processes to access the port. If the TapiDevice property of the TApdReceiveFax component is assigned, a call to the StartReceive method will begin passive monitoring of the line. When TAPI detects an incoming call, the TApdReceiveFax component will acquire the port, receive the fax, and return to passively monitoring for more calls.

To illustrate this, create a new project and drop a TApdComPort, TApdTapiDevice, and TApdReceiveFax on the form. Set the TapiDevice property of the TApdReceiveFax component to the TApdTapiDevice, which will enable TAPI integration. Drop the obligatory TButton on the form and add the following code:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ApdReceiveFax1.StartReceive;
end;
```

When this is executed, the TApdTapiDevice will enter “autoanswer” mode and wait for incoming calls. Once TAPI has detected the appropriate number of ring signals, the TApdReceiveFax component will take over and answer the call.

To add fax-sending capabilities, drop another TApdComPort and another TApdTapiDevice and a TApdSendFax component on the form. Make sure that the new TApdTapiDevice’s ComPort property is pointing to the new TApdComPort; and that the ComPort and TapiDevice properties of the TApdSendFax are pointing to the new components also. Drop another button on the form and add the following to the new button’s OnClick event handler:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  if ApdReceiveFax1.InProgress then begin
    ShowMessage('Receiving a fax, try again later');
    Exit;
  end;
  ApdSendFax1.PhoneNumber := '555 1212';
  ApdSendFax1.FaxFile := 'c:\default.apf';
  ApdSendFax1.StartTransmit;
end;
```

The InProgress property of the TApdReceiveFax will be True when the component is actually receiving a fax, and False when waiting for an incoming call. Check InProgress to make sure the line is available for sending.

To make the TApdSendFax and TApdReceiveFax components work together through TAPI, their TapiDevice properties must be pointing to separate TApdTapiDevice components, and the SelectedDevice properties of the TApdTapiDevice components must be pointing to the same TAPI device.

Related examples

EXTAPIFAX.DPR

Inserting a Delay

This topic shows how to delay execution of the next statement or section in your code for a specified amount of time.

Even though working in the event-driven environment of Windows, there may come a time when you want to momentarily pause execution of your application. One example of this is if you send a command to a modem and need to wait for the command to be processed, but do not want to set up a TApdDataPacket component or DataTrigger for a single use.

Required components

None

Prerequisite topics

None

Related components

None

What to do

There are several useful utilities provided with Async Professional. Perhaps the most useful is the DelayTicks function in the OOMISC.PAS unit. This function pauses execution of your code for a specified number of clock ticks. Before trying to use this function, you must include "OoMisc" to the uses clause of the unit that calls DelayTicks. The function takes two parameters, Ticks and Yield, and returns a LongInt.

```
function DelayTicks(Ticks : LongInt; Yield : Bool) : LongInt;
```

Ticks is the number of clock ticks for which to delay. There are about 18 clock ticks per second. Yield determines whether DelayTicks prevents other tasks from executing, or if it delays execution only within the current method. For most cases, you can disregard the result of this function. However, for the record, the low word of the result is the last Windows message number and the high word is the windows handle that received the message.

The following example sends an Init command to a modem, pauses for 2 seconds, and then sends a dial command:

```
procedure TForm1.ConfigAndDialBtnClick(Sender : TObject);
begin
  ApdComPort1.Output := 'ATZ'#13;
  DelayTicks(36, True);
  ApdComPort1.Output := 'ATDT 260 9726'#13;
end;
```

Related examples

None

Displaying Status Lights

This topic shows how to display the status of the different line states of the serial port.

Most communications applications display virtual lights that show when a connection is made, if data is being sent or received, whether or not flow control is in effect, and so on. The TApdSLController and TApdStatusLight components display these line states, providing visual cues to the current state of the connection.

Required components

TApdComPort

TApdSLController

TApdStatusLight

Prerequisite topics

“Setting Up a Comport” on page 74.

Related components

TAdTerminal

What to do

There are eight line states that can be monitored in Async Professional through status lights as shown in Table 5.1.

Table 5.1: *Async Professional line states*

State	Description
BREAK	Line break
CTS	Clear to send
DCD	Data carrier detect
DSR	Data set ready
ERROR	Line error
RING	Ring indicator
RXD	Receiving data
TXD	Transmitting data

Drop a TApdComPort component on the form and configure it to match your system setup. Next, place a TApdSLController component on the form. This component is the interface between the comport and the TApdStatusLight components. Now drop a TApdStatusLight component on the form for each state that you want to monitor. For an added touch, you might place a label next to each light to indicate which state each light is monitoring. After all the lights are on the form, select the TApdSLController component and double-click the Lights property in the Object Inspector. For each state to be monitored, select one of the TApdStatusLight components. The last step is to enable the lights. This is done via the run-time Monitoring property of the TApdSLController. Set the property to True in your code to enable the lights. Set it to False to disable the lights. The following example opens and closes the port along with enabling and disabling the status lights:

```
procedure TForm1.OpenBtnClick(Sender : TObject);
begin
  ApdComPort1.Open := True;
  ApdSLController1.Monitoring := True;
end;

procedure TForm1.CloseBtnClick(Sender : TObject);
begin
  ApdSLController1.Monitoring := False;
  ApdComPort1.Open := False;
end;
```

Related examples

TERMDEMO.DPR

Detecting Line State Changes

This topic shows how to detect line state changes programmatically.

The status lights are nice to have, but they are only visual clues for your users (i.e., you cannot use them to handle different line state changes in your code). To do that, you must delve into triggers.

Required components

TApdComPort

Prerequisite topics

“Setting Up a Comport” on page 74.

Related components

TApdSLController

TApdStatusLight

What to do

The same status events that the TApdSLController component monitors can be handled with Status Triggers. A Trigger is a hook into the activities of the TApdComPort. When a trigger condition is met, its trigger event is fired. Status Triggers are internal flags used to fire one of several events of the TApdComPort component (i.e., OnTriggerStatus, OnTriggerModemStatus, and OnTriggerLineStatus). There are two steps required to make use of Status Triggers: first they must be added and, second, they must be set. The trigger type, either modem, line, output buffer, or transmission, is specified when it is added and the exact condition(s) is specified when it is set. When the conditions are met, the general purpose OnTriggerStatus event fires. Other events fire according to the trigger type.

The status trigger types and events they fire are shown in Table 5.2.

Table 5.2: *Status trigger types*

Trigger	Description
stModem	Trigger on modem status change
stLine	Trigger on line status change
stOutBuffFree	Trigger on output buffer free value
stOutBuffUsed	Trigger on output buffer used value
stOutSent	Trigger on characters sent

A trigger type of stModem can trigger on the state changes shown in Table 5.3 and fires the OnTriggerModemStatus event.

Table 5.3: *stModem state changes*

Trigger	Description
msCTSDelta	Trigger when CTS (Clear To Send) changes
msDSRDelta	Trigger when DSR (Data Set Ready) changes
msRingDelta	Trigger when a ring is detected
msDCDDelta	Trigger when DCD (Data Carrier Detect) changes

A trigger type of stLine can trigger on these state changes and fires the OnTriggerLineError event:

Table 5.4: *stLine state changes*

Trigger	Description
lsOverrun	Trigger on UART overrun errors
lsParity	Trigger on parity errors
lsFraming	Trigger on framing errors
lsBreak	Trigger on a received line break signal

A trigger type of stOutBuffFree occurs when the output buffer has more than the specified space available and fires the OnTriggerOutBuffFree event.

A trigger type of stOutBuffUsed occurs when the output buffer has less than the specified space used and fires the OnTriggerOutBuffUsed event.

A trigger type of stOutSent occurs when any call to PutChar or PutBlock is made, including assignments to the Output property and fire the OnTriggerOutSent event.

Each trigger that is added returns a value of type Word that indicates which trigger fired the event. This result is an index into an internal trigger array. The specific value of it may change between each instance of your application so don't assume it's always the same. The following example detects changes in DCD, when the output buffer is 90% full, when characters have been sent to the output buffer, and updates a status label. Except for the OutSent trigger, all the status triggers must be reset after they fire (i.e., they are “one shot” triggers and won't fire again).

The following example shows how to set up the triggers:

```
type
  TForm1 = class(TForm)
    ApdComPort1 : TApdComPort;
    StartBtn    : TButton;
    StatusLabel : TLabel;
    procedure StartBtnClick(Sender : TObject);
    procedure ApdComPort1TriggerStatus(CP : TObject;
TriggerHandle : Word);
  public
    { Public declarations }
    DCDTrig         : Word;
    OutBuffUsedTrig : Word;
    OutSentTrig     : Word;
  end;

var
  Form1 : TForm1;

implementation

{$R *.DFM}
```

```
procedure TForm1.StartBtnClick(Sender : TObject);
begin
  DCDTrig := ApdComPort1.AddStatusTrigger(stModem);
  ApdComPort1.SetStatusTrigger(DCDTrig, msDCDDelta, True);
  OutBuffUsedTrig := ApdComPort1.AddStatusTrigger(stOutBuffUsed);
  ApdComPort1.SetStatusTrigger(OutBuffUsedTrig, 500, True);
  OutSentTrig := ApdComPort1.AddStatusTrigger(stOutSent);
  ApdComPort1.SetStatusTrigger(OutSentTrig, 0, True);
end;

procedure TForm1.ApdComPort1TriggerStatus(CP : TObject;
  TriggerHandle: Word);
begin
  if TriggerHandle = DCDTrig then begin
    StatusLabel.Caption := 'DCD changed';
    ApdComPort1.SetStatusTrigger(DCDTrig, msDCDDelta, True);
  end else if TriggerHandle = OutBuffUsedTrig then begin
    StatusLabel.Caption := 'Output buffer has more that 500
                            chars pending';
    ApdComPort1.SetStatusTrigger(OutBuffUsedTrig, 500, True);
  end else if TriggerHandle = OutSentTrig then
    StatusLabel.Caption := 'Something was transmitted'
end;

end.
```

Related examples

None

Flow Control

This topic shows how to set and monitor flow control settings.

There are times when the receiving application cannot process incoming data as fast as it is being sent. To keep the receiver from being overloaded, there are line states that can be toggled to tell the sender to wait until the receiver is ready.

Required components

TApdComPort

Prerequisite topics

“Setting Up a Comport” on page 74.

Related components

None

What to do

There are two methods of flow control: Hardware and Software. Hardware flow control uses the physical lines of the port to signal its state. Software flow control sends specific characters indicating that flow control is on or off. As a general rule, you should use Hardware Flow Control, since the Software Flow Control requires that characters be sent in the data stream.

Hardware Flow Control

To implement Hardware Flow Control, set the TApdComPort.HWFlowOptions property to the set of options you want. For Receive Flow Control, where the remote device is prohibited from sending more characters, use hwfUseDTR and hwfUseRTS. These properties lower the DTR and/or RTS signals when the local input buffer reaches the value of BufferFull. For Transmit Flow Control, where the local device acknowledges the remote’s Receive Flow Control and stops sending characters, use hwfRequireDSR and hwfRequireCTS. These types of flow control work because one system’s RTS is the other’s CTS, likewise for DTR and DSR.

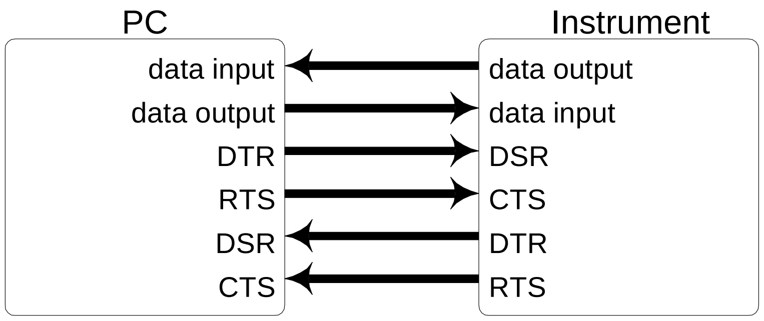


Figure 5.1: Data transmission.

The RTS and DTR properties of the TApdComPort refer to the RTS and DTR lines of the local serial port. With hardware flow control, RTS and DTR are changed automatically depending on the state of the local system. These lines can also be changed explicitly. The DSR and CTS properties are read-only, run-time properties since they refer to the state of the remote system.

If that all seems confusing, it is. Simply put, if you want to implement hardware flow control, the usual procedure is to set only the hwfUseRTS and hwfRequireCTS properties to True. It is far less common to set hwfUseDTR and hwfRequireDSR to True. It is even more rare (almost never) to set all four to True.

Software Flow Control

To implement Software Flow Control, set the TApdComPort.SWFlowOptions to the desired state of flow control. For Receive Flow Control, use swfReceive; for Transmit flow control, use swfTransmit; for both use swfBoth. When the BufferFull value is reached, the XOn character is sent to the sender. The sender will not send further characters until the XOff character is sent.

Monitoring Flow Control

The TApdComPort.FlowState property returns the current flow control state. If FlowState is fcOff, flow control is not enabled. If FlowState is fcOn, flow control is enabled, but no flow control is in effect. If FlowState is fcDsrHold, fcCtsHold, or fcDcdHold, the remote has lowered DSR, CTS, or DCD, preventing your application from sending characters. If FlowState is fcXOutHold, fcXInHold, or fcXBothHold, software flow control is in effect.

Related examples

None

Dialing with RAS

This topic shows how to dial up a network connection with the TApdRasDialer.

Before you can use a TApdWinsockPort or TApdFtpClient to communicate with another machine on an Inter/Intranet, you need to have a network connection. If you do not have an network connection, the easiest way to make such a connection is through Dial-Up Networking or Remote Access Service. See “RAS Dialer Demo” on page 163 for a demonstration.

Required components

TApdRasDialer

Prerequisite topics

None

Related components

TApdRasStatus

What to do

This tutorial assumes that RAS (*Remote Access Service*, also known as *Dial-Up Networking*) is installed on your machine. RAS is installed by default on most Win95/98/ME machines. On NT/2000 machines, however, RAS is not installed until you add a modem device to the system configuration.

First, you need to specify the desired phonebook EntryName which contains necessary dialup networking connection settings like which modem to use and the phone number to dial. If you have already used Dial-Up Networking then you probably have at least one entry in the phonebook. Windows NT/2000 users can simply call the PhonebookDlg method to manipulate phonebook entries (and even dial with, in which case you’re done). Windows 95/98/ME users can obtain a list of the entries with the ListEntries method. Otherwise, you can create a phonebook entry with the CreatePhonebookEntry method.

If you choose, you can alter the network login information, such as UserName, Password, and Domain, or you can enter the information when prompted by a RAS dialog.

Next, you must specify the phone number to dial and the dial mode (synchronous or asynchronous) and call the Dial method. If you are dialing asynchronously you should create an OnDialStatus event handler or assign a TApdRasStatus component to the StatusDisplay property to monitor dialing progress.

The following example shows how to dial up a network connection using synchronous dialing mode:

```
procedure TForm1.Dial1Click(Sender: TObject);
var
  Error : Integer;
begin
  ApdRasDialer1.EntryName   := 'MyServer';
  ApdRasDialer1.PhoneNumber := '9,800-555-1212';
  ApdRasDialer1.DialMode    := dmSync;
  Error := ApdRasDialer.Dial;
  if (Error = ecOK) then
    Caption := 'Connected'
  else
    Caption := ApdRasDialer.GetErrorText(Error);
end;
```

Related examples

EXRAS1.DPR

Setting up a Winsock Port

This topic shows you how to set up a TApdWinsockPort component and connect to a Winsock Socket.

Direct connection over phone lines, where one modem calls another modem, does not fit the requirements of all applications. Sometimes, you will want to connect over a network using Winsock.

Required components

TApdWinsockPort

Prerequisite topics

“Setting Up a Comport” on page 74.

Related components

TApdRasDialer

TApdSocket

What to do

The TApdComPort component is designed for communications through the serial ports while the TApdWinsockPort component is designed for communications through Winsock. The Async Professional implementation of Winsock relies on an existing Winsock/network connection. The easiest way to make such a connection is usually through Dial-Up Networking or Remote Access Service, although you can get a Winsock connection through an Inter/Intranet connection. The TApdWinsockPort component is the interface between your application and the Winsock DLL. It is a direct descendant of the TApdComPort component and shares much of the same functionality, but there are a few important differences. Winsock is a level above serial ports so it does not use the ComNumber property. The Parity, StopBits, and DataBits properties are also not used, since Winsock defines its own settings.

The TApdWinsockPort component can handle a single connection, either in Client or Server mode. A Winsock Client connects to a Winsock Server, which is essentially the same as a Client that dials and a Host that answers with serial communications. Set the WsMode property to the appropriate mode. For a Client, also set the WsAddress property to the network address of the server, and the WsPort property to the port on the server you intend to connect to. When a Server is opened (using the Open property) it listens for a connection attempt from a client. When the Client is opened, it tries to connect to the server. The OnWsConnect event fires when the connection is made and the OnWsDisconnect event fires when the connection is broken.

The network connections are listed in the WsLocalAddresses property. A network connection to use can be chosen by setting the WsLocalAddressIndex property to one of the addresses specified in WsLocalAddresses.

If you need to connect to the remote site through a proxy server, the WsSocksServerInfo property allows you to specify the proxy server to use. This property expands into several subproperties that allow you to provide the address, port, user name, password and the version of SOCKS support to use.

The following example connects to an Internet chess site through an existing Winsock connection:

```
procedure TForm1.ConnectBtnClick(Sender : TObject);
begin
  ApdWinsockPort1.WsAddress := 'ics.onenet.net'
  ApdWinsockPort1.WsPort := 'telnet';
  ApdWinsockPort1.Open;
end;

procedure TForm1.ApdWinsockPort1WsConnect(Sender : TObject);
begin
  Caption := 'Connected';
end;
```

Related examples

EXCLIENT.DPR

EXSERVER.DPR

EXWZSEND.DPR

EXWZRECV.DPR

Logging in to an FTP Server

This topic shows how to log in anonymously to an FTP server with the TApdFtpClient.

FTP servers require successful login before they allow a file download.

Required components

TApdFtpClient

Prerequisite topics

“Dialing with RAS” on page 161.

Related components

None

What to do

The TApdFtpClient is designed to communicate through Winsock to an FTP server and it relies on an existing Winsock/network connection. If you do not have an Inter/Intranet connection, the easiest way to make such a connection is through Dial-Up Networking or Remote Access Service. To establish a Dial-Up connection, see the section describing how to use the TApdRasDialer.

To login to an FTP server you must first set the ServerAddress property to the domain name or IP address of an FTP server, set the UserName property to the user’s login ID, and set the Password property to the user’s login password. At this point the Login method can be called to open the connection. The OnFtpStatus event fires with scOpen status code when a control connection to the server is established, and then again with scLogin when the server has authenticated the user ID information.

Most FTP servers allow a user to login anonymously for restricted access to files at the server. To login anonymously, use “ANONYMOUS” for the login ID and your e-mail address for the password.

The following example connects to the TurboPower FTP server through an existing Winsock connection:

```
procedure TForm1.LoginBtnClick(Sender : TObject);
begin
  ApdFtpClient1.ServerAddress := 'ftp.turbopower.com';
  ApdFtpClient1.UserName := 'anonymous';
  ApdFtpClient1.Password := 'somebody@somedomain.com';
  ApdFtpClient1.Login;
end;

procedure TForm1.ApdFtpClient1FtpStatus(Sender: TObject;
  StatusCode: TFtpStatusCode; Info: PChar);
begin
  case StatusCode of
    scClose      : Caption := 'Disconnected';
    scOpen       : Caption := 'Connected';
    scLogin      : Caption := 'Logged in';
    scLogout     : Caption := 'Logged out';
    scFtpError   : Caption := 'Cannot log in';
    scWsError    : Caption := 'Winsock error';
  end;
end;
```

Related examples

EXFTP1.DPR

Changing the Current Working Directory of an FTP Server

This topic shows how to change the working directory at an FTP server with the TApdFtpClient.

Downloading several files from an FTP server where the files are located deep in a directory tree can be difficult if you are typing in the remote file path name each time. It can be much simpler to set the current working directory at the server and then work solely with file names.

Required components

TApdFtpClient

Prerequisite topics

"Logging in to an FTP Server" on page 153.

Related components

None

What to do

To change the current working directory after logging in to an FTP server, simply call the ChangeDir method of the TApdFtpClient with the desired remote path name. To change to the parent directory, use ".." for the remote path name.

The following example changes the client's current working directory on the TurboPower FTP server. It is assumed the user is already logged in.

```
procedure TForm1.AproUpdatesDirBtnClick(Sender : TObject);
begin
  ApdFtpClient1.ChangeDir('pub/apro/updates');
    {CWD is now pub/apro/updates}
end;

procedure TForm1.ParentDirBtnClick(Sender : TObject);
begin
  ApdFtpClient1.ChangeDir('..');
    {CWD is now pub/apro}
end;

procedure TForm1.ApdFtpClient1FtpStatus(Sender: TObject;
  StatusCode: TFtpStatusCode; Info: PChar);
begin
  case StatusCode of
    scComplete   : Caption := 'Directory changed';
    scFtpError   : Caption := 'check remote path name';
    scWsError    : Caption := 'Winsock error';
  end;
end;
```

Related examples

EXFTP1.DPR

Displaying the Contents of a Directory on an FTP Server

This topic shows how to display a listing of the contents of a directory (all file details or file names only) at an FTP server with the TApdFtpClient.

Before you can download a file from an FTP site, you may need to know what files are available and in which directory they are located.

Required components

TApdFtpClient

Prerequisite topics

“Logging in to an FTP Server” on page 153.

“Changing the Current Working Directory of an FTP Server” on page 143.

Related components

None

What to do

To obtain a listing of the contents of a remote directory call the ListDir method of the TApdFtpClient with the desired remote path name. If no remote path name is specified then the current working directory is assumed. A full listing includes file properties such as size, timestamp and attributes, whereas a names-only listing consists only of the file names.

The following is a few lines of a full listing of the contents of the TurboPower FTP server’s PUB directory:

```
09-02-99  10:14AM        71          00index.txt
09-02-99  10:14AM       <DIR>        abbrevia
03-02-99  11:47AM       <DIR>        analyst
09-02-99  10:14AM       <DIR>        apro
```

The following example obtains a full listing of the contents of the TurboPower FTP server's PUB directory. It is assumed the user is already logged in.

```
procedure TForm1.DirectoryBtnClick(Sender : TObject);
begin
  ApdFtpClient1.ListDir('pub', True);
end;

procedure TForm1.ApdFtpClient1FtpStatus(
  Sender: TObject; StatusCode: TFtpStatusCode; Info: PChar);
begin
  case StatusCode of
    scDataAvail  : Memo1.Lines.Add(StrPas(Info));
    scFtpError   : Caption := 'check remote path name';
    scWsError    : Caption := 'Winsock error';
  end;
end;
```

Related examples

EXFTP1.DPR

Downloading a File from an FTP Server

This topic shows how to download a file from an FTP server with the TApdFtpClient.

Now that you can login, change the remote working directory and display its contents, it is time to do what you intended to do in the first place: download a file.

Required components

TApdFtpClient

Prerequisite topics

“Logging in to an FTP Server” on page 153.

“Displaying the contents of a directory on an FTP Server” on page 149.

Related components

None

What to do

To transfer a file from an FTP server you simply need to call the RetrieveFile method of the TApdFtpClient with the remote and local file names. An additional parameter, RetrieveMode, is required to specify what to do if the local file already exists. You will probably want to write an event handler for the OnFtpStatus event to let you know when the transfer is complete and to provide progress updates.

The following example copies the file FILES.ALL from the TurboPower FTP server's PUB directory to c:\temp. It is assumed the user is already logged in.

```
procedure TForm1.DownloadBtnClick(Sender : TObject);
begin
  ApdFtpClient1.Retrieve(
    'pub/FILES.ALL', 'c:\temp\Files.all', rmReplace);
end;

procedure TForm1.ApdFtpClient1FtpStatus(
  Sender: TObject; StatusCode: TFtpStatusCode; Info: PChar);
begin
  case StatusCode of
    scTransferOK : Caption := 'download complete';
    scProgress   : Caption :=
                    IntToStr(ApdFtpClient1.BytesTransferred) +
                             'bytes transferred';
    scFtpError   : Caption := 'check remote path name';
    scWsError    : Caption := 'Winsock error';
  end;
end;
```

Related examples

EXFTP1.DPR

Paging with Winsock

This topic describes how to send an alphanumeric page over an Internet (TCP/IP) connection.

The TApdSNPPPager component provides the ability to send a page from your application. The page to be sent should consist of plain ASCII text of up to 255 characters in length (this may vary with TAP Paging servers, check with your service provider if you are in doubt).

Required components

TApdWinsockPort

TApdSNPPPager

Prerequisite topics

"Setting up a Winsock Port" on page 120.

Related components

TApdTAPPager

What to do

TApdSNPPPager is designed to send a single page to a single recipient on a single paging server at a time. For each page the relevant properties must be set and the Send method called.

The following example shows obtaining the relevant paging parameters from some common VCL components placed on a form and sending the page in response to a button click. It also shows simple page status monitoring by setting the caption of a TLabel within OnSNPPError and OnSNPPSuccess event handlers.

To create this, first create a new form and drop the following components onto it:

- A TApdSNPPPager(naturally).
- Two TEdits(one for the Socket Address, one for the Pager ID).
- A TMemo(for the message).
- A TLabel (for the status display).
- A TButton (to make it all happen).

Arrange these on the form in any way you find esthetic. Give the TButton some meaningful caption like “Send.” You might also want to drop on some additional TLabels and place and caption them to indicate what the TEdits and TMemo are for.

Next, double-click on the TButton and add the following code to the empty OnClick event handler generated in the Code Editor.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ApdWinsockPort1.wsAddress  := Edit1.Text;
  ApdWinsockPort1.wsPort     := '1234';
  ApdSNPPPager1.PagerID      := Edit2.Text;
  ApdSNPPPager1.Message      := Memo1.Lines;
  ApdSNPPPager1.Send;
end;
```

Next, click on the TApdSNPPPager component, switch to the Object Inspector, click on the Events tab, double-click in the OnSNPPError event, and add this code to the event handler.

```
procedure TForm1.ApdSNPPPager1SNPPError(
  Sender : TObject; Code : Integer; Msg : String);
begin
  Label1.Caption := Msg;
end;
```

Now, go back to the Object Inspector, click in the space next to the OnSNPPSuccess event, click on the down arrow that appears at the right and select the same method (ApdSNPPPager1SNPPError) used above.   This works because both event handlers are designed with the same “signature” so both of them can use the same event handler method.

Related examples

EXSNPP.DPR

Paging with Modems

This topic describes how to send a simple alphanumeric page over a phone line and modem.

The TApdTAPPager component provides the ability to send a page from your application. The page to be sent should consist of plain ASCII text of up to 255 characters in length (this may vary with TAP Paging servers, check with your service provider if you are in doubt).

Required components

TApdComPort

TApdTAPPager

Prerequisite topics

“Setting Up a Comport” on page 74.

Related components

TApdSNPPPager

What to do

TApdTAPPager is designed to send a single page to a single recipient on a single paging server at a time. For each page the relevant properties must be set and the Send method called.

The following example shows obtaining the relevant paging parameters from some common VCL components placed on a form and sending the page in response to a button click. It also shows simple page status monitoring by setting the caption of a TLabel within an OnPageStatus event handler.

To create this, first, create a new form and drop the following components on to it:

- A TApdTAPPager(naturally)
- Two TEdits(one for the Phone Number, one for the Pager ID)
- A TMemo(for the message)
- A TLabel (for the status display)
- A TButton (to make it all happen).

Arrange these on the form in any way you find esthetic. Give the TButton some meaningful caption like “Send.” You might also want to drop on some additional TLabels and place and caption them to indicate what the TEdits and TMemo are for.

Next, double-click on the TButton and add the following code to the empty OnClick event handler generated in the Code Editor.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ApdTAPPager1.PhoneNumber := Edit1.Text;
  ApdTAPPager1.PagerID     := Edit2.Text;
  ApdTAPPager1.Message     := Memo1.Lines;
  ApdTAPPager1.Send;
end;
```

Next, click on the TApdTAPPager component, switch to the Object Inspector, click on the Events tab, double-click in the OnPageStatus event, and add this code to the event handler.

```
procedure TForm1.ApdTAPPager1PageStatus(
  Sender : TObject; Event : TTapStatus);
begin
  Label1.Caption := TAPStatusMsg(Event);
end;
```

Related examples

EXPAGING.DPR

EXTAP.DPR

Sending an SMS Message

This topic shows how to access cellular phones, or other GSM compatible devices, and how to send a SMS message.

The TApdGSMPhone component provides the ability to send or receive a SMS message from your application. The TApdGSMPhone component implements the text-mode interface and each message can consist of up to 160 characters. If your cell phone is not GSM capable, most cellular service providers will offer a TAP or SNPP gateway. See the TApdTAPPager and TApdSNPPPager components for details on sending messages using those paging protocols.

Required components

TApdComPort

TApdGSMPhone

Prerequisite topics

"Setting Up a Comport" on page 74.

Related components

TApdTAPPager

TApdSNPPPager

What to do

Since the TApdGSMPhone component was designed for SMS messaging, the following example shows how to connect to a GSM phone and send a message. If you do not have a GSM phone or device, you can still use the TApdTAPPager or TApdSNPPPager components to send a message through the SMS gateway. Consult your cell phone service provider for message format and addresses.

To create this application, first, create a new form and drop a TApdComPort component, and a TApdGSMPhone component of course. Two TEdit components can be dropped on the form. One TEdit is for the phone number, which will be the destination address for the message. The other TEdit would be for the message itself. Two TLabel components would be appropriate for a title next to each TEdit to explain what they will contain. Finally, drop the obligatory TButton on the form, change the Caption and Name to “SendMessage” and create the OnClick event handler.

```
procedure TForm1.SendMessageClick(Sender: TObject);
begin
  { set the message properties }
  ApdGSMPhone1.SMSAddress := Edit1.Text;
  ApdGSMPhone1.SMSMessage := Edit2.Text;
  { send the message }
  ApdGSMPhone1.SendMessage;
end;
```

When the message has been sent to the phone, the OnSessionFinish event will be generated. The ErrorCode parameter of that event will tell you whether the message was sent successfully (ErrorCode = ecOK) or whether it failed (ErrorCode = one of the ecSMSXxx error codes). Create the OnSessionFinish event handler and make it look like the following:

```
procedure TForm1.ApdGSMPhone1SessionFinish(Pager:
TApdCustomGSMPhone; ErrorCode: Integer);
begin
  ShowMessage('Message status: ' + ErrorMsg(ErrorCode));
end;
```

Compile and run your project. Enter a destination address in the first edit control and a short message in the second edit control, and then click the button. The “Message status” dialog box will be displayed once the phone responds to the commands.

Related examples

EXSMSPGR.DPR

Managing SMS Messages

This topic shows how to access a cellular phone message store using the TApdGSMPhone component.

In addition to sending an SMS message, the TApdGSMPhone also provides access to the cell phone’s internal message store. This message store usually contains messages that have been received, but it can also contain messages queued for sending.

Required components

TApdComPort

TApdGSMPhone

Prerequisite topics

“Setting Up a Comport” on page 74.

“Sending an SMS Message” on page 134.

Related components

TApdTAPPager

TApdSNPPPager

What to do

The TApdGSMPhone component provides the MessageStore property, which can be synchronized with the cell phone’s internal message store. If the QuickConnect property of the TApdGSMPhone component is set to False, a call to the Connect method will simply initialize the device for Text mode. If QuickConnect is True, the Connect method will initialize the device, then will retrieve all of the messages stored in the phone’s internal message storage. At any time, you can call the Synchronize method, which will reload all of the messages from the phone.

The MessageStore property is a TStringList. After synchronization, each Strings item in the MessageStore is a string depicting the date and time of the message, and each Objects item is a TApdSMSMessage class which describes the message.

```
    TApdSMSMessage = class(TObject)
    public
      property Address : string;
      property Message : string;
      property MessageIndex : Integer;
      property Status : TApdSMSStatus;
      property TimeStamp : TDateTime;
      property TimeStampStr : string;
    end;
```

After synchronization, the MessageStore property can be used to manage the phone's internal message store. The Delete method will delete the message from storage and the AddMessage method will add a new message. Alternatively, the WriteToMemory method of the TApdGSMPhone component can be used instead of the AddMessage method. The SendFromMemory method of the TApdGSMPhone component will send a single message from storage, and the SendAllMessages method will send all unsent message from storage.

Create a new project and drop a TApdComPort and TApdGSMPhone component on the form. Change the ComNumber property of the TApdComPort component to reflect the serial port where your GSM phone is connected. Set the QuickConnect property of the TApdGSMPhone component to False. Drop a TButton on the form and change the Caption and Name properties to "Connect", then create the OnClick event handler and add the following code:

```
  procedure TForm1.ConnectClick(Sender : TObject);
  begin
    ApdGSMPhone1.Connect;
  end;
```

The phone will be initialized and the phone's internal message store will be retrieved. When this is complete, the OnGSMComplete event of the TApdGSMPhone component will be generated with the State parameter equal to gsListAll. We will use a TListBox to display the message store, so drop one of those on the form. Create the OnGSMComplete event handler and add the following code to display the time stamps of the messages in the list box.

```
  procedure TForm1.ApdGSMPhone1GSMComplete(Pager:
  TApdCustomGSMPhone; State: TApdGSMStates; ErrorCode: Integer);
  begin
    if State = gsListAll then
      ListBox1.Items.AddStrings(Pager.MessageStore);
  end;
```

In this example we will be displaying the details of the message when the list box is double clicked. Drop three TEdit components, one for the address, one for the timestamp, and one for the status of the message. Drop a TMemo component on the form to display the message text. Feel free to add corresponding TLabel components. Create the OnDblClick event handler for the TListBox and add the following code:

```
procedure TForm1.ListBox1DblClick(Sender: TObject);
var
  I : Integer;
  Msg : TApdSMSMessage;
begin
  if ListBox1.ItemIndex > -1 then begin
    I := ListBox1.ItemIndex;
    Msg := ApdGSMPhone1.MessageStore.Messages[I];
    Edit1.Text := Msg.Address;
    Edit2.Text := Msg.TimeStampStr;
    Edit3.Text := ApdGSMPhone1.StatusToStr(Msg.Status);
    MemoMessage.Text := Msg.Message;
  end;
end;
```

You can just as easily delete messages from the phone, create the OnKeyDown event handler for the TListBox and add the following:

```
procedure TForm1.ListBox1KeyDown(Sender: TObject; var Key: Word;
Shift: TShiftState);
begin
  if Key = VK_DELETE then begin
    if ListBox1.ItemIndex > -1 then
      ApdGSMPhone1.MessageStore.Delete(ListBox1.ItemIndex);
end;
```

You can also use the TEdit components that were previously added to add a new message to the message store. Drop another TButton on the form and change the Caption and Name properties to “AddMessage”, create the OnClick event handler and add the following code:

```
procedure TForm1.AddMessageClick(Sender: TObject);
begin
  { Edit1 is the Address TEdit, Memo1 is the message text }
  ApdGSMPhone1.WriteToMemory(Edit1.Text, Memo1.Text);
end;
```

To send an unsent message from the MessageStore, drop another TButton component on the form, change the Caption and Name properties to “SendFromMemory”, create the OnClick event handler and add the following code:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Msg : TApdSMSMessage;
  I : Integer;
begin
  if ListBox1.ItemIndex > -1 then begin
    I := ListBox1.ItemIndex;
    Msg := ApdGSMPhone1.MessageStore.Messages[I];
    { only send messages that have not been sent }
    if Msg.Status = ssUnsent then
      ApdGSMPhone1.SendFromMemory(ListBox1.ItemIndex);
  end;
end;
```

Related examples

EXSMSPGR.DPR

EXGSM.DPR

Converting a Document to Fax Format

This topic shows how to convert an existing ASCII text, BMP, PCX, DCX or TIFF document to a faxable format.

When faxing a document, the first step is converting it into a faxable format, something that the TApdSendFax component can understand and transmit to the receiving fax machine.

Required components

TApdFaxConverter

Prerequisite topics

None

Related components

TApdFaxUnpacker

TApdSendFax

What to do

If you want to fax an existing document, it must be in an APF, ASCII Text, BMP, PCX, DCX, or TIFF format. The TApdFaxConverter component converts the file into the Async Professional Fax format (APF), which can then be processed by the Async Professional faxing components. The TApdFaxConverter.DocumentFile property specifies the file to be converted. The conversion method depends on the format of the DocumentFile. The TApdFaxConverter.InputDocumentType property indicated the type of file. You can either specify the path and name of the resulting fax file in the OutFileName property or leave the property blank; in that case the file has the same path and name but an extension of “APF.” If you are converting an ASCII text file, you can specify which font to use in the converted APF file in the EnhFont property.

The following example allows the user to select a text file and converts it to the APF format:

```
procedure TForm1.Button1Click(Sender : TObject);
begin
  OpenDialog1.Filter := 'Text files (*.TXT)|*.TXT';
  if OpenDialog1.Execute then begin
    ApdFaxConverter1.DocumentFile := OpenDialog1.FileName;
    ApdFaxConverter1.InputDocumentType := idText;
    ApdFaxConverter1.ConvertToFile;
  end;
end;
```

Related examples

CVT2FAX.DPR

Configuring a Device for Faxing

This topic shows how to select and configure a fax modem for faxing.

The fax protocol requires specific settings for faxes to be transmitted successfully. The TApdComPort component properties are automatically changed from their defaults when a TApdSendFax or TApdReceiveFax component is dropped on the form. The fax protocols are well defined, and the default commands sent by the faxing components work for the majority of fax modems. To ensure reliable operations, the fax modem should be configured explicitly.

Required components

TApdComPort

Prerequisite topics

“Setting Up a Comport” on page 74.

“Selecting and Configuring a Modem” on page 85.

Related components

TApdTapiDevice

What to do

With the TApdTapiDevice component, call the ConfigAndOpen method. Call the appropriate faxing method when the OnTapiPortOpen event fires. The following example configures the device and sends a fax.

```
procedure TForm1.ConfigAndSendBtnClick(Sender : TObject);
begin
  ApdTapiDevice1.ConfigAndOpen;
end;

procedure TForm1.ApdTapiDevice1TapiPortOpen(Sender : TObject);
begin
  ApdSendFax1.StartTransmit;
end;
```

Related examples

SENDFAX.DPR

Sending Faxes to One Recipient

This topic shows how to send a fax to a single recipient and to send several faxes to multiple recipients in the same fax session.

The TApdSendFax component provides the ability to send a fax in your application. The fax to be sent can consist of a single or several documents, with or without a cover page. The TApdSendFax component can send a fax over a dedicated fax call, where the dialing is handled by the component, or it can send a fax over an existing voice connection.

Required components

TApdComPort

TApdSendFax

Prerequisite topics

"Setting Up a Comport" on page 74.

"Converting a Document to Fax Format" on page 140.

"Configuring a Device for Faxing" on page 142.

Related components

TApdFaxStatus

TApdFaxLog

TApdFaxConverter

What to do

The fax document to be sent must be either an ASCII text file with or without replaceable tags or an Async Professional Fax (APF) file. The APF file can be created with the fax printer driver, converted from another document with the TApdFaxConverter component, or created by appending separate APF files into one APF file.

Once you have an APF file on disk, and know where it is, you can fax it. The component that handles the fax handshaking and the transmission of the fax pages is the TApdSendFax component. For simple, single APF file faxing, there are only two TApdSendFax component properties that must be set, FaxFile and PhoneNumber. The FaxFile property specifies the APF file to send, including the full path and file name. The PhoneNumber property is the phone number of the receiving fax machine.

According to FCC regulations, all faxes sent in the United States must have the date/time of transmission, sender's identification, and phone number of the sending fax machine or individual. The TApdSendFax.HeaderLine property is used to overlay this information at the top of each fax page. The font can be changed by setting EnhTextEnabled to True and specifying the font in the EnhHeaderFont property. You can hard-code the overlaid text in the HeaderLine property at run time or design time, or you can use replaceable tags to allow changes at run time. Replaceable tags are codes inserted into the text that are replaced with real information. The Async Professional replaceable tags are two character strings that begin with '$' and are followed by the code. Table 5.5 lists the replaceable tags and their meanings.

Table 5.5: *Replaceable tags*

Tag	Description
$D	Current date in MM/DD/YY format
$I	Station ID
$N	Total number of pages
$P	Current page number
$R	Recipient's name (from HeaderRecipient property)
$F	Sender's name (from HeaderSender property)
$S	Title of fax (from HeaderTitle property)
$T	Current time in HH:MMpm format
$$	Inserts the '$' character

Following is an example of a replaceable tagged HeaderLine that has the date/time of transmission, station ID, and sender's name:

```
ApdSendFax1.HeaderLine := 'Date/Time : $D $T, $I, $F';
```

The TApdSendFax component can also send a cover page as the first page of the fax or as the fax in its entirety. The cover page can be either an ASCII text file or an APF file. Regardless of the file type, set the TApdSendFax.CoverPage property to the file's path and name. The advantage of an ASCII text file as the cover page is that you can use the same replaceable tags that were used in the HeaderLine property in it, the advantage of an APF is that you can include graphics and different fonts in the same document. If you specify an ASCII text file, the replaceable tags are replaced with the actual values and then the document is converted to APF format. The font used for the ASCII text cover page can be changed by setting the EnhTextEnabled property to True and specifying the font in the EnhFont property.

A send-fax call can send only a single APF fax file and cover page. To send several APF files you must append the files together to make a single APF file. To specify the APF files to be appended, use the TApdSendFax.FaxFileList string list property. You can use the ConcatFaxes method to concatenate the APF files in the FaxFileList property if you want to save the appended APF file. If you do not use the ConcatFaxes method, the APF files in the FaxFileList property are concatenated to a temporary file and then deleted when the fax has been sent.

To send the fax, call the TApdSendFax.StartTransmit method. The faxmodem is initialized for faxing and the component dials the number specified in the PhoneNumber property. After the call is answered and the fax devices negotiate a connection, the fax is sent. To provide a visual indication of the progress of the fax call, the TApdFaxStatus component displays a dialog box showing several status indicators. The TApdFaxLog component creates and maintains a text file containing the start and end times of all transmitted faxes.

Once the fax session is complete, either through normal termination or due to an unrecoverable error, the OnFaxFinish event fires. The ErrorCode property of this event has the reason the session ended. You can pass the ErrorCode parameter to the ErrorMsg function to get a text version of the message.

The following example gets a file name from a TOpenDialog component and then sends the fax. This example handles the selection of either a single or several APF files. Once the transmission is complete, the result of the fax sessions is shown.

```
procedure TForm1.SelectSendClick(Sender : TObject);
begin
  OpenDialog1.Filter := 'APF Files (*.APF)|*.APF';
  OpenDialog1.Options := [ofAllowMultiSelect];
  if OpenDialog1.Execute then begin
    ApdSendFax1.FaxFileList.Assign(OpenDialog1.Files);
    ApdSendFax1.StartTransmit;
  end;
end;

procedure TForm1.ApdSendFax1FaxFinish(
  CP : TObject; ErrorCode : Integer);
begin
  ShowMessage(ErrorMsg(ErrorCode));
end;
```

In addition to using the TApdSendFax component to place a dedicated fax call, the component can use a call already in progress. The StartManualTransmit method takes over the call when it is called and transmits the cover page and fax file specified in the appropriate properties. The following example sends a fax over an existing voice connection.

```
procedure TForm1.SendFaxNowBtnClick(Sender : TObject);
begin
  ApdSendFax1.FaxFile := 'REPORT.APF';
  ApdSendFax1.StartManualTransmit;
end;
```

Related examples

None

Sending Faxes to Different Recipients

This topic shows how to send a single fax to several recipients or several faxes to different recipients.

Some fax-enabled applications allow scheduling several faxes to one or more recipients or broadcasting the same fax to many recipients. The TApdSendFax component can initiate a single fax session and place multiple fax calls during that session.

Required components

TApdComPort

TApdSendFax

Prerequisite topics

“Setting Up a Comport” on page 74.

“Configuring a Device for Faxing” on page 142.

“Sending Faxes to One Recipient” on page 143.

Related components

TApdFaxStatus

TApdFaxLog

What to do

The task of sending a single fax to a single recipient is easily expanded to sending the same fax to multiple recipients or sending different faxes to different recipients in the same fax session by using the TApdSendFax.OnFaxNext event. This event fires before placing each separate fax call in the fax session. The parameters passed to the event are CP (the TApdComPort component that owns the session), APhoneNumber (the phone number of the next fax recipient), AFaxFile (the APF path and file name of the next fax), and ACoverFile (the cover file of the next fax). The TApdSendFax component keeps sending faxes as long as valid information is returned in the APhoneNumber and AFaxFile parameters.

The OnFaxNext event fires before each fax call is placed and ignores the FaxFile, FaxFileList, PhoneNumber, and CoverPage properties of the TApdSendFax component. The APF files specified in the FaxFileList property must be appended prior to starting the fax call since the AFaxFile parameter does not accept a TStringList. To manually append several APF files, add them to the FaxFileList property and then call the TApdSendFax.ConcatFaxes method. Pass a temporary file name in the ConcatFaxes method and use that file name in the OnFaxNext event. The following example appends several faxes selected with a TOpenDialog component. Once this method is complete, the APF file specified in the ConcatFaxes parameter can be used in the OnFaxNext event.

```
procedure TForm1.PrepareFaxesToBroadcast;
begin
  OpenDialog1.Filter := 'APF Files (*.APF)|*.APF';
  OpenDialog1.Options := [ofAllowMultiSelect];
  if OpenDialog1.Execute then begin
    ApdSendFax1.FaxFileList.Assign(OpenDialog1.Files);
    ApdSendFax1.ConcatFaxes('C:\FAXES\OUTFAX.APF');
  end;
end;
```

Since all fax calls are placed in the same fax session, the OnFaxFinish event fires after the last fax is sent or an unrecoverable error occurs. To keep track of each fax call, use the TApdSendFax.OnFaxLog event. This event fires at the start and finish of each individual fax call in the fax session.

The following example iterates through a database, sends all specified faxes, and flags the database record, depending on whether or not the fax was successful.

```
procedure TForm1.StartSendingBtnClick(Sender : TObject);
begin
  ApdSendFax1.HeaderSender := 'Joe Cool';
  ApdSendFax1.StationID := '719 260 7151';
  Table1.First;
  if not Table1.EOF then
    ApdSendFax1.StartTransmit;
end;

procedure TForm1.ApdSendFax1FaxNext(
  CP : TObject; var APhoneNumber, AFaxFile,
  ACoverFile : TPassString);
```

```
begin
  Table1.Next;
  if not Table1.EOF then begin
    ApdSendFax1.HeaderRecipient :=
      Table1.FieldByName('RecipientName').AsString;
    ApdSendFax1.HeaderTitle :=
      Table1.FieldByName('Title').AsString;
    APhoneNumber := Table1.FieldByName('PhoneNumber').AsString;
    AFaxFile := Table1.FieldByName('APFName').AsString;
    ACoverFile := ApdSendFax1.CoverFile;
  end;
end;

procedure TForm1.ApdSendFax1FaxLog(
  CP : TObject; LogCode : TFaxLogCode);
begin
  case LogCode of
    lfaxTransmitStart : Table1.FieldByName(
      'Status').AsString := 'Sending';
    lfaxTransmitOK : Table1.FieldByName(
      'Status').AsString := 'Sent OK';
    lfaxTransmitFail : Table1.FieldByName(
      'Status').AsString := 'Failed';
  end;
end;
```

Related examples

SENDFAX.DPR

Receiving Faxes

This topic shows how to receive a single fax or multiple faxes.

Receiving documents over telephone lines is a time-saving process that has nearly revolutionized modern business practices. Getting a document from across country used to take days; now can be done in minutes. The TApdReceiveFax component gives your application this efficiency.

Required components

TApdComPort

TApdReceiveFax

Prerequisite topics

“Setting Up a Comport” on page 74.

“Configuring a Device for Faxing” on page 142.

Related components

TApdFaxStatus

TApdFaxLog

What to do

The TApdReceiveFax component is designed to receive faxes manually or automatically from a single fax call or throughout an entire fax session. A single receive fax call starts with the TApdReceiveFax component monitoring the activities of its associated TApdComPort When an incoming call is detected, the component answers the call, negotiates, and receives the fax. The fax call ends when the connection is dropped, due to either the end of the fax document or an unrecoverable error. Manual fax reception can be done from a new incoming call or during a voice call that is already in progress. A receive fax session starts with the TApdReceiveFax component monitoring for the incoming call and ends when the TApdReceiveFax component monitoring is cancelled.

The TApdReceiveFax.StationID property defines the identification string that is sent to the sending fax machine. The sending fax machine usually displays this information so the sender can verify the fax is being received by the right location. Common values for the StationID property are the phone number or business name.

When the fax is being received, the fax page data is saved to an Async Professional Fax (APF) file. The naming scheme used to name the received APF file will depend on your application's requirements. There are three ways to name the incoming fax:

1. Set the FaxNameMode property to fnCount. This uses a sequential numbering system to name the received faxes (FAX0001.APF, FAX0002.APF, and so on).
2. Set the FaxNameMode property to fnMonthDay. This uses the month and day the fax is received along with a sequential numbering system (02100001.APF {Feb 10, 1st fax of the day}, 02100002.APF, and so on).
3. Create your own naming scheme with the OnFaxName event. You can also specify the directory/folder the received in which faxes are stored with the DestinationDir property.

Now that you know where the received fax will be stored and what they will be named, you can start receiving faxes. The TApdReceiveFax.OneFax property determines if the fax session receives a single or several faxes. If the OneFax property is True, the next call is answered and the fax received. After that, the TApdReceiveFax component does not answer any further calls. If OneFax is False, the TApdReceiveFax component answers and receives faxes until the TApdReceiveFax.CancelFax method is called or there is an unrecoverable error. In either case, to resume receiving faxes, call the TApdReceiveFax.StartReceive method. The TApdReceiveFax component monitors the TApdComPort component until the number of rings specified in the AnswerOnRing property are detected, then it answers the call and receives the fax.

The OnFaxLog event fires when the incoming call is verified to be a fax call and when the fax call terminates, whether as the result of a successful transmission or an unrecoverable error. If OneFax is True, the OnFaxFinish event fires after the single fax is received. If OneFax is False, the OnFaxFinish event fires after CancelFax is called or an unrecoverable error occurs.

The following example opens the port, saves the received faxes in a specific directory, uses the fnCount method of naming a file, records the start and end of each fax call, and displays a message when the fax session is finished.

```
procedure TForm1.Button1Click(Sender : TObject);
begin
  ApdComPort1.Open := True;
  ApdReceiveFax1.DestinationDir := 'C:\RECVFAX';
  ApdReceiveFax1.FaxNameMode := fnCount;
  ApdReceiveFax1.StationID := '719 260 7151';
  ApdReceiveFax1.StartReceive;
end;

procedure TForm1.ApdReceiveFax1FaxFinish(
  CP : TObject; ErrorCode : Integer);
begin
  ShowMessage('Fax received : ' + ErrorMsg(ErrorCode));
end;

procedure TForm1.ApdReceiveFax1FaxLog(
  CP : TObject; LogCode : TFaxLogCode);
begin
  case LogCode of
    lfaxReceiveStart : Memo1.Lines.Add(
      'Receiving ' + ApdReceiveFax1.FaxFile);
    lfaxReceiveOK : Memo1.Lines.Add(
      ApdReceiveFax1.FaxFile + ' OK');
    lfaxReceiveFail  : Memo1.Lines.Add(
      ApdReceiveFax1.FaxFile + ' failed');
  end;
end;
```

To begin a manual fax reception, use the TApdReceiveFax.StartManualReceive method. This method has one parameter, SendATAToModem, which determines whether or not the faxmodem has to answer the call. If the call is already in progress, SendATAToModem should be False, if the phone has not yet been answered, the property should be set to True. The following example uses a button to begin receiving a fax over an existing voice connection:

```
procedure TForm1.ReceiveFaxNowBtnClick(Sender : TObject);
begin
  ApdReceiveFax1.StartManualReceive(False);
end;
```

Related examples

RCVFAX.DPR

Converting a Fax to Another Format

This topic shows how to convert an existing fax file into a BMP, PCX, DCX or TIFF document.

The APF format is good for sending and receiving faxes, but what about doing something else with the fax file? The APF format is a proprietary format, based on the Group 3 standard. For it to be of use outside the context of the Async Professional components, it must be converted to a format that is more widely understood.

Required components

TApdFaxUnpacker

Prerequisite topics

None

Related components

TApdFaxConverter

TApdReceiveFax

What to do

The first step in converting an APF file to another file type is to have an APF file to convert. Most likely this is a received fax, but you can use an APF created with the TApdFaxConverter component, or the APROLOGO.APF file in the EXAMPLES\DELPHI or EXAMPLES\BUILDER directory, or an APF file created by the fax printer driver. Set the TApdFaxUnpacker.InFileName property to the APF file's path and file name. Next, determine the format for the resulting file. The method to convert the document is UnPackFileToXxxx, where Xxxx is the format. If you want to convert just a single page in the APF file, use the UnPackPageToXxxx methods. The resulting file can be specified with the OutFileName property. If you leave the property blank, the resulting file has the same path and name as the APF file but an extension appropriate to the new file's format.

The following example converts a user-selected APF file to a bitmap file and then converts the same APF file into a TIFF file:

```
procedure TForm1.ConvertBtnClick(Sender : TObject);
begin
  OpenDialog1.Filter := 'APF Files (*.APF)|*.APF';
  if OpenDialog1.Execute then begin
    ApdFaxUnpacker1.InFileName := OpenDialog1.FileName;
    ApdFaxUnpacker1.OutFileName := 'C:\FAXIMAGE.BMP';
    ApdFaxUnpacker1.UnpackFileToBitmap;
    ApdFaxUnpacker1.OutFileName := 'C:\FAXIMAGE.TIF';
    ApdFaxUnpacker1.UnpackFileToTiff;
  end;
end;
```

Alternatively, you can use the APF TPicture registration to handle the fax conversion. To use this technique, first make sure that the AdFaxCvt unit is in the uses clause of your application. Then drop a TImage and a TOpenPictureDialog component on your form. Add a new filter of "*.apf" with a filter name of "APF Files (*.apf)" to the Filter property of the TOpenPictureDialog component. Drop a button on the form and add the following code to its OnClick event:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenPictureDialog1.Execute then
    Image1.Picture.LoadFromFile (OpenPictureDialog1.FileName);
end;
```

This code will allow you to load an APF file and view it on the TImage component.

To save the fax image in a different format, drop a SavePictureDialog and another button on the form. On the button's OnClick event, add the following code:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  if SavePictureDialog1.Execute then
    Image1.Picture.SaveToFile (SavePictureDialog1.FileName);
end;
```

Related examples

None

Viewing a Fax

This topic shows how to view an APF file.

Once you convert a document to the Async Professional Fax file format (APF) or receive a fax you will want to view it. The APF file is a Group 3 fax, but it is in a proprietary format so you'll need to use the TApdFaxViewer component.

Required components

TApdFaxViewer

Prerequisite topics

None

Related components

TApdFaxConverter

TApdReceiveFax

What to do

The TApdFaxViewer component is designed to display APF files, to process keystrokes to navigate through file, and to scale and rotate the fax image. The first step is to drop the TApdFaxViewer component on a form and then specify the name of the APF file to view in the TApdFaxViewer.FileName property. The fax file is loaded and displayed in the component.

Navigation through the fax is achieved by keystrokes from the user or via code. Use the FirstPage, NextPage, PrevPage and LastPage methods to display the appropriate pages of the fax file. Selected portions of the fax image can be copied to the clipboard in bitmap format with the CopyToClipboard method. The following example selects, loads, and displays an APF file, navigates to the last page, and copies it to the clipboard:

```
procedure TForm1.ViewAndCopyLastPage;
begin
  OpenDialog1.Filter := 'APF Files (*.APF)|*.APF';
  if OpenDialog1.Execute then begin
    ApdFaxViewer1.FileName := OpenDialog1.FileName;
    ApdFaxViewer1.LastPage;
    ApdFaxViewer1.SelectImage;
    ApdFaxViewer1.CopyToClipboard;
  end;
end;
```

Related examples

VIEWFAX.DPR

Printing a Fax

This topic shows how to print an APF file.

The ability to print a fax file is almost a necessity for most fax applications. The TApdFaxPrinter component provides this functionality.

Required components

TApdFaxPrinter

Prerequisite topics

None

Related components

TApdFaxPrinterStatus

TApdFaxPrinterLog

TApdFaxConverter

TApdFaxViewer

TApdReceiveFax

What to do

To print a fax, drop a TApdFaxPrinter component on the form, specify the file name in the TApdFaxPrinter.FileName property, and then call the TApdFaxPrinter.PrintFax method. This sends the specified APF file to the currently selected printer. To change which printer is used, call the TApdFaxPrinter.PrintSetup method. You can also specify a print range with the FirstPageToPrint and LastPageToPrint properties. This technique is adequate for most applications, but there are a few things that can be added or changed to improve the results.

The TApdFaxPrinterStatus component displays a dialog showing the progress of the print job. The TApdFaxPrinterLog component creates a text file that shows the time each print job was started and finished. The FaxFooter and FaxHeader properties add a footer and header on the printed page to provide additional information.

The following example selects an APF file to print and allows the user to select which printer is used to print the fax:

```
procedure TForm1.LoadAndPrintBtnClick(Sender : TObject);
begin
  OpenDialog1.Filter := 'APF Files (*.APF)|*.APF';
  if OpenDialog1.Execute then begin
  ApdFaxPrinter1.FileName := OpenDialog1.FileName;
  ApdFaxPrinter1.PrintSetup;
  ApdFaxPrinter1.PrintFax;
end;
```

Related examples

None

Installing the Fax Printer Driver Programmatically

This topic shows how to programmatically install the Async Professional fax printer driver.

The Async Professional fax printer driver operates the same as most other Windows printer drivers. Print jobs from Windows applications are converted to a bitmap and then sent to the printer driver. The printer driver converts the bitmap to a format the printer can understand. For this to work, the printer driver must be installed correctly.

Required components

Async Professional Fax Printer Driver

Prerequisite topics

None

Related components

None

What to do

There are a two techniques that can be used to install a printer driver, manually and programmatically. The manual technique relies on properly written INF files and the experience of the user. The programmatic technique installs the printer driver automatically for the user, and can react to any problems that occur during the installation. Installing the printer driver from your code also gives your application a more polished appearance.

The grunt work of installing the fax printer driver is done in the PDRVINST.PAS unit. This unit has two methods that install the printer driver. Which one is used depends on the operating system on which the driver is being. To assist your installation program in determining the operating system, the IsWinNT method in PDRVINST.PAS returns True if running under Windows NT and False if it is not. For Windows NT/2000 installations, call the InstallDriver32 method; for Windows 95/98/ME, call the InstallDriver method. These methods install the fax printer driver and return a result code in the DrvInstallError variable. The possible result codes are described in the PINST.DPR example project.

The following example determines if the application is running under Windows NT, calls the appropriate installation method, and, when done, displays the results of the installation:

```
procedure TForm1.InstallFaxPrinterDriver;
begin
  if IsWinNT then
    InstallDriver32('')
  else InstallDriver('APFGEN.DRV');
  case DrvInstallError of
    ecOK : ShowMessage('Printer driver installed successfully');
    ecDrvDriverNotFound : ShowMessage(
      'Printer driver not found');
    else
      ShowMessage('Other installation error : ' +
        IntToStr(DrvInstallError));
  end;
end;
```

Related examples

PINST.DPR

Intercepting a Fax Printer Print Job

This topic shows how to detect a print job sent to the Async Professional fax printer driver.

The Async Professional fax printer driver converts Windows print jobs from any Windows application to the Async Professional Fax (APF) file format. The fax printer driver generates callbacks when the print job starts and when it is complete. To fax the APF file, or to save it, you must detect the print job callbacks.

Required components

TApdFaxDriverInterface

Prerequisite topics

“Installing the Fax Printer Driver Programmatically” on page 159.

Related components

TApdSendFax

TApdFaxUnpacker

What to do

The TApdFaxDriverInterface component receives messages from the Async Professional fax printer driver when the print job is received by the driver and when the print job processing is complete. The OnDocStart event of the TApdFaxDriverInterface component fires when a print job is sent to the fax printer driver from any Windows application. When the fax printer driver has completed the conversion of the print job to an APF file, the OnDocEnd event fires. The FileName property determines the path and name of the resulting APF file.

To intercept the print job callbacks from the Async Professional fax printer driver, a TApdFaxDriverInterface component must be instantiated on the system. This is done by dropping the component on an application that is running when the print job starts. The application with the TApdFaxDriverInterface component can be a Tray icon, minimized, hidden, or an inactive window, etc. What’s critical is that TApdFaxDriverInterface component is created before the print job starts. The actual print job is nothing more than converting the document into a Device Independent Bitmap (DIB) and then converting that to an APF. When a print job is sent to the fax printer driver, the TApdFaxDriverInterface.OnDocStart event fires. The OnDocStart event should be used to set the path and file name of the resulting APF. When the print job is complete, the TApdFaxDriverInterface.OnDocEnd event fires. The OnDocEnd event signals when the APF is complete and ready for further processing.

The following example intercepts the fax printer driver print jobs, saves the resulting APF in a specific path, and then sends the fax:

```
procedure TForm1.ApdFaxDriverInterface1DocStart(
  Sender : TObject);
begin
  ApdFaxDriverInterface1.FileName := 'C:\FAX\SENDME.APF';
end;

procedure TForm1.ApdFaxDriverInterface1DocEnd(Sender : TObject);
begin
  ApdSendFax1.FaxFile := ApdFaxDriverInterface1.FileName;
  ApdSendFax1.PhoneNumber := '260-7151';
  ApdSendFax1.StartTransmit;
end;
```

Related examples

FAXMON.DPR

FXCLIENT.DPR

Faxing a Document from your Application

This topic shows how to fax a document from within your application using the fax printer driver.

If you are creating documents to fax from within your application, there are two techniques you can use: convert the document to the Async Professional Fax file format (APF) then process it, or print the document to the Async Professional fax printer driver.

Required components

TApdComPort

TApdFaxDriverInterface

TApdSendFax

Prerequisite topics

“Setting Up a Comport” on page 74.

“Configuring a Device for Faxing” on page 142.

“Sending Faxes to One Recipient” on page 143.

“Intercepting a Fax Printer Print Job” on page 161.

Related components

TApdFaxStatus

TApdFaxLog

What to do

In order to fax a document using the Async Professional fax components, the document must be in the Async Professional Fax file format (APF). If the current document format is one of the formats supported by the TApdFaxConverter component, the document can be converted without the fax printer drivers. If the format is not one handled by the TApdFaxConverter, you must convert it using the fax printer driver.

The process of converting and faxing a document in a format supported by the TApdFaxConverter component begins with converting the document to the APF file format with the TApdFaxConverter.ConvertToFile method. The method does not return until after the document has been converted; therefore, the call to TApdSendFax.StartTransmit can be made immediately afterwards in your code (i.e., you don't have to force a delay or wait for an event to fire).

The following example saves the contents of a TMemo component, converts the saved document to the APF file format, and then faxes the APF file:

```
procedure TForm1.ConvertAndFaxBtnClick(Sender : TObject);
begin
  Memo1.Lines.SaveToFile('C:\OUTFAX.TXT');
  ApdFaxConverter1.DocumentFile := 'C:\OUTFAX.TXT';
  ApdFaxConverter1.InputDocumentType := idText;
  ApdFaxConverter1.OutFileName := 'C:\OUTFAX.APF';
  ApdFaxConverter1.ConvertToFile;
  ApdSendFax1.FaxFile := ApdFaxConverter1.OutFileName;
  ApdSendFax1.PhoneNumber := '260 7151';
  ApdSendFax1.StartTransmit;
end;
```

If you are printing from your application to the Async Professional fax printer driver, you must correctly select the fax printer driver programmatically. The TPrinter object contains the printers that are installed on the system. It would seem a simple matter to select the appropriate printer from the TPrinter.Printers string list. The difficulty comes from the way the printer canvas is configured once a new printer is selected. When a new printer is selected with the TPrinter object, the capabilities of the default Windows printer are maintained in the new printer's canvas. This behavior is not limited to the Async Professional fax printer driver, but can be seen in all printers selected this way. To fix the problem, use the GetPrinter and SetPrinter API methods. Once the printer is correctly selected and configured, print the document to the Async Professional printer driver with the same methods used to print to any other Windows printer driver. If required, use the TApdFaxDriverInterface component to intercept the print job so the resulting APF file can be further processed or faxed.There are also two Async Professional fax printer driver, one for Windows 95/98/ME and another for Windows NT/2000. These have different names, so you will want to look for either of the printers. In Windows 95/98/ME, the printer is named "Print to Fax on PRINTFAX:". In Windows NT/2000, the printer is named "APF Fax Printer."

The following example shows how to find the Async Professional fax printer driver and correctly select it with the Delphi TPrinter object:

```
with Printer do begin
  I := PrinterIndex;
  P := Printers.IndexOf('Print to Fax on PRINTFAX:');
  if P = -1 then
    P := Printers.IndexOf('APF Fax Printer');
  if P = -1 then begin
    ShowMessage(
      'The TurboPower Fax Printer was not found'#13#10+
      'The print job will not be submitted');
    Exit;
  end else begin
    PrinterIndex := P;
    GetPrinter(Device, Name, Port, DevMode);
    SetPrinter(Device, Name, Port, 0);
  end;
```

Once the printer is correctly selected and configured, print the document to the Async Professional printer driver with the same methods used to print to any other Windows printer driver. If required, use the TApdFaxDriverInterface component to intercept the print job so the resulting APF file can be further processed or faxed.

Related examples

None

Creating a Fax Client

This topic shows how to create a functional TApdFaxClient component.

The fax server components in Async Professional need fax jobs to process; otherwise, they will just be a pretty icon on the component palette. To create fax jobs you will need the TApdFaxClient component.

Required components

TApdFaxClient

Prerequisite topics

“Overview: Using the Fax Server Components” on page 50.

“Intercepting a Fax Printer Print Job” on page 161.

Related components

TApdFaxServer

TApdFaxServerManager

TApdFaxDriverInterface

What to do

The TApdFaxClient component creates fax jobs. The properties of the TApdFaxClient component are used for the fields in the job header, which applies to all recipients of the job; and the recipient header, which applies to a specific recipient. A simple fax client can be created that intercepts fax printer print jobs and makes a fax job file (APJ), ready for submission to a TApdFaxServerManager.

For the sake of clarity, we’ll do a simple single-recipient fax job each time a print job is sent to our fax printer driver. Create a new project and drop a TApdFaxDriverInterface, TApdFaxClient, and a TButton on the form. Now, we’ll need a way for the user to enter the minimum information about the job and recipient. Drop two TEdit components on the form, and name them “edtRecipient” and “edtPhone”, with a corresponding TLabel for each. Change the caption of the edtRecipient’s label to “Recipient name”; and the caption of edtPhone’s label to “Phone number”. Change the caption of the button to “OK.”

The TApdFaxDriverInterface.OnDocEnd event will fire when the print job is complete, so we'll do some of the background work there. We will use the name of the document that was just printed as the FaxJobName, and will use the APF that the printer driver just created as the FaxFileName. Make your OnDocEnd event look something like the following:

```
procedure TForm1.ApdFaxDriverInterface1DocEnd(Sender:  TObject);
begin
  edtRecipient.Text := '';
  edtPhone.Text := '';
  ApdFaxClient1.CoverFileName := '';
  ApdFaxClient1.FaxFileName := ApdFaxDriverInterface1.FileName;
  ApdFaxClient1.HeaderLine := '$S sent by $F to $R on $D $T';
  ApdFaxClient1.HeaderTitle := ApdFaxDriverInterface1.DocName;
  ApdFaxClient1.JobFileName := 'C:\Faxes\' + NextJobFileName;
  ApdFaxClient1.JobName := ApdFaxDriverInterface1.DocName;
  ApdFaxClient1.ScheduleDateTime := Now;
  ApdFaxClient1.Sender := 'Me';
end;
```

Everything is set up now, except for the recipient's name and phone number. Use the OK button's OnClick event to retrieve that information from the edit controls and create the job file.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ApdFaxClient1.HeaderRecipient := edtRecipient.Text;
  ApdFaxClient1.PhoneNumber := edtPhone.Text;
  ApdFaxClient1.MakeFaxJob;
end;
```

That's all there is to it, you've just created a fax client.

There are a few things about this code that could use a bit more explanation. The HeaderLine is using replaceable tags. $S will be replaced by the ApdFaxClient.HeaderTitle property; $F will be replaced by the ApdFaxClient.Sender property; $R will be replaced by the ApdFaxClient.HeaderRecipient property; $D and $T will be replaced by the date and time that the fax is sent. This example does not do it, but you could use the same technique to customize a cover file.

The ApdFaxClient.JobFileName property assignment uses a NextJobFileName function, which is not defined here. The purpose of this function is to return a unique name for the fax job file.

You could expand upon this example by letting the user enter several recipients. To add another recipient, call the TApdFaxClient.AddRecipient method.

Related examples

FXCLIENT.DPR

FXSRVR.DPR

Setting Up a Fax Server

This topic shows how to set up a TApdFaxServer component and TApdFaxServerManager component.

Fax servers must be able to receive, send, and schedule faxes. The Async Professional fax server components are there to make your job easier by handling the integration for you.

Required components

TApdComPort

TApdFaxServer

TApdFaxServerManager

Prerequisite topics

“Overview: Using the Fax Server Components” on page 50.

“Creating a Fax Client” on page 166.

Related components

TApdFaxClient

TApdSendFax

TApdReceiveFax

What to do

Fax documents used with the Async Professional fax server components are in the Async Professional Fax Job format (APJ). This format embeds information about the entire fax job, information about individual recipients of the fax, cover file text, and the actual APF data.

The TApdFaxServer component connects to the faxmodem through the TApdComPort (or optionally through the TApdTapiDevice).

To enable monitoring for incoming faxes, set the TApdFaxServer.Monitoring property to True. The TApdFaxServer will answer incoming calls, negotiate fax session parameters, and receive the fax. The directory where received faxes will be saved is specified by the TApdFaxServer.DestinationDir property. You can set the FaxNameMode property to fnCount and the faxes will be named sequentially; or set that property to fnMonthDay and the faxes will be named according to the date and time that the fax was received. The OnFaxServerFinish event will fire for each fax received successfully; or the OnFaxServerFatalError event will fire if the fax failed.

Scheduling and sending faxes is a bit more involved, and requires a TApdFaxServerManager. The TApdFaxServerManager component manages a specific directory for fax jobs. The directory that the TApdFaxServer component manages is specified by the MonitorDir property. (The fax jobs can be created by the TApdFaxClient and copied to the monitored directory). Any given directory can be monitored by only one TApdFaxServerManager component. If more than one TApdFaxServerManager components try to look at the same directory, an ecAlreadyMonitored exception will be raised. The TApdFaxServer component requests fax jobs from the TApdFaxServerManager component at intervals specified by the TApdFaxServer.SendQueryInterval property. If that property (measured in seconds) is 0, the TApdFaxServer will not ask for fax jobs. The OnFaxServerFinish event will fire when a fax was sent successfully, and the OnFaxServerFatalError event will fire if the fax failed.

Create a new project and drop a TApdComPort component on the form. Set the ComNumber property as required for your faxmodem. Next, drop a TApdFaxServer component on the form and set the Monitoring property to True, the SendQueryInterval property to 60, and the DestinationDir property to the directory where you want received faxes to be stored. This will make the TApdFaxServer component listen for incoming faxes, and look for new fax jobs every 60 seconds. Drop a TApdFaxServerManager on the form and set the MonitorDir property to a directory where you want to hold pending fax jobs. Compile and run the project. Incoming faxes will be automatically received into the directory specified by the DestinationDir property. To send faxes, create the fax job file using the TApdFaxClient component and copy the APJ file to the directory specified by the TApdFaxServerManager.MonitorDir property.

Related examples

FXCLIENT.DPR

FXSRVR.DPR

Sending and Receiving Faxes with TApdFaxServer

This topic shows how to configure the Async Professional Fax Server and how to use it for receiving and sending faxes.

The TApdFaxServer component is the faxing engine for the Fax Server Components, and handles the physical communication with the fax modem to send and receive faxes. Since this component can both transmit and receive faxes, it shares many properties with the ApdSendFax and ApdReceiveFax components.

Required components

TApdFaxServer

TApdFaxServerManager

Prerequisite topics

“Setting Up a Comport” on page 74.

“Selecting and Configuring a Modem” on page 85.

Related components

TApdAbstractFax

TApdComPort

TApdReceiveFax

TApdSendFax

What to do

Configuration for faxing

The TApdFaxServer component is similar to TApdAbstractFax in that it accesses the physical faxmodem through the ApdComPort component. Set the ApdComPort.ComNumber property to the port enumeration of the faxmodem that you wish to use. If an ApdTapiDevice is assigned to the TapiDevice property, the port will be opened through the TAPI interface. When a fax is ready to be sent, or the component is monitoring for incoming faxes, the following TApdComPort properties are forced to these values:

```
ApdComPort.DataBits := 8;
ApdComPort.StopBits := 1;
ApdComPort.Parity := pNone;
ApdComPort.Baud := 19200;
ApdComPort.InSize := 8192;
ApdComPort.OutSize := 8192;
ApdComPort.HWFlowOptions := [hwfUseRTS, hwfRequireCTS];
```

The faxmodem will be configured with the same configuration string as the TApdAbstractFax. If your modem requires special configuration, set the ModemInit property to your special configuration.

Receiving faxes

The ApdFaxServer component receives faxes by monitoring for incoming calls. To begin monitoring, set the Monitoring property to True. To stop monitoring, set the Monitoring property to False. When an incoming fax is detected, the call is answered, and the received fax is saved to the DestinationDir folder, and named according to the FaxNameMode property. If the call is a fax call, and the fax is successfully received, the OnFaxServerFinish event will fire; otherwise, the OnFaxServerFatalError event will fire. If the fax was successful, the ApdFaxServer will continue monitoring for new calls until Monitoring is explicitly set to False. If the call was unsuccessful, all faxing operations are disabled so the problem can be addressed. The resulting fax file is in the standard Async Professional APF format.

Sending faxes

The ApdFaxServer component sends faxes by querying an ApdFaxServerManager component for fax jobs. Call the ForceSendQuery method to query the TApdFaxServerManager component manually. Set the SendQueryInterval property to a non-zero number to enable automatic querying. To stop querying for new fax jobs set the SendQueryInterval property to 0. Every SendQueryInterval seconds, the ApdFaxServerManger will be queried for fax jobs that are ready to be sent. If a fax job is ready, it will be sent immediately. Since the TApdFaxServerManager.MonitorDir is scanned each time a query is made, it is a good idea to keep SendQueryInterval relatively large (30 seconds or above). If the fax is successfully sent the OnFaxServerFinish event will fire. If the fax was unsuccessful, the OnFaxServerFatalError event will fire and all faxing operations will be disabled so the problem can be addressed. The fax job file is in the Async Professional APJ format, which is discussed fully in Chapter 16 of the Async Professional Reference Guide.

Related examples

FXSRVR.DPR

Detecting DTMF

This topic shows how to detect Dual Tone Modulation Frequencies (DTMF).

DTMF tones are the tones that are generated each time a phone number button is pressed. These tones are often used in automated voice mail, information gathering, fax on demand, and fax back applications.

Required components

TApdComPort

TApdTapiDevice

Prerequisite topics

“Setting Up a Comport” on page 74.

“Selecting and Configuring a Modem” on page 85.

Related components

None

What to do

The DTMF detection requires a voice modem and a voice-capable TAPI service provider. If your setup supports this, you can detect which keys are pressed at the telephone key pad during a connection.

The EnableVoice property of the TApdTapiDevice component determines whether the modem makes a connection in voice mode or data mode. DTMF tones are available only in voice mode, so EnableVoice must be True. When a DTMF tone is detected, the TApdTapiDevice.OnTapiDTMF event fires, passing the key that was pressed in the Digit parameter. There are twelve keys on a telephone number pad; they are passed as ‘0’ through ‘9’, ‘*’ or ‘#’.

The following example answers incoming phone calls in voice mode and display the keys that the caller presses in an edit control:

```
procedure TForm1.Button1Click(Sender : TObject);
begin
  ApdTapiDevice1.SelectDevice;
  ApdTapiDevice1.EnableVoice := True;
  ApdTapiDevice1.AutoAnswer;
end;

procedure TForm1.ApdTapiDevice1TapiDTMF(
  CP : TObject; Digit : Char; ErrorCode : Integer);
begin
  Edit1.Text := Edit1.Text + Digit;
end;
```

Related examples

EXVOICE.DPR

Recording a WAVE File

This topic shows how to record a WAVE file through a TAPI device.

The ability to record voice messages is a vital feature in any voice mail application, and is a good extra for other communications programs.

Required components

TApdComPort

TApdTapiDevice

Prerequisite topics

“Setting Up a Comport” on page 74.

“Selecting and Configuring a Modem” on page 85.

Related components

None

What to do

WAVE file recording requires a voice modem and a voice-capable TAPI service provider. If your setup supports this, you can record incoming messages in WAVE file format.

The EnableVoice property of the TApdTapiDevice component determines whether or not the voice extensions of TAPI are enabled. Since WAVE recording depends on the TAPI voice extensions, set EnableVoice to True. Once you have determined when to begin recording the WAVE file, call the TApdTapiDevice.StartWaveRecord method. The incoming sounds, be they voice, tones, whistling, or dogs barking, are stored in a wave input buffer. The size of the wave input buffer is specified by the MaxMessageLength property and is the number of seconds of wave data to record. Once the wave input buffer is full, or the StopWaveRecord method is called, the OnTapiWaveNotify event fires with the Msg parameter equal to waDataReady, letting you know the wave data is ready to be processed. To save the contents of the wave input buffer, call the SaveWaveFile method. The FileName parameter is the name of the file to be saved, while the Overwrite parameter determines whether an existing file of the same name is replaced with the new file.

The following example records a message from a voice connection in response to a button click event and then saves the recorded wave data once the input buffer is full:

```
procedure TForm1.RecordMessageBtnClick(Sender : TObject);
begin
  ApdTapiDevice1.StartWaveRecord;
end;

procedure TForm1.ApdTapiDevice1TapiWaveNotify(CP : TObject;
  Msg : TWaveMessage);
begin
  if Msg = waDataReady then
    ApdTapiDevice1.SaveWaveFile('F:\RECORD.WAV', True);
end;
```

Related examples

EXRECORD.DPR

Playing WAVE Files

This topic shows how to play a WAVE file over a TAPI device.

An automated voice system requires voice prompts to be sent to the other side of the connection to let the person know what actions are available or required. This is often seen in voice mail systems where the caller is given the choice to press a number on their telephone key pad to select an extension, voice mailbox, or further menu selections.

Required components

TApdComPort

TApdTapiDevice

Prerequisite topics

“Setting Up a Comport” on page 74.

“Selecting and Configuring a Modem” on page 85.

Related components

None

What to do

Playing a WAVE file through a TAPI device requires a voice modem with a valid wave output driver, and a voice-capable TAPI service provider. If your setup supports this, you can play WAVE files over the phone to provide voice prompts or other information.

The EnableVoice property of the TApdTapiDevice component determines whether or not the voice extensions of TAPI are enabled. Since playing WAVE files depends on the TAPI voice extensions, set EnableVoice to True.

The TApdTapiDevice.PlayWaveFile method starts playing the wave file specified in the FileName parameter of the method. The OnTapiWaveNotify event fires when the wave file is initially opened and the Msg parameter is set to waPlayOpen. When the wave file is closed, the Msg parameter is set to waPlayClose. If the wave file has finished playing, the Msg parameter is set to waPlayDone.

The following example plays a wave file when a connection is made:

```
procedure TForm1.ApdTapiDevice1TapiConnect(Sender : TObject);
begin
  ApdTapiDevice1.PlayWaveFile('GREETING.WAV');
end;
```

Related examples

EXVOICE.DPR

Installing SAPI4

This topic shows how to install SAPI 4.

The TApdSapiEngine component utilizes the services of SAPI (Speech API) version 4. Microsoft has released SAPI 5, but there are several reasons why APRO uses SAPI 4 instead: SAPI 5 is not available for Windows 95, SAPI 5 still has some design issues that are being worked on, and SAPI 4 is usually installed by default in Windows ME/2000.

Prerequisite topics

None

Required components

None

What to do

The TApdSapiEngine component utilizes the services of SAPI 4 (Speech API). SAPI 4 must be installed prior to use. The SAPI 4 installer is not provided in the APRO installation due to possible licensing issues, however it can be downloaded from http://www.microsoft.com/speech/. For distribution to your customers, or on your development machine, download SPCHAPI.EXE, which contains the core SAPI 4 engines. The SAPI 4a SDK contains all of the required files, as well as several additional utilities and Visual C++, Visual Basic and ActiveX examples.

Once the SAPI 4 API (SPCHAPI.ZIP) has been downloaded, it can be installed through the following procedures. Be sure to check the documentation provided on Microsoft's site for updated installation instructions and system requirements. Microsoft provides these instructions for installing the SAPI 4 supporting components:

- Have your setup program install SpchAPI.exe into a temporary directory.
- Run SpchAPI.exe. It is a self-extracting executable that will install all the necessary files and registry entries.
- Delete the temporary copy of SpchAPI.exe after it has been installed.
- Your setup program does not need to uninstall the Microsoft Speech API components since it is a system component. Users wishing to uninstall the Microsoft Speech API can do so through the Control Panel, Add/Remove Programs.

SpchAPI.exe has several options to somewhat customize the install process.

The SpchAPI.exe installer is provided by Microsoft, and contains the speech engines. Microsoft also provides these engines in the SAPI 4 SDK, which can also be downloaded from http://www.microsoft.com/speech/. The SAPI 4 SDK is not required, although the supplied help files do provide more details on the inner workings and options than the APRO documentation.

Related examples

None

Setting up Speech Synthesis

This topic shows how to set up speech synthesis.

The TApdSapiEngine component provides speech synthesis and recognition capabilities. The Speak method of this component provides an easy way to add speech synthesis to your programs.

Prerequisite topics

"Installing SAPI4" on page 180.

Required components

TApdSapiEngine

What to do

The TApdSapiEngine component is designed to provide support for speech synthesis and recognition. To make use of this component for speech synthesis, drop a TApdSapiEngine component on your form. The Speak method of the TApdSapiEngine component configures the SAPI engine and instructs it to synthesize the text provided in the string parameter to the method. The following code will cause the SAPI engine speak to the default audio output device:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ApdSapiEngine1.Speak ('All your base are belong to us!');
end;
```

Related examples

EXSAPI.DPR

Selecting Speech Synthesis Voices

This topic shows how to select voices for speech synthesis.

The TApdSapiEngine component provides speech synthesis and recognition capabilities. Most speech synthesis engines provide a variety of voices. These voices can be accessed via the SSVoices property of the TApdSapiEngine component.

Required components

TApdSapiEngine

Prerequisite topics

“Installing SAPI4” on page 180.

“Setting up Speech Synthesis” on page 182.

What to do

Drop a TApdSapiEngine component of the form. The SSVoices property of the TApdSapiEngine component lists all the available speech synthesis voices and engines. In addition, subproperties of the SSVoices property provide additional information about the voices.

The following example will list all the speech synthesis voices to a TMemo component:

```
procedure TForm1.Button1Click (Sender : TObject);
var
  i : Integer;

begin
  Memo1.Lines.Clear;
  for i := 0 to ApdSapiEngine1.SSVoices.Count - 1 do
    Memo1.Lines.Add (ApdSapiEngine1.SSVoices[i]);
end;
```

To select a voice, set the CurrentVoice property of the Voices class to the index of the voice you want to use. Remember that the voices are numbered from 0. The following code will use the third voice.

```
ApdSapiEngine1.SSVoices.CurrentVoice := 2;
```

It is often necessary to select a speech synthesis engine that has specific features. Most commonly, you will need to find engines that have been optimized for either PC or telephony usage.

The following code will list only those engines that are optimized for PC use:

```
procedure TForm1.Button1Click (Sender : TObject);
var
  i : Integer;
begin
  for i := 0 to ApdSapiEngine1.SSVoices.Count - 1 do
    if tfPCOptimized in
        ApdSapiEngine1.SSVoices.Features[i] then
      Memo1.Lines.Add (ApdSapiEngine1.SSVoices[i]);
end;
```

A simple modification to the code will list only those engines that optimized for telephony usage:

```
procedure TForm1.Button1Click (Sender : TObject);
var
  i : Integer;
begin
  for i := 0 to ApdSapiEngine1.SSVoices.Count - 1 do
    if tfPhoneOptimized in
        ApdSapiEngine1.SSVoices.Features[i] then
      Memo1.Lines.Add (ApdSapiEngine1.SSVoices[i]);
end;
```

Related examples

EXSAPI.DPR

Setting up Speech Recognition

This topic shows how to set up speech recognition.

The TApdSapiEngine component provides speech synthesis and recognition capabilities. The Listen method, in conjunction with a vocabulary, provides an easy way to add speech synthesis to your programs.

Prerequisite topics

“Installing SAPI4” on page 180.

“Setting up Speech Synthesis” on page 182.

Required components

TApdSapiEngine

What to do

Drop a TApdSapiEngine component on the form. Before speech recognition can occur, a vocabulary must be set up. This is done through the WordList property. The WordList property is a TStringList that contains words that you want to recognize. If the Dictation property is False, only words contained in WordList will be recognized; if Dictation is True WordList is ignored and SAPI will attempt to recognize all words. Dictation mode is usually much slower, and is very much less accurate.

Determine which words you want to recognize first. If you are requesting numeric input, add “one”, “two”, “hundred”, etc. Once the vocabulary is set up, a call to Listen will start the speech recognition. When SAPI recognizes a word, the OnPhraseFinish event will be generated and the recognized word is passed in the Phrase parameter on the event handler.

The following example listens for “red”, “green”, or “blue.” If one of those words are spoken, it will be added to a memo.

```
procedure TForm1.Button1Click (Sender : TObject);
begin
  { Set up the vocabulary }
  ApdSapiEngine1.WordList.Clear;
  ApdSapiEngine1.WordList.Add ('red');
  ApdSapiEngine1.WordList.Add ('blue');
  ApdSapiEngine1.WordList.Add ('green');
  { Listen }
  ApdSapiEngine1.Listen;
end;

procedure TForm1.ApdSapiEngine1PhraseFinish (
  Sender : TObject; const Phrase : WideString);
begin
  if Phrase <> '' then
    Memo1.Lines.Add (Phrase);
end;
```

To stop speech recognition, call the StopListening method of the TApdSapiEngine component.

Related examples

EXSAPI.DPR

Selecting a Speech Recognition Engine

This topic shows how to select an engine for speech recognition.

The TApdSapiEngine component provides speech synthesis and recognition capabilities. Frequently multiple engines with differing features are provided for speech recognition.

Required components

TApdSapiEngine

Prerequisite topics

“Installing SAPI4” on page 180.

“Setting up Speech Recognition” on page 185.

What to do

Drop a TApdSapiEngine component of the form. The SREngines property of the TApdSapiEngine component lists all the available speech recognition engines. In addition, subproperties of the SREngines property provide additional information about the engines.

The following example will list all the speech recognition engines in a TMemo component:

```
procedure TForm1.Button1Click (Sender : TObject);
var
  i : Integer;
begin
  Memo1.Lines.Clear;
  for i := 0 to ApdSapiEngine1.SREngines.Count - 1 do
    Memo1.Lines.Add (ApdSapiEngine1.SREngines[i]);
end;
```

To select an engine, set the CurrentEngine property of the SREngines class to the index of the engine you want to use. Remember that the engines are numbered from 0. The following code will use the third engine:

```
ApdSapiEngine1.SSVoices.CurrentEngine := 2;
```

It is often necessary to select a speech recognition engine that has specific features. Most commonly, you will need to find engines that have been optimized for either PC or telephony usage.

The following code will list only those engines that are optimized for PC use:

```
procedure TForm1.Button1Click (Sender : TObject);
var
  i : Integer;
begin
  for i := 0 to ApdSapiEngine1.SREngines.Count - 1 do
    if sfPCOptimized in
        ApdSapiEngine1.SREngines.Features[i] then
      Memo1.Lines.Add (ApdSapiEngine1.SREngines[i]);
end;
```

A simple modification to the code will list those engines that optimized for telephony usage:

```
procedure TForm1.Button1Click (Sender : TObject);
var
  i : Integer;
begin
  for i := 0 to ApdSapiEngine1.SREngines.Count - 1 do
    if sfPhoneOptimized in
        ApdSapiEngine1.SREngines.Features[i] then
      Memo1.Lines.Add (ApdSapiEngine1.SREngines[i]);
end;
```

Related examples

EXSAPI.DPR

Using the Speech Recognition VU Meter

This topic shows how to use the speech recognition VU meter to provide feedback to the user.

The speech recognition functions TApdSapiEngine provide an OnVUMeter event that can be used to provide feedback to the user when speech recognition is active.

Required components

TApdSapiEngine

Prerequisite topics

“Installing SAPI4” on page 180.

“Setting up Speech Recognition” on page 185.

What to do

Drop a TApdSapiEngine component on the form. The OnVUMeter event will fire periodically when the SAPI engine is listening.

The following example will expand upon the example developed in “Setting up Speech Recognition”. Add a TProgressBar component to the form. Set the Orientation of the progress bar to pbVertical and adjust its size accordingly. Set the progress bar’s Max property to 65535.

```
procedure TForm1.Button1Click (Sender : TObject);
begin
  { Set up the vocabulary }
  ApdSapiEngine1.WordList.Clear;
  ApdSapiEngine1.WordList.Add ('red');
  ApdSapiEngine1.WordList.Add ('blue');
  ApdSapiEngine1.WordList.Add ('green');
  { Listen }
  ApdSapiEngine1.Listen;
end;
```

```
procedure TForm1.ApdSapiEngine1PhraseFinish (Sender : TObject;
  const Phrase : WideString);
begin
  if Phrase <> '' then
    Memo1.Lines.Add (Phrase);
end;

procedure TForm1.ApdSapiEngine1VUMeter (Sender : TObject;
  Level : Integer);
begin
  ProgressBar1.Position := Level;
end;
```

Related examples

EXSAPI.DPR

Using Speech Synthesis and Recognition Over a Phone

This topic shows how to provide speech capabilities over a voice telephony connection.

Speech capabilities over a voice connection are provided by a specialized TAPI device, the TApdSapiPhone. This component provides all the functionality of TAPI, but is also capable of routing speech synthesis or recognition through the phone line.

Required components

TApdComPort

TApdSapiEngine

TApdSapiPhone

Prerequisite topics

“Installing SAPI4” on page 180.

“Configuring a TAPI Device” on page 91.

“Setting up Speech Synthesis” on page 182.

“Selecting Speech Synthesis Voices” on page 183.

“Setting up Speech Recognition” on page 185.

“Selecting a Speech Recognition Engine” on page 187.

What to do

The TApdSapiPhone device handles the phone connection exactly like a TApdTapiDevice. When a connection is made to this device, it will automatically configure a linked TApdSapiEngine component to route it’s speech synthesis and recognition over the phone line.

The following example will place the TApdSapiPhone in auto-answer mode. When a call is made to it, it will say a few words over the phone and hang up.

It is important to make sure that the speech synthesis and recognition engines that you are using support telephony. The SetTelephoneSS and SetTelephoneSR methods below will select the first telephone capable engines. Note that the TApdSapiPhone component requires a voice-capable modem and either Unimodem/V or Unimodem/5.

```
procedure TForm1.Button1Click(Sender: TObject);
  procedure SetTelephoneSS;
  { Set the speech synthesis engine to the first telephone
    optimized voice.  This assumes that at least one telephone
    optimized voice is installed. }
  var
    i : Integer;
  begin
    for i := 0 to ApdSapiEngine1.SSVoices.Count - 1 do
      if tfPhoneOptimized in
        ApdSapiEngine1.SSVoices.Features[i] then begin
          ApdSapiEngine1.SSVoices.CurrentVoice := i;
          Exit;
        end;
  end;

  procedure SetTelephoneSR;
  { Set the speech recognition engine to the first telephone
    optimzied engine.  This assumes that at least one
    telephone optimized engine is installed. }
  var
    i : Integer;
  begin
    for i := 0 to ApdSapiEngine1.SREngines.Count - 1 do
      if sfPhoneOptimized in
        ApdSapiEngine1.SREngines.Features[i] then begin
          ApdSapiEngine1.SREngines.CurrentEngine := i;
          Exit;
        end;
  end;

begin
  { Make sure that telephone optimized voices are in use }
  SetTelephoneSS;
  SetTelephoneSR;
  { Connect the phone to a SAPI engine }
  ApdSapiPhone1.SapiEngine := ApdSapiEngine1;
  { Configure the phone and answer }
  ApdSapiPhone1.AnswerOnRing := 2;
  ApdSapiPhone1.AutoAnswer;
end;
```

```
procedure TForm1.ApdSapiPhone1TapiConnect(Sender: TObject);
begin
  ApdSapiEngine1.Speak ('All your base are belong to us!');
  ApdSapiEngine1.WaitUntilDoneSpeaking;
  ApdSapiPhone1.CancelCall;
end;
```

Adding speech recognition to this is easy. Make a call to ApdSapiPhone1.Listen. The OnPhraseFinish event will fire when words are recognized. StopListening is used to end the speech recognition.

A word of caution, most telephony connections are half duplex. You cannot have the SAPI engine speaking and listening at the same time.

Related examples

EXSAPIPH.DPR

5

Asking the User for Information Over a Voice Connection

This topic shows how to ask the user questions using the TApdSapiEngine and TApdSapiPhone components.

The TApdSapiPhone component provides several methods for asking the user for information. These methods can be called any time that a call is active.

Required components

TApdComPort

TApdSapiEngine

TApdSapiPhone

Prerequisite topics

“Installing SAPI4” on page 180.

“Configuring a TAPI Device” on page 91.

“Setting up Speech Synthesis” on page 182.

“Selecting Speech Synthesis Voices” on page 183.

“Setting up Speech Recognition” on page 185.

“Selecting a Speech Recognition Engine” on page 187.

“Using Speech Synthesis and Recognition Over a Phone” on page 191.

What to do

After a connection has been made, call one of the AskFor methods of the TApdSapiPhone component. Several of these are provided for the most common questions that a user may need to be asked. These are, AskForDate, AskForExtension, AskForList, AskForPhoneNumber, AskForSpelling, AskForTime, and AskForYesNo.

The AskFor methods will return when some reply is received from the user. The following example shows how to use AskForYesNo. It assumes that a phone call is already in progress:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Reply : Boolean;
begin
  case ApdSapiPhone1.AskForYesNo(Reply, 'Say yes or no') of
    prOk : { The user response is in Reply. }
    prAbort : { A fatal error when getting the response }
    prNoResponse : { The user did not respond }
    prOperator : { The user asked for an operator }
    prHangUp : { The user asked to hang up }
    prBack : { The user wants to go back a step }
    prCheck : { The user gave an ambiguous reply }
    prError : { There was a recoverable error }
    prUnknown : { There was an inexplicable reply }
  end;
end;
```

The AskFor methods return a variety of codes. These codes need to be handled by your application. Some of them (prOperator, prBack, and prHangUp) can be turned off in the TApdSapiPhone Options property.

Related examples

EXSAPIPH.DPR

RS-485 Support

This topic discusses how to use the TApdComport with an RS-485 serial port.

Required components

TApdComPort

Prerequisite topics

“Setting Up a Comport” on page 74.

Related components

None

What to do

RS-485 is a standard for multidrop communications. It’s typically used when a computer needs to communicate with several devices. RS-485 can be used over greater distances than RS-232 because of its lower impedance requirements and the way voltage is represented on the lines. RS-485 also allows several devices to be connected in a daisy chain fashion. The devices on the line can either be receiving or sending data, but not both. In order to send data—the device has to take control of the line, send the data, and quickly release the line so it can listen for replies from the remote devices.

RS-485 boards for PCs handle this transition between sending and receiving in a couple different ways (and most boards allow you to configure the board for the option that best suits your application).

One method is placing your data in a packet with special characters. This method has always worked fairly well with APRO, but has the limitation that the special characters can't exist in your data. Another quite popular method is known as "RTS Control"—where the RTS line is raised, the data is sent, and RTS is immediately lowered. This method has the advantage of not requiring special characters, but is a bit more difficult to handle from a timing point of view. If RTS is lowered too soon (while characters are still in the UART), data can be lost. If RTS is lowered too late, a response from a remote device could be lost.

Due to these tight timing requirements, the TApdComport has a RS485Mode property that can be set to handle RTS control (as well as can be done under Windows without a special device driver)—and will work in the vast majority of situations.

Related examples

None

Setting Up a Terminal

This topic shows how to configure a TAdTerminal component to display incoming and outgoing information.

The ability to display all characters being sent and received is an integral part of most communications applications. The TAdTerminal component displays the incoming and outgoing characters as they are received at the comport.

Required components

TApdComPort

TAdTerminal

Prerequisite topics

"Setting Up a Comport" on page 74.

Related components

TAdTTYEmulator

TAdVT100Emulator

What to do

Drop a TAdTerminal component on the form. If a TApdComPort component is already on the form, it is automatically associated with the TAdTerminal.ComPort property and the two components are linked together. The TAdTTYEmulator and the TAdVT100Emulator components can be used to provide terminal emulation for the TAdTerminal. If an emulator component is not specified, the TAdTerminal component will create its own internal TAdTTYEmulator instance to provide generic emulation services.

If the TApdComPort.AutoOpen property is True, the TAdTerminal component opens the port when it is created, enabling you to type commands immediately. If AutoOpen is False, the terminal window is inactive until the port is opened. Change the TAdTerminal.Active property to False to disable displaying characters or True to display all available characters.

To insert characters into the terminal window without sending them to the remote machine, use the TAdTerminal.WriteString method. The string being inserted should be bracketed by a CR/LF pair (#13#10) to ensure that the new text is on a separate line in the terminal. The following example opens a TApdComPort component, activates a TAdTerminal component, and displays “Hello world” in the terminal window:

```
procedure TForm1.ActivateBtnClick(Sender : TObject);
begin
  ApdComPort1.Open := True;
  AdTerminal1.ComPort := ApdComPort1;
  AdTerminal1.Active := True;
  AdTerminal1.WriteString(#13#10+'Hello world'#13#10);
end;
```

Related examples

TERMDEMO.DPR

EXMDI.DPR

Setting Up a Terminal Emulator

This topic shows how to set up a terminal to use a different emulator.

There are two terminal emulators available with Async Professional: the teletype (TTY) emulator and the VT100 emulator. The first one just echoes all data to the terminal window and does not attempt to interpret anything in the data stream. Indeed, this is the default emulation for the terminal, the one that is force is you don't use a specific emulator component. The second one is a full VT100 emulation that identifies terminal control sequences in the incoming data stream and interprets them as commands to alter the terminal's display.

Required components

TApdComPort

TAdTerminal

TAdVT100Emulator

Prerequisite topics

"Setting Up a Comport" on page 74.

"Setting Up a Terminal" on page 198.

Related components

TAdTTYEmulator

What to do

Drop a TAdVT100Emulator onto the form and then a TAdTerminal. Dropping the components in this order means that the terminal component finds the emulator and links up with it automatically. Alternatively you can set the Emulator property of the TAdTerminal to point to the TAdVT100Emulator. At run time, the terminal and emulator combination will pretend to be a VT100 terminal to any host computer that uses it.

If you wish to provide another emulator for use with the terminal, there are several steps you must go through. You must provide a set of mappings for the keyboard to tell the emulator how to map PC keystrokes onto terminal keystrokes; a character set mapping to inform the emulator how to draw characters from different character sets; and a new parser class that interprets the incoming data stream, identifying terminal control sequences. Our recommendation is to study the source code for the TAdVT100Emulator class.

Related examples

EXNEWTRM.DPR

TERMDEMO.DPR

Chapter 6: Demonstration Programs

The supplied demonstration programs combine many Async Professional features into each program. While you might not want to use the larger ones as your first introduction to Async Professional, they show what is possible and in several cases demonstrate advanced features that are not used in the example programs.

The terminal demo, TermDemo, is a simple terminal-oriented communications program.

The modem database demo, ModDemo, demonstrates a user interface for adding, modifying, and deleting records in the Async Professional modem database.

The four fax demonstration programs, SendFax, RcvFax, Cvt2Fax, and ViewFax, are fairly simple programs that can send and receive faxes, convert text or image files into fax files for transmission, and view fax files.

RasDemo is a simple Remote Access Service dialer program that can dial and manipulate RAS phonebooks. It is based on the TApdRasDialer component.

FtpDemo is a simple FTP client program that can connect to an FTP server, login, transfer files, display directory contents, etc. It is based on the TApdFtpClient component.

The paging demo, ExPaging, demonstrates a user interface for maintaining a list of pager IDs and access addresses and using them to send alphanumeric pages.

TCom3 is a comprehensive communications demonstration program that provides one of the endless possible ways to use Async Professional’s components in a real-life business application.

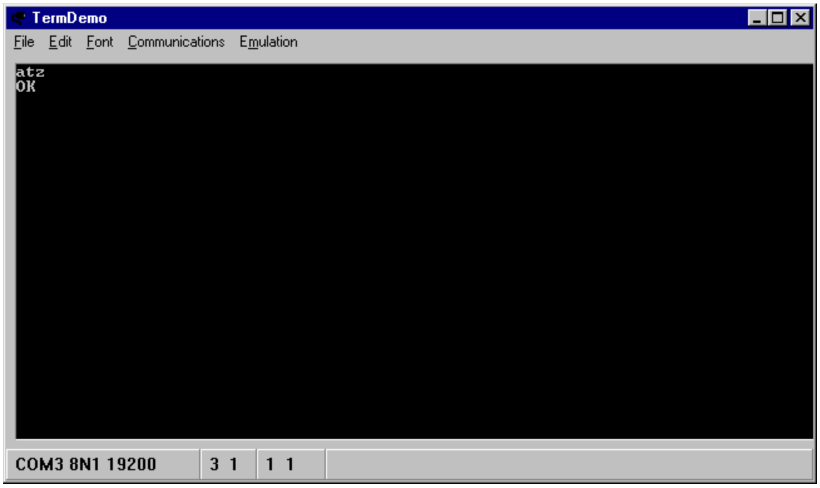
Terminal Demo

TermDemo is a simple terminal communications program. It includes only terminal window features without protocol or modem support, which makes it a clearer example of terminal window programming techniques than TCom3.

The main window

TermDemo's main window, shown in Figure 6.1, consists of the terminal window and a menu bar. The menu bar provides access to all of TermDemo's functions. From the menu you can clear the window, change the communication parameters, or use a different emulation.

6



```
atz
OK
```

```
COM3 8N1 19200 | 3 1 | 1 1
```

Figure 6.1: The main window.

The terminal portion of the main window is TermDemo's primary work area. The terminal displays characters received at the serial port and transmits characters that you type. Additionally, the terminal provides VT100 emulation, which translates escape sequences sent by the remote system into cursor positioning and color changes.

The terminal has two modes: normal mode and scrollback mode. Normal mode is the default. Pressing the Ins key toggles scrollback mode. In scrollback mode, you can look back through previously received lines that have scrolled off your screen. To navigate in scrollback mode, simply use the arrow keys to move up, down, left, and right through the buffer. Data flow is blocked while in scrollback mode. Pressing Ins again returns to normal mode.

The menu bar

The File menu

Playback file

Use this menu option to play back a file through the emulator. TermDemo reads data from the file and processes it just as if it was received from the serial port. Each byte is routed through the attached emulator and, if it is a displayable character, displayed at the current cursor location in the terminal window. One of the common uses of this feature is to play back capture files created by TCom3 or other communications programs.

Clear screen

This option clears the terminal window and the screen buffer. The cursor is placed at row 1, column 1.

Exit

Exits TermDemo.

The Edit menu

Copy

This is the only menu choice for this submenu. Cut and Paste options are not available (as they are with most windows) because the terminal window does not permit data to be removed, and the only data that can be added is data received from the serial port or added using the WriteXxx routines.

Selecting the Copy menu item copies the currently selected text to the Windows clipboard. Text can be selected by moving the mouse cursor to the start of the area to be marked, pressing and holding the mouse button while moving the mouse cursor to the end of the area, and then releasing the button.

The Communications menu

Set parameters

Selecting this item pops up the Communication Parameters dialog box as shown in Figure 6.2.

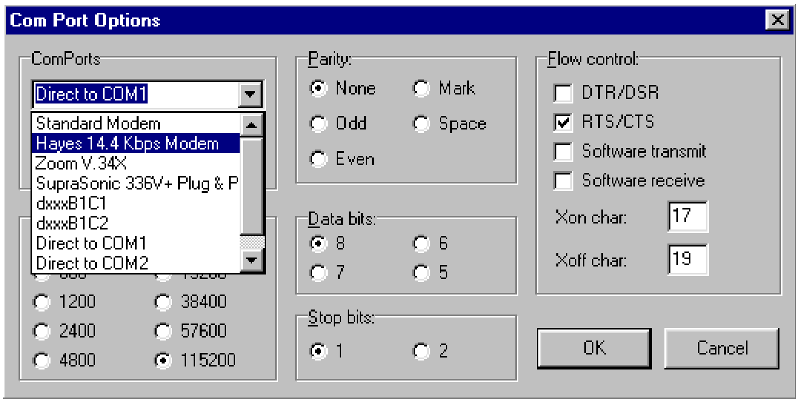


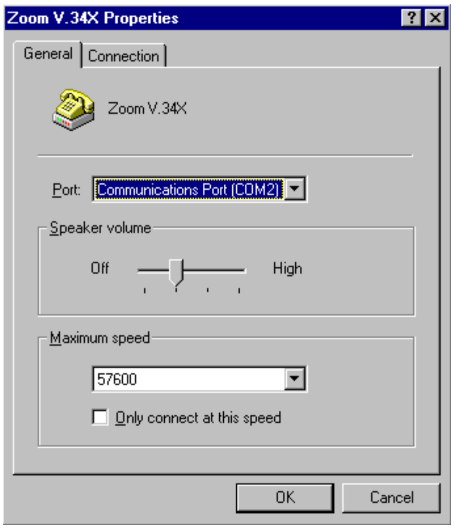
Figure 6.2: Communications Parameters dialog box.

You can use this dialog box to change the current communications port, baud rate, parity, data bits, and stop bits settings. The current settings are indicated by the selected radio buttons and the combo box. Change the settings by clicking on the radio button of the desired new setting, or selecting the serial port in the combo box.

When you are satisfied with the selected parameters, click OK or press Enter. TermDemo closes the current port and reopens the port or device with the new settings. To discard your changes click Cancel or press Esc.

Configure TAPI

Selecting this menu item allows you to view or change the properties of the selected TAPI device, as shown in Figure 6.3.



Zoom V.34X

Port: Communications Port (COM2)

Speaker volume: Off — High

Maximum speed: 57600

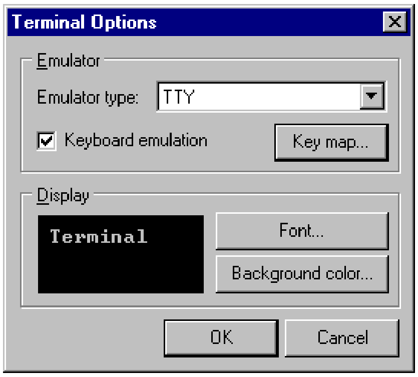
☐ Only connect at this speed

Figure 6.3: TAPI device properties dialog box.

The Emulation menu

Set parameters

Selecting this menu item pops up the Emulator Options dialog box, shown in Figure 6.4, to enable you to switch between TTY and VT100 emulation.



Emulator

Emulator type: TTY

☑ Keyboard emulation

Display

Terminal

Figure 6.4: Emulator Options dialog box.

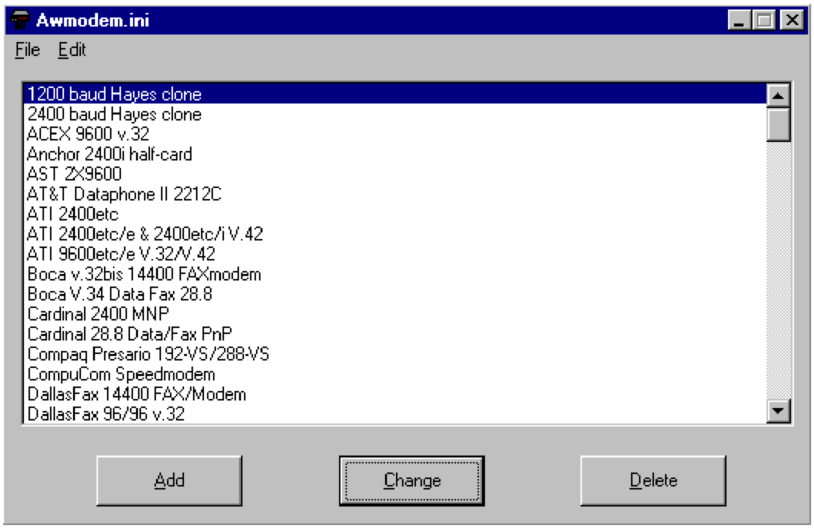
Modem Database Demo

A modem database is a collection of information about one or more modems, including all the strings necessary to initialize, configure, dial, and answer a particular modem. A database of popular modems is shipped with Async Professional in the file AWMODEM.INI.

ModDemo is a program that creates or modifies modem databases like AWMODEM.INI. You can use it to add, delete, or change modems in the supplied modem database, or create your own modem databases.

The main window

ModDemo's main window, shown in Figure 6.5, consists of the following list box, which contains a list of all the modems in the current database, a menu bar, and push buttons for adding, changing, and deleting modems in the database.

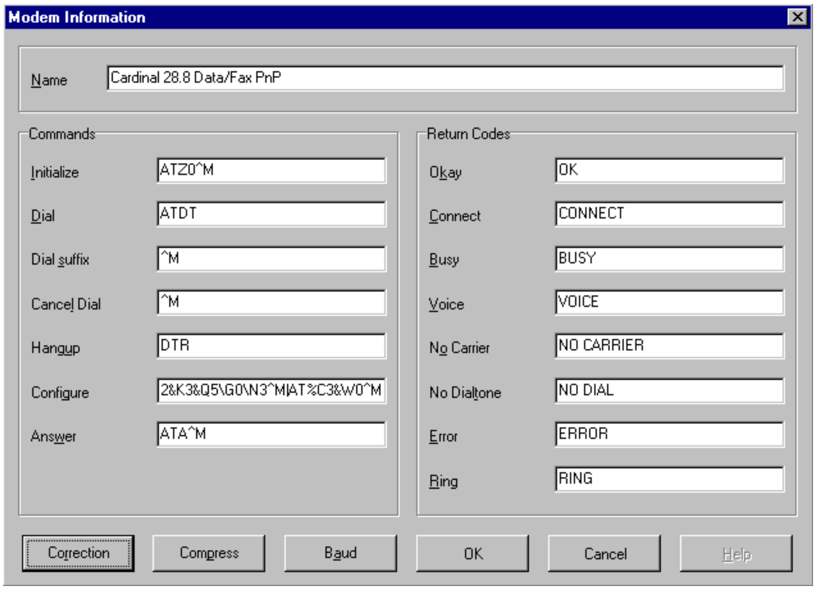


1200 baud Hayes clone
2400 baud Hayes clone
ACEX 9600 v.32
Anchor 2400i half-card
AST 2X9600
AT&T Dataphone II 2212C
ATI 2400etc
ATI 2400etc/e & 2400etc/i V.42
ATI 9600etc/e V.32/V.42
Boca v.32bis 14400 FAXmodem
Boca V.34 Data Fax 28.8
Cardinal 2400 MNP
Cardinal 28.8 Data/Fax PnP
Compaq Presario 192-VS/288-VS
CompuCom Speedmodem
DallasFax 14400 FAX/Modem
DallasFax 96/96 v.32

Figure 6.5: ModDemo's main window.

The add button

Click on the add button to add a new modem definition to the database. The screen shown in Figure 6.6 appears.



Modem Information

Name: Cardinal 28.8 Data/Fax PnP

Commands		Return Codes	
Initialize	ATZ0^M	Okay	OK
Dial	ATDT	Connect	CONNECT
Dial suffix	^M	Busy	BUSY
Cancel Dial	^M	Voice	VOICE
Hangup	DTR	No Carrier	NO CARRIER
Configure	2&K3&Q5\G0\N3^MAT%C3&W0^M	No Dialtone	NO DIAL
Answer	ATA^M	Error	ERROR
		Ring	RING

Figure 6.6: Modem Information dialog box.

The dialog box is initially empty when you are adding a new modem. Use this dialog box to enter general information about the modem you wish to add.

In the Name field, at the top of the dialog box, you can enter an arbitrary name for the modem. The modem database uses this name to modify and retrieve information about the modem. After adding the new modem, the name that you enter here will be displayed in the list box on the main window.

The Commands group of fields, to the left and down from the Name field, is where you enter information about the commands that are sent to the modem to perform specific operations. For instance, in the Initialize field, you would enter the command that is sent to the modem to initialize it.

When entering commands, there are a few special characters and tokens that you can enter to alter the behavior of Async Professional when it sends these commands to the modem. The '~' character indicates that Async Professional should stop for a moment (by default, half a second) before sending the next character in the command. You can also enter control characters into a command by prefacing the control character letter with the '^' character. For instance, to insert a carriage return in a command string, you would enter "^M".

The Configure command can also contain one special character that the other commands cannot. If you insert a pipe ('|') character into the Configure command, Async Professional waits until it receives an OK result from the modem before processing the next command.

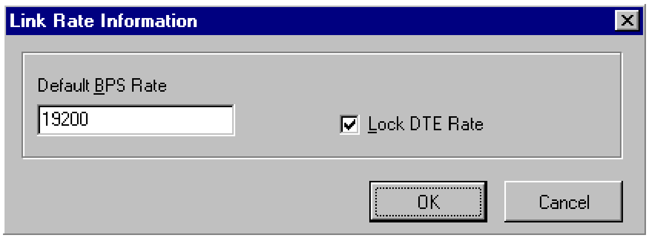
The Return Codes group of fields, to the immediate right of the Commands group, is where you enter all the strings that your modem can return when commands are sent to it. You can get these strings from your modem manual.

At the bottom of the dialog box are six buttons. The three buttons on the left are used to enter more information about the modem.

Clicking on the Correction or Compress button will pop up another dialog box into which you can enter error correction or data compression indicator strings. For instance, many modems return the string "LAPM" during connection handshaking when they detect a connection that can handle data compression. If that is the case with the modem you are adding to the database, click on Compress and enter the string "LAPM" in the first field in the dialog that pops up.

The tag entry dialog box has room to hold five different data compression or error correction tags. This is because many modems return different strings depending on what type of data compression or error correction is available for a given connection. For instance, the Practical Peripherals 14400 baud fax modem returns, at various times, the strings "MNP5", "LAPM", or "V.42BIS", depending on the type of error correction available. If you are adding such a modem to the database, you would enter each string on its own line in the tag entry dialog box.

Also at the bottom of the Modem Information dialog box is a button labeled Baud. Clicking on this button pops up the dialog box shown in Figure 6.7.



Default BPS Rate: 19200

[✓] Lock DTE Rate

Figure 6.7: Link Rate Information dialog box.

The first field on this dialog box is labeled Default BPS Rate. In this field, enter the default baud rate for the modem. Terminal programs can use this information when determining a set of default communications parameters for this modem. The next field is a check box labeled Lock DTE Rate. Many modems perform best when they communicate at a fixed rate (usually 19200 baud or greater) between the PC and the modem, regardless of the data rate negotiated with the remote modem. If yours is such a modem, check this box; otherwise leave it unchecked.

When you are finished entering data about the new modem, click OK at the bottom of the modem entry dialog box to save the modem to the database. If you do not want to save the modem definition, click Cancel.

The Change button

The Change button at the bottom of the main window allows you to change a modem's information. To change a modem, highlight the modem whose record you want to modify in the list box, click Change. Alternately, you can simply double-click on the item's name in the list box.

Clicking the Change button, or double clicking on a modem's name, pops up the same modem data entry dialog box that is used to add a new modem to the database. Make the modifications that you want, then click OK to save the modified modem definition, or click Cancel to abandon the changes.

The Delete button

You can delete a modem definition from the database by clicking on the Delete button. Highlight the name of the modem you want to delete in the list box, then click Delete. A message box will appear, asking if you are certain that you want to delete the modem. Click Yes and the modem will be deleted, or click No to cancel the deletion.

The File menu

The File menu, attached to the main window, provides various services for opening databases, creating new databases, and saving changes to a database.

New

If you want to create a new modem database, select the File | New option. This will clear any previously loaded modem database from memory. If you have made changes to the current database and have not yet saved them, you will be prompted to do so.

Open

You can load an existing modem database, such as Async Professional's AWMODEM.INI, into the program by selecting File | Open. A dialog box will pop up asking you which file you want to load. Select the filename from that dialog box and click OK. The names of all the modems contained in that database will be loaded into the list box on the main window.

Save

Once you have made additions, changes, or deletions to a modem database, you must save them to a file. Selecting File | Save from the menu will write all of your changes to disk. If the current database was not loaded from an existing file, you will be prompted to enter a file name before the database is written to disk.

Save As

Selecting File | Save as from the menu allows you to save the current modem database under a new name. After selecting the menu item, a dialog box appears asking you for the new filename. Enter the name you want and click OK.

Exit

Selecting File | Exit closes the ModDemo program. If you have made changes to the loaded database and have not yet saved them, you will be prompted to do so before the program closes.

The Edit menu

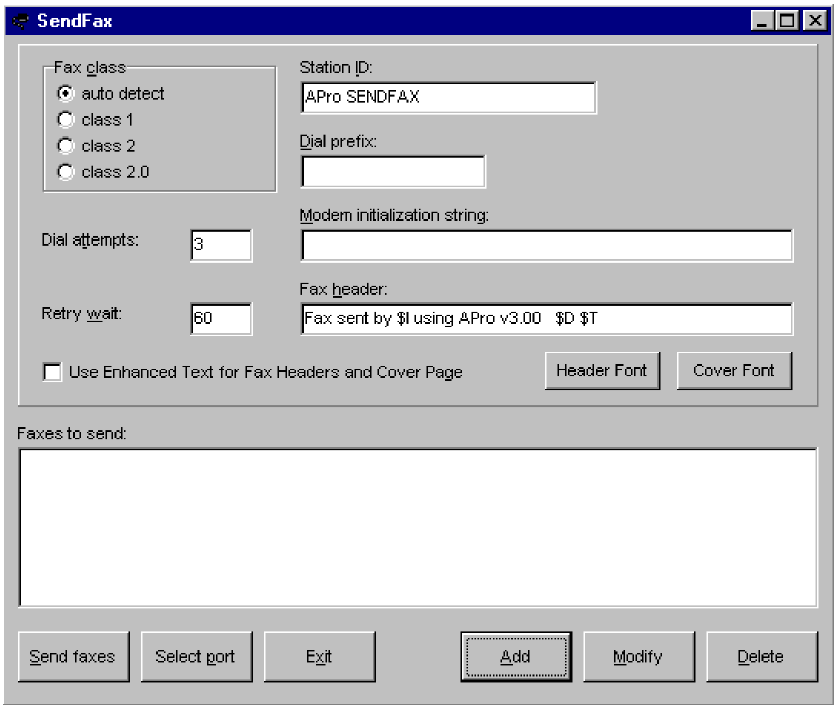
The Edit menu duplicates the functions of the push buttons at the bottom of the main window. Select Edit | Add to add a record, Edit | Change to modify the highlighted record, or Edit | Delete to delete the highlighted record.

Send Fax Demo

SendFax is a simple fax program that can send multiple faxes, with optional cover pages to multiple fax numbers. It supports only sending faxes, not receiving them, which makes it a clear example of the issues involved in transmitting faxes.

The main window

SendFax does not use menus or toolbars. All of the options are set from the main form, shown in Figure 6.8, and faxing operations are controlled by the row of buttons at the bottom of the form.



SendFax

Fax class: [✓ auto detect] class 1 class 2 class 2.0

Station ID: APro SENDFAX

Dial prefix:

Modem initialization string:

Dial attempts: 3

Fax header: Fax sent by $I using APro v3.00 $D $T

Retry wait: 60

☐ Use Enhanced Text for Fax Headers and Cover Page

Faxes to send:

Figure 6.8: SendFax main window.

The top half of the main form contains the configurable fax options. Use the Fax class radio button group to select the fax class supported by your modem. If you aren't sure of the class of your modem, select "auto detect" and the TApdSendFax component automatically detects the class.

The Dial attempts and Retry wait edit controls control how many times each call is tried if busy signals are encountered, and how long to wait, in seconds, between attempts.

The Station ID edit control contains the station ID, the identification string that is sent to the remote fax device.

The Dial prefix edit control, which defaults to an empty string, is for a standard dialing prefix, if one is required to dial an outside line (e.g., dial 9 for an outside line).

The Modem initialization string, which defaults to an empty string, can be used to send modem initialization commands. For example, you could set the modem initialization string to M1 (the modem command to turn on the speaker).

Fax header allows you to specify the header at the top of each transmitted page. The default fax header contains several replacement tags. The $I tag is replaced by the station ID, $D is replaced by today's date, and $T is replaced by the current time. So, the header that appears at the top of the received fax would look something like this:

Fax sent by APro SENDFAX using APro 3.0 04/15/97 12:15pm

See HeaderLine in the Reference Guide for more information.

The Use Enhanced Fonts check box control allows you to use TrueType fonts for the fax Header and Cover Page. If it is checked, the fonts specified in from the Header Font and Cover Font buttons will be used to render the Header and Cover Page of the fax.

The Head Font and Cover Font buttons will display the standard Font Selection Dialog, which allows you to select the font to use if the Use Enhanced Fonts check box control is checked.

Faxes to send contains a list of the fax files queued for sending. As each fax file is successfully transmitted, it is removed from the list box. If SendFax encounters a fatal error while sending a fax, the file name is not removed from the list box. If the error is a correctable one (e.g., the wrong phone number was specified), the fax entry can be corrected and Send faxes clicked again. SendFax will continue sending queued faxes, starting with the last failed fax.

The Add, Modify, and Delete buttons are used to modify the list of faxes displayed in the Faxes to send list box. Selecting Add or Modify displays a dialog box, shown in Figure 6.9, that allows you to specify or change the phone number, fax file name, and an optional cover sheet.

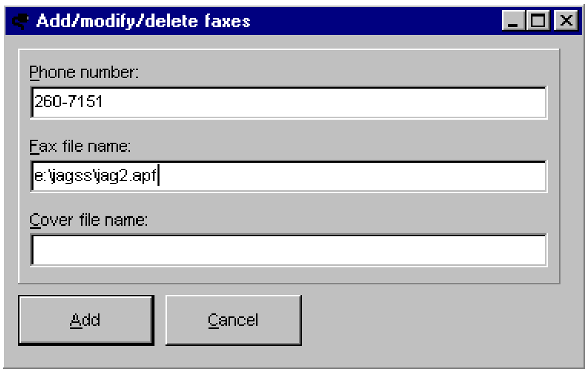


Figure 6.9: Add/modify/delete faxes dialog box.

After you finish adding or modifying, click on Add to return to the main window. To remove a fax from the list, highlight it and click Delete.

The Send faxes button starts the fax transmission process, starting with the first fax in the list and continuing through all faxes, stopping at the end of the list or when an error is encountered.

The Exit button ends the program. Any faxes still queued for transmission are not sent, nor are they saved for the next run of SendFax.

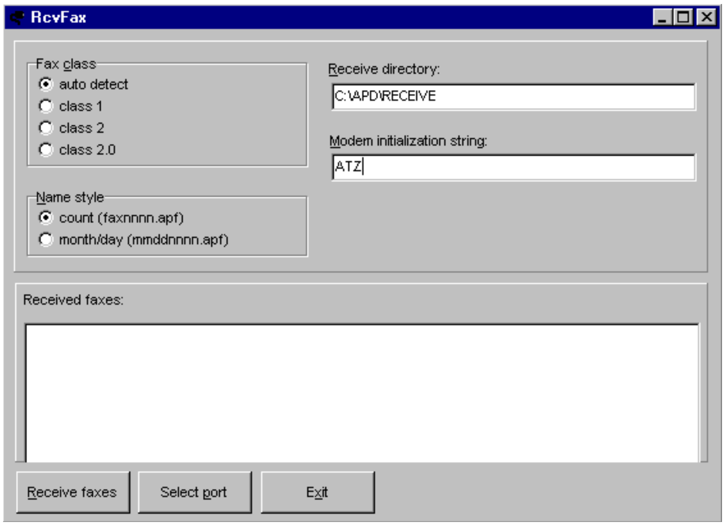
The SendFax demonstration program contains standard TApdFaxStatus and TApdFaxLog components for displaying faxing progress and creating a history file of all faxes transmitted. See “TApdFaxStatus Component” in the Reference Guide for a picture of the fax status display.

Receive Fax Demo

RcvFax is a simple fax program that waits for and answers incoming fax calls until it encounters a fatal error or is cancelled. It supports only receiving faxes, not transmitting them, which makes it a clear example of the issues involved in receiving faxes.

The main window

RcvFax does not use menus or toolbars. All of the options are set from the main form, shown in Figure 6.10, and faxing operations are controlled by the row of buttons at the bottom of the form.



Fax class: [✓ auto detect] class 1 class 2 class 2.0

Receive directory: C:\APD\RECEIVE

Modem initialization string: ATZ

Name style: [✓ count (faxnnnn.apf)] month/day (mmddnnnn.apf)

Figure 6.10: RcvFax main window.

The top half of the main form contains the configurable fax options. Use the Fax class radio button group to select the fax class supported by your modem. If you aren't sure of the class of your modem, select "auto detect" and the TApdReceiveFax component automatically detects the class.

The Name style radio buttons determine how incoming fax files are named. The default choice is "count", where incoming faxes are named:

```
faxnnnn.apf, where nnnn is a sequential number (starting at 0001)
that is the first free number for the current directory
```

The other available choice is month/day, where incoming faxes are named:

```
mmddnnn.apf, where mm is the month, dd is the day, and nnnn is a
sequential number (starting at 0001) for the number of files
received this day
```

The Receive directory edit control contains the directory name where incoming fax files are stored. If it is blank (the default), incoming fax files are stored in the current directory.

The Modem initialization string, which defaults to an empty string, can be used to send modem initialization commands. For example, you could set the modem initialization string to M1 (the modem command to turn on the speaker).

The Received faxes list box contains a list of all the fax files successfully received since RcvFax was started, along with the size of the fax file and the date and time it was received.

The Receive faxes button tells RcvFax to begin listening for faxes. RcvFax then displays its status form to show the progress of incoming faxes.

The Exit button ends the program.

The RcvFax demonstration program contains standard TApdFaxStatus and TApdFaxLog components for displaying faxing progress and creating a history file of all faxes received. See “TApdFaxStatus Component” in the Reference Guide for a picture of the fax status display.

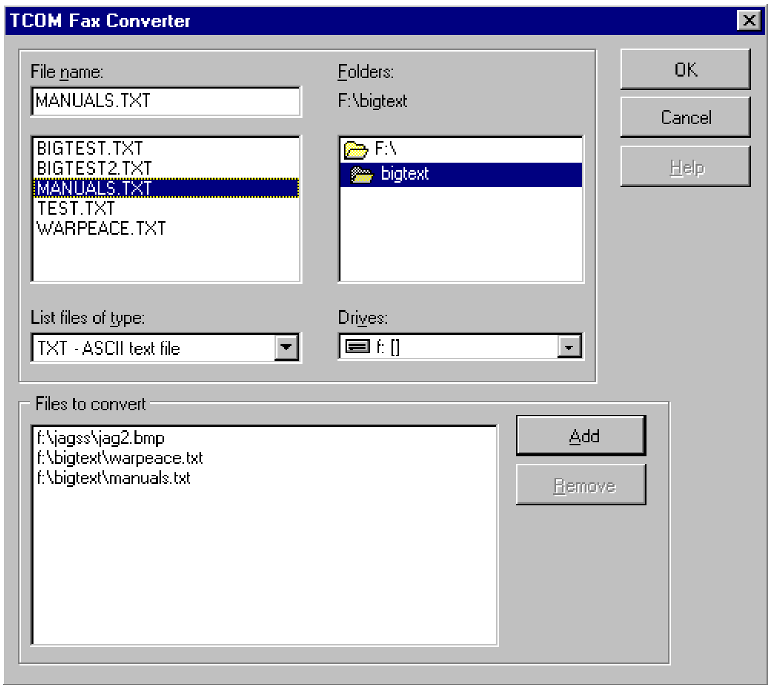
Fax Converter Demo

Cvt2Fax is a program that converts text, BMP, PCX, DCX, and TIFF files into APF (Async Professional Fax) files. An image or text document must be converted to an APF file before it can be transmitted.

The source files and forms are specially designed to allow you to simply add the forms and units to your own programs.

The main window

Cvt2Fax's main window is used for selecting files to convert.


TCOM Fax Converter

File name: MANUALS.TXT

BIGTEST.TXT
BIGTEST2.TXT
MANUALS.TXT
TEST.TXT
WARPEACE.TXT

Folders: F:\bigtext

F:\
bigtext

List files of type: TXT - ASCII text file

Drives: f: []

Files to convert

f:\jagss\jag2.bmp
f:\bigtext\warpeace.txt
f:\bigtext\manuals.txt

Figure 6.11: Cvt2Fax main window.

Files that are queued for conversion are shown in the lower list box. You can put a file directly in the conversion queue by typing its name in the edit control and pressing Enter. If the name you enter contains wildcard characters ('*' or '?'), the top two list boxes (the ones containing file and directory listings) are reloaded with information from the path you specify and the edit control is cleared. The file extension must be TXT, BMP, PCX, TIF, or DCX depending on the file format.

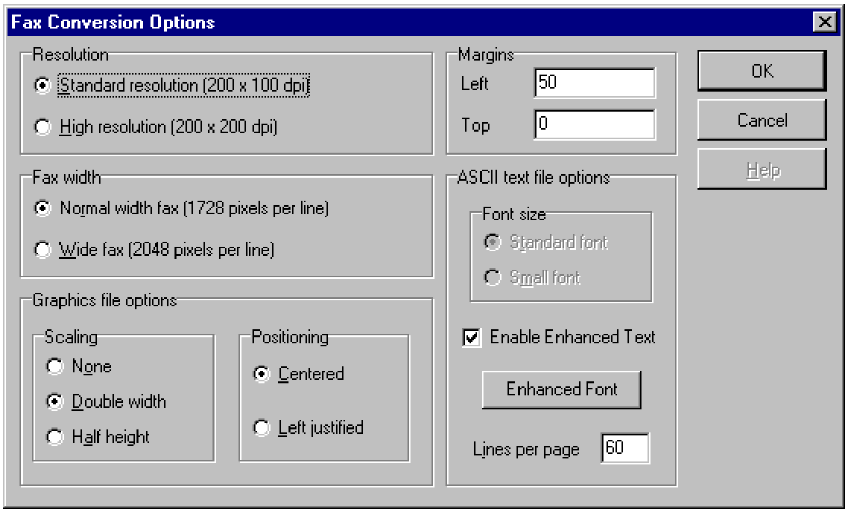
Alternately, the Folders list box displays all of the files in the current directory that match the mask you enter or that match the filter in the List files of type combo box. You can queue files from this multiple selection list box by selecting them and clicking the Add button, or by double-clicking on the file.

To remove files from the Files to convert list box, select the files and click the Remove button.

The Drives and Folders list boxes can be used to access all the drives and directories on your system.

Conversion options

Cvt2Fax allows you to specify several options for converting faxes. Click Options on the main window. The dialog box shown in Figure 6.12 appears.



Resolution
[✓ Standard resolution (200 x 100 dpi)]
High resolution (200 x 200 dpi)

Fax width
[✓ Normal width fax (1728 pixels per line)]
Wide fax (2048 pixels per line)

Graphics file options
Scaling: None, [✓ Double width], Half height
Positioning: [✓ Centered], Left justified

Margins
Left: 50
Top: 0

ASCII text file options
Font size: [✓ Standard font], Small font
[✓ Enable Enhanced Text]
Lines per page: 60

Figure 6.12: Fax Conversion Options dialog box.

The Resolution radio buttons allow you to choose the resolution of converted faxes. The default resolution is Standard (200x100 dpi).

The Fax width radio buttons allow you to choose the width of converted faxes. The default width—the only width most fax devices support—is 1728 pixels.

The Scaling radio buttons allow you to choose how graphics images are scaled when converted to standard resolution faxes. Because most image files contain images with a 1:1 aspect ratio, and because standard resolution faxes have a 2:1 aspect ratio, the Cvt2Fax program allows you to choose options that compensate for this. The default scaling setting is Double width. This doubles the width of converted images so that the resulting fax appears normal. The second option, Half height, halves the height of images to achieve the same effect. However, the image looks smaller than images converted with the double width option.

The Positioning radio buttons determine the position of image files on the fax page. Graphic images can either be centered in the page or at the left edge of the page.

The Font size radio buttons allow you to choose a font size for ASCII text files. Standard font (the default) allows text line lengths of up to 85 characters on a 1728 pixel wide fax. Small font allows about 144 characters per line.

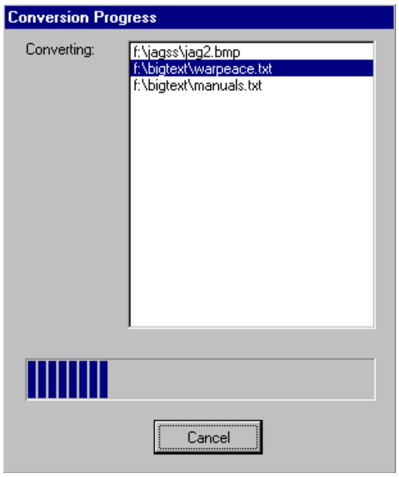
The Enable Enhanced Text check box gives you the option to use TrueType fonts in the conversion of ASCII text files. If this is checked, the Enhanced Font button is enabled, allowing the selection of the font to use.

An ASCII text file can be converted into one long page, or it can be broken up into multiple pages. The number of lines per page is set in the Lines per page edit control. The default setting is 60 lines per page (about 10 inches of text using the standard font). To leave the entire fax as one page, set lines per page to 0.

When you are finished setting options, click the OK button and the new options take effect. To discard any changes you made to the options, click the Cancel button.

Converting

After you select all the files you want to convert and set the desired options, click OK in the main window to begin the conversion. During the conversion, the status dialog box shown in Figure 6.13 is displayed.



f:\jagss\jag2.bmp
f:\bigtext\warpeace.txt
f:\bigtext\manuals.txt

Figure 6.13: Conversion Progress status dialog box.

The Converting list box displays the names of the files to be converted and highlights the name of the file that is currently being converted. The meter control shows the progress of the conversion of the current file. The Cancel button stops the conversion process.

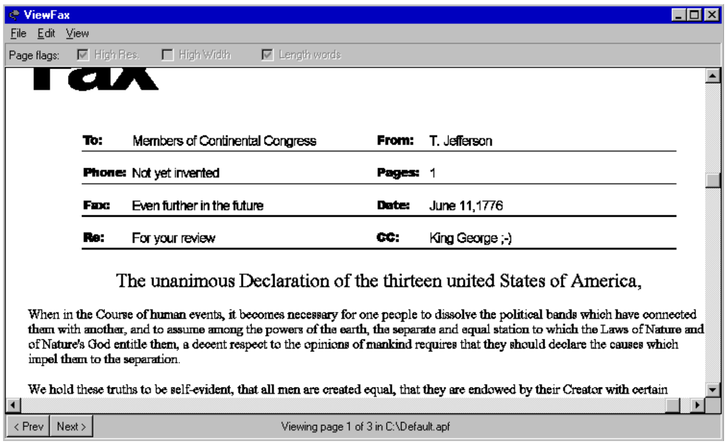
After all of the files are converted, you are returned to the fax converter main window where you can select more files to convert or exit.

Fax Viewer Demo

ViewFax implements a fax viewer component that allows you to view APF files. The source files and forms are specially designed to allow you to simply add the forms and units to your own programs. In fact, the fax viewer in the TCom demo program is based on the source code for ViewFax.

The main window

ViewFax's main window, shown in Figure 6.14, consists of a viewer area and a menu bar.


ViewFax
File Edit View
Page flags: High Res. High Width Length words
< Prev Next >
Viewing page 1 of 3 in C:\Default.apf

Fax

To:	Members of Continental Congress	From:	T. Jefferson
Phone:	Not yet invented	Pages:	1
Fax:	Even further in the future	Date:	June 11,1776
Re:	For your review	CC:	King George ;-)

The unanimous Declaration of the thirteen united States of America,

When in the Course of human events, it becomes necessary for one people to dissolve the political bands which have connected them with another, and to assume among the powers of the earth, the separate and equal station to which the Laws of Nature and of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.

We hold these truths to be self-evident, that all men are created equal, that they are endowed by their Creator with certain

Figure 6.14: ViewFax main window.

The viewer area, which constitutes the bulk of the screen, is where faxes are displayed. If the fax is too large to fit in the viewer area, the window can be resized to accommodate the size of the fax, or the scroll bars, which are displayed automatically, can be used to scroll through the fax. Navigational keystrokes are also accepted, see the section on the TApdFaxViewer component for the specific navigational keys.

The menu bar

The File menu

Open

This option allows you to load fax files (APF files) into the viewer. The Windows standard file open dialog box, as shown in Figure 6.15, is displayed.

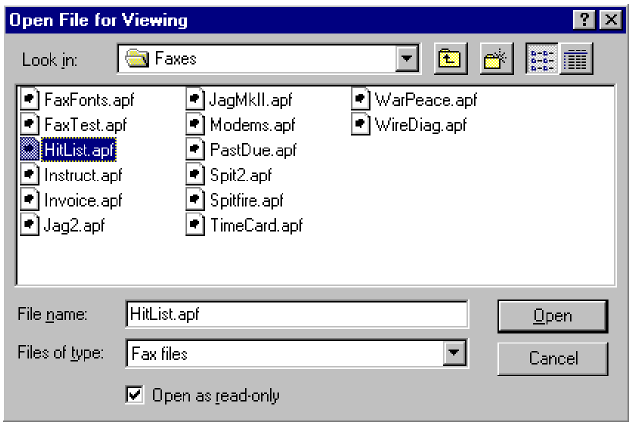
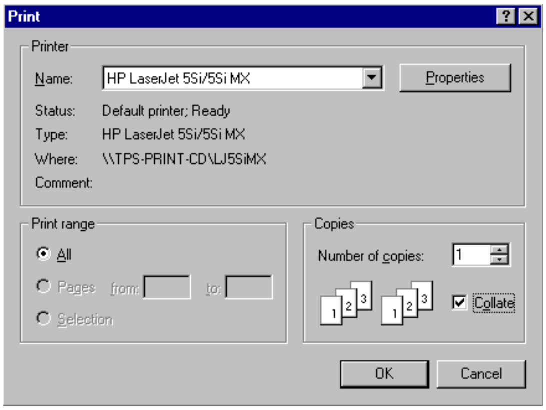


Figure 6.15: Open File for Viewing dialog box.

Select the file to view and it is loaded into the viewer. When a file is loaded, scaling or rotation settings are reset to their defaults. To scale or rotate a newly loaded fax, set the appropriate options in the View menu.

Print setup
ViewFax can print faxes in addition to viewing them. Choose Print Setup to display the standard Windows printer setup dialog box, shown in Figure 6.16, and choose the printer and print options.


Print

Printer
Name: HP LaserJet 5Si/5Si MX
Status: Default printer; Ready
Type: HP LaserJet 5Si/5Si MX
Where: \\TPS-PRINT-CD\LJ5SiMX
Comment:

Print range: [✓ All] Pages from: to: Selection

Copies
Number of copies: 1
Collate: ✓

Figure 6.16: Print dialog box.

Print
This option prints the viewed fax to the printer chosen using the Print Setup option. If no printer was chosen, the fax is printed to the default printer.

Faxes are printed using the psFitToPage option of the TApdFaxPrinter component. This means that each page in the fax, no matter how large, is scaled to fit exactly on one page of paper.

If no fax is loaded, the Print option does nothing.

Exit
Exits ViewFax.

The Edit menu

Select All
This option selects the fax page that is currently being viewed. The selection is displayed in inverted colors. Once the page is selected, the Copy option can be used to copy the selected image to the Windows clipboard.

Copy
This option copies the selection to the clipboard. You can select an entire fax page by using the Select All option. Smaller portions of the fax can be selected by clicking the mouse in the upper left hand corner of the image you wish to select and dragging the mouse to form a rectangle. The selected rectangle of image is shown on the screen in inverted colors.

The View Menu

Zoom In

This option causes the displayed fax to be increased in size by 25%. The default display scaling is 100%, or “normal” size. If you select the Zoom In option when the scaling settings are at the default, the fax is displayed at 125% of its normal size.

The maximum size at which an image can be displayed is 400%. Choosing Zoom In when the display is already scaled to 400% has no effect.

Zoom Out

This option causes the displayed fax to be decreased in size by 25%. The default display scaling is 100%, or “normal” size. If you select the Zoom Out option when the scaling settings are at the default, the fax is displayed at 75% of its normal size.

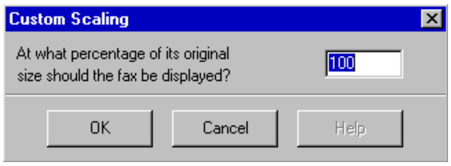
The minimum size at which an image can be displayed is 25%. Choosing Zoom Out when the display is already scaled to 25% has no effect.

25%, 50%, 75%, 100%, 200%, and 400%

Selecting a percentage causes the fax to be viewed at that percentage of its original size. When one of these options is chosen, a check mark appears next to the item indicating the current scaling settings.

Other

If the pre-defined scaling settings are not sufficient for your viewing, choose the Other option to manually enter the size at which you want the fax displayed. The dialog box shown in Figure 6.17 is displayed.


Custom Scaling

At what percentage of its original size should the fax be displayed? 100

Figure 6.17: Custom Scaling dialog box.

You can enter any number between 25 and 400, but even numbers and multiples of 5 usually produce the best results.

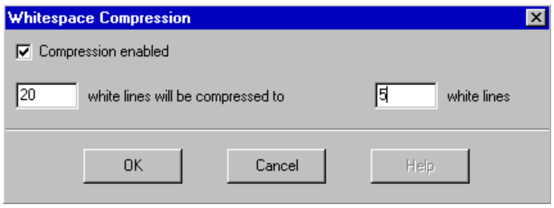
No Rotation, Rotate 90 degrees, Rotate 180 degrees, and Rotate 270 degrees

Because paper can be fed into a fax machine four different directions, it is sometimes useful to be able to rotate a displayed fax. For example, choosing 180 degree rotation properly displays a fax that was received upside-down. When the fax is rotated, a check mark appears next to the option that indicates the current rotation setting.

Whitespace Compression

ViewFax has the ability to compress large amounts of vertical white space in displayed faxes into smaller amounts of white space. Often, this results in being able to fit more of a displayed fax on the screen.

When the Whitespace Compression option is selected, the dialog box (shown in Figure 6.18) is displayed.



☑ Compression enabled

20 white lines will be compressed to 5 white lines

Figure 6.18: Whitespace Compression dialog box.

The check box at the top of the dialog determines whether white space compression is enabled. The two edit controls determine how the white space is to be compressed. In the example above, every instance of 20 or more blank lines will be compressed to 5 blank lines.

White space compression does not take effect immediately. The settings take effect when you next load a fax using the Open option.

Page Flags bar

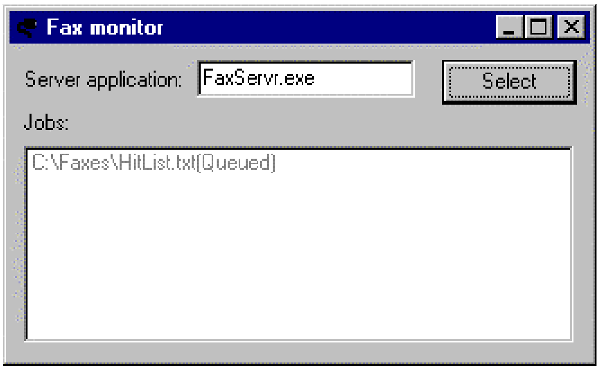
The Page Flags status bar displays the options currently in effect for the fax in the display. These flags represent properties of the fax being viewed. The High Res check box is checked if the fax is 200X200 and unchecked if the fax is 200X100. The High Width check box is checked if the fax width is 2048 pixels and unchecked if the fax width is 1728 pixels. The Length Words check box is checked if the raster lines of the fax include a length cardinal.

Fax Monitor and Fax Server Demo

FaxMon and FaxServr are two projects that work together to monitor fax printer print jobs and to send them once they are complete. FaxMon monitors the Async Professional fax printer driver for print jobs and notifies FaxServr. FaxServr retrieves the phone number to send the fax to, then sends the fax. These two projects do not use the TApdFaxServer, TApdFaxServerManager, or TApdFaxClient components.

FaxMon

The FaxMon project detects and monitors the print jobs sent to the Async Professional fax printer driver. The main form of FaxMon, shown in Figure 6.19, displays the print jobs as they are created, queued, and sent by the server application. FaxMon can be compiled as a TrayIcon for systems that support TrayIcons.



Server application: FaxServr.exe

C:\Faxes\HitList.txt(Queued)

Figure 6.19: FaxMon dialog box.

The Server application edit control designates the application that processes the fax once the fax printer driver has completed the print job. The Select button displays an open dialog where the server application can be selected.

The Jobs list box control displays the print jobs that have been sent to the fax printer driver. The name of the print job is shown along with the current status of the job. If the state is “Printing”, the document has been sent to the fax printer driver, but has not been completed. If the state is “Generated”, the fax printer driver has completed the conversion. If the state is “Queued”, the server application has been activated and the fax is waiting processing. If the state is “Sent”, the server application has sent the fax.

FaxServr

The FaxServr project receives the custom messages from FaxMon that are sent when a print job is complete. The FaxServr dialog box is shown in Figure 6.20. FaxServr gets the destination phone number, then sends the fax.

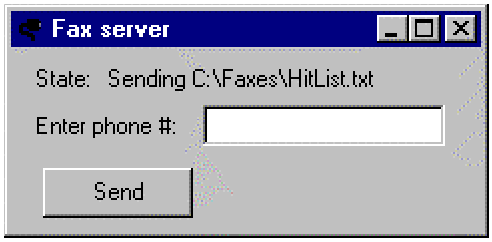


Figure 6.20: Fax server dialog box.

The State label displays the current state of the fax transfer. Enter the phone number of the receiving fax machine in the Enter phone # edit control. The Send button begins sending the fax to the designated number.

Fax Server Demo

FaxSrvX is a demonstration program which will monitor print jobs sent to the Async Professional fax printer driver and send the faxes. FaxSrvX is based on SendFax, see page 213 for additional descriptions, and does not use the TApdFaxServer, TApdFaxServerManager, or TApdFaxClient components.

When a print job is sent to the fax printer driver, the Add/modify/delete faxes dialog is shown. Enter the phone number of the receiving fax machine in the Phone number edit control. The Fax file name edit control will contain the name of the fax file that was generated by the fax printer driver. Enter the optional cover page in the Cover file name edit control. The Add button will add this fax to the list of pending faxes. The Cancel button will not add this fax to the list of pending faxes.

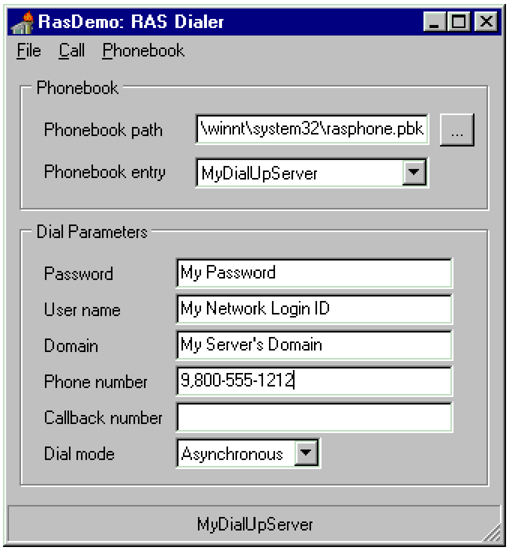
RAS Dialer Demo

RASDEMO is a simple RAS dialer program that can dial and manipulate RAS phonebooks. It is based on the TApdRasDialer component.

To use RASDEMO, you must have RAS is installed on your machine. RAS is installed by default on most Win95/98 machines. On NT machines, however, RAS is not installed until you add a modem device to the system configuration.

The main window

The main window, shown in Figure 6.21, consists of a menu bar, edit controls and a status line.



Phonebook path: \winnt\system32\rasphone.pbk
Phonebook entry: MyDialUpServer
Password: My Password
User name: My Network Login ID
Domain: My Server's Domain
Phone number: 9,800-555-1212
Callback number:
Dial mode: Asynchronous
MyDialUpServer

Figure 6.21: RasDemo main window.

The menu bar provides access to all of RasDemo's functions. From the menu you can dial, hangup and manipulate entries in a RAS phonebook. The edit controls allow you to specify the phonebook (Windows NT), the phonebook entry, and dialing parameters. The status line reflects the connection status during dialing.

The menu bar

The File menu

Exit

Exits RasDemo.

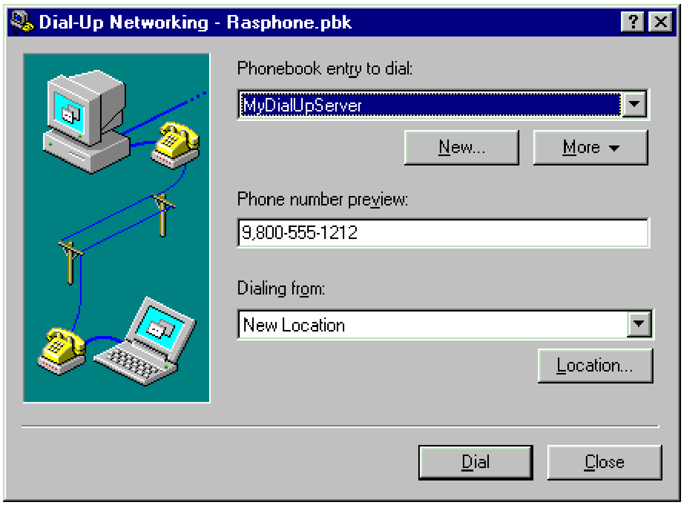
The Call menu

Dial (95/98/ME)

Initiates dialing for the specified phonebook entry and displays a connection status dialog during dialing and user ID authorization. This is the only dialing option available for Windows 95/98/ME users.

Phonebook dialog (NT/2000)

For Windows NT/2000 users, another dialing option is available via NT's main dial-up networking dialog box, shown in Figure 6.22. From this dialog box, you can choose a phonebook entry to dial, can edit, copy, or delete entries, and initiate dialing.



Phonebook entry to dial: MyDialUpServer

Phone number preview: 9,800-555-1212

Dialing from: New Location

Figure 6.22: Dial-Up Networking dialog box.

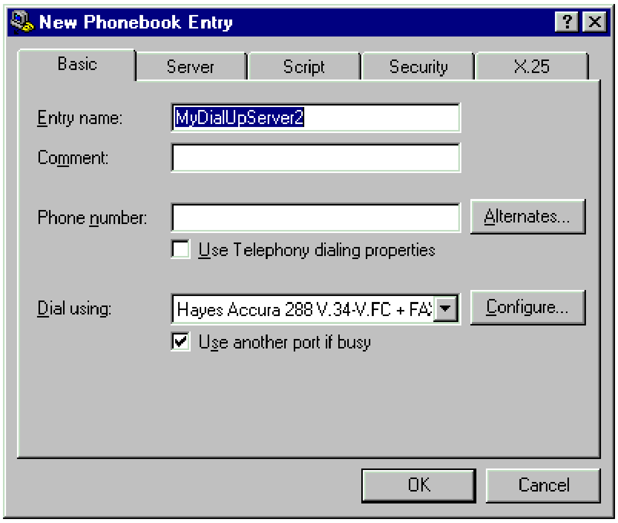
Hangup

Terminates a RAS connection.

The Phonebook menu

New entry

Invokes a multi-page dialog box, shown in Figure 6.23, that takes you through the process of creating a new phonebook entry.



Entry name: MyDialUpServer2

Comment:

Phone number:

☐ Use Telephony dialing properties

Dial using: Hayes Accura 288 V.34-V.FC + FA

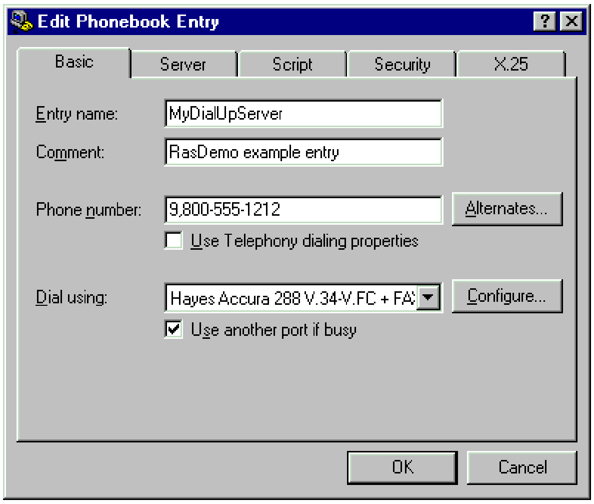
☑ Use another port if busy

Figure 6.23: New Phonebook Entry dialog box.

6

Edit entry

Invokes a multi-page dialog box, shown in Figure 6.24, that allows you to edit an existing phonebook entry. With this dialog you can change any configuration options.



Entry name: MyDialUpServer

Comment: RasDemo example entry

Phone number: 9,800-555-1212

☐ Use Telephony dialing properties

Dial using: Hayes Accura 288 V.34-V.FC + FA

☑ Use another port if busy

Figure 6.24: Edit Phonebook Entry dialog box.

Delete entry

Deletes the current entry from the current phonebook. If a connection has been established for the entry, it will be terminated. Note: this option is not available on early versions of Windows 95.

Refresh list

Refreshes the list of entries for the current phonebook in the Phonebook Entry combo box. This is useful to make sure the list is up to date after creating or deleting an entry.

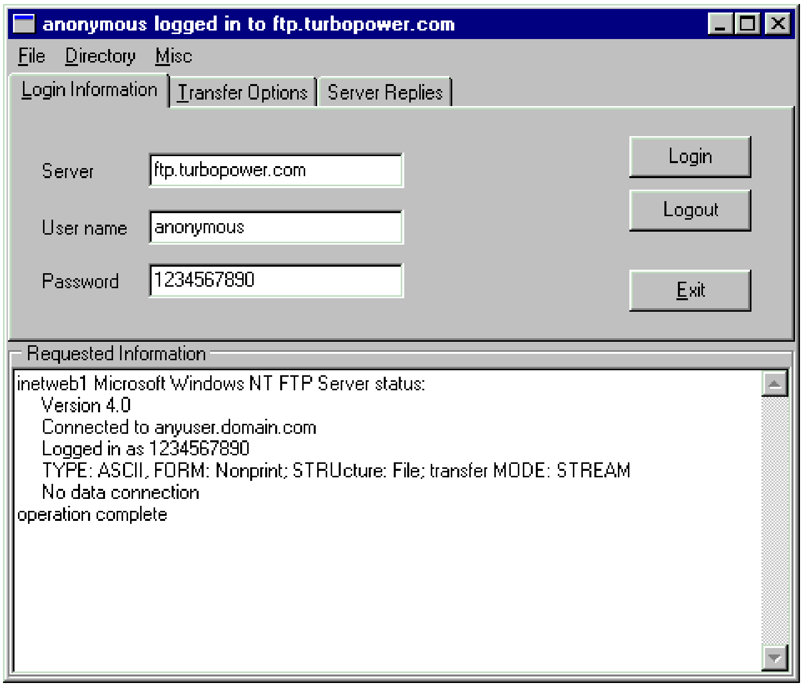
FTP Client Demo

FTPDEMO is a simple FTP client program that can connect to an FTP server, login, transfer files, display directory contents, etc. It is based on the TApdFtpClient component.

To use FTPDEMO, you need an existing Winsock/network connection. If you do not have an Inter/Intranet connection, the easiest way to make such a connection is usually through Dial-Up Networking or Remote Access Service.

The main window

The main window, shown in Figure 6.25, consists of a menu bar, three tabbed notebook pages, and an information display window.



Server: ftp.turbopower.com
User name: anonymous
Password: 1234567890

```
inetweb1 Microsoft Windows NT FTP Server status:
    Version 4.0
    Connected to anyuser.domain.com
    Logged in as 1234567890
    TYPE: ASCII, FORM: Nonprint; STRUcture: File; transfer MODE: STREAM
    No data connection
operation complete
```

Figure 6.25: FtpDemo main window.

The menu bar provides access to all of FtpDemo's functions. From the menu you can login in to an FTP server, transfer files, restart an interrupted file transfer, rename and delete remote files, list the contents of a remote directory, create and delete remote directories, obtain server status information, issue an FTP command string, and produce a log of FTP operations.

The tabbed notebook pages provides access to various user login ID information, file transfer options, and a window that displays the replies received from the server.

The information display window shows various remote directory and status information that is requested from the server.

The menu bar

The File menu

Login

Opens a control connection to an FTP server and logs in with the user login ID information specified on the Login Information page.

Logout/Quit

Logs the user out and closes the control connection to an FTP server.

Send

Uploads a local file from the local machine to an FTP server. A dialog prompts for the remote and local file names and initiates the transfer. If the remote file already exists, the file is replaced, appended, or given a unique name according to the settings on the Transfer Options page.

Receive

Downloads a remote file from an FTP server to the local machine. A dialog prompts for the remote and local file names and initiates the transfer. If the local file already exists, the file is replaced or appended to according to the settings on the Transfer Options page.

Rename

Renames a remote file. A dialog prompts for the remote file name and the new file name.

Delete

Deletes a remote file. A dialog prompts for the remote file name.

Exit

Exits FtpDemo.

The Directory menu

List

Obtains a List of the contents of a remote directory and displays it in the Requested Information window. If a Full listing is selected, various file information such as timestamp, size, and attributes are included. If a Names listing is selected, only the file names are displayed. A dialog prompts for the remote directory name. If no directory name is entered, the contents of the current working directory will be displayed.

Change
Changes the current working directory at the server. A dialog prompts for the remote directory name.

Create
Creates a new remote directory. A dialog prompts for the remote directory name.

Rename
Renames a remote directory. A dialog prompts for the remote directory name and the new directory name.

Delete
Deletes a remote directory. A dialog prompts for the remote directory name.

The Misc menu

Help
Obtains help information from the server and displays it in the Requested Information window. A dialog prompts for the FTP command. If a command is entered then the help information consists of the syntax for the command. Otherwise the help information consists of a list of the FTP commands that are supported by the server.

Server status
Obtains status information about a remote file, a remote directory, or the server itself and displays it in the Requested Information window. A dialog prompts for the remote path name. If the remote path name specifies a file, then the size of the file is displayed. If the remote path name specifies a directory, then a full listing is displayed. If no remote path is specified, then general server status information is displayed.

Send Ftp command
This option allows the user to issue an FTP command (as specified by RFC 959) directly to the server. A dialog prompts for the FTP command string. Note: commands requiring a separate data connection cannot be initiated by this method.

Log dialog
This options enables and disables FTP operation logging. When logging is enabled, a log entry is made to APROFTP.HIS each time an FTP operation is initiated. APROFTP.HIS is a text file located in the directory where the application is run.

Clear displays
This clears the Requested Information and Server Reply display windows.

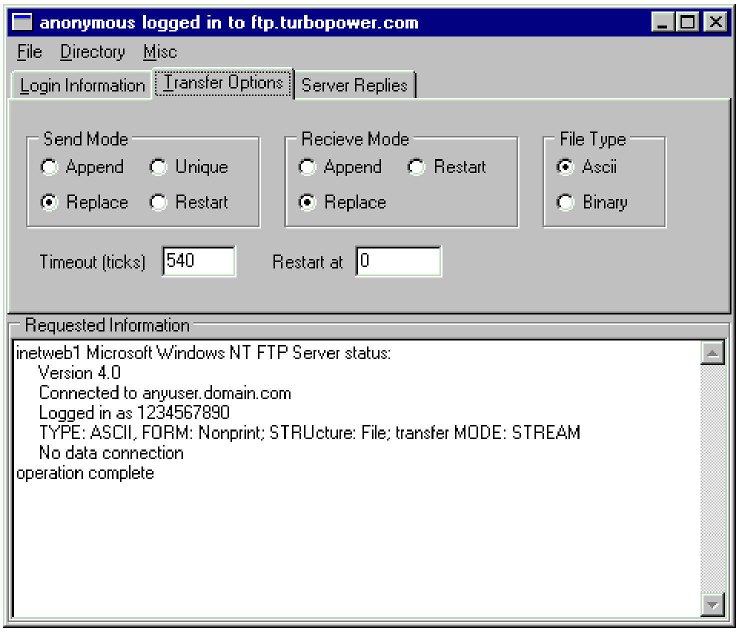
Login information

Use this page to specify the domain name of the FTP server, user login ID, and to log in to the server and log off.

Be sure to set these fields appropriately before attempting to login to an FTP server.

Transfer options

This page, shown in Figure 6.26, allows you to specify various file transfer options.



Send Mode: ( ) Append ( ) Unique [✓ Replace] ( ) Restart

Recieve Mode: ( ) Append ( ) Restart [✓ Replace]

File Type: [✓ Ascii] ( ) Binary

Timeout (ticks) 540

Restart at 0

Requested Information

```
inetweb1 Microsoft Windows NT FTP Server status:
    Version 4.0
    Connected to anyuser.domain.com
    Logged in as 1234567890
    TYPE: ASCII, FORM: Nonprint; STRUcture: File; transfer MODE: STREAM
    No data connection
operation complete
```

Figure 6.26: Transfer options page.

Send mode and receive mode determines how an existing destination file will be handled. If the destination file already exists:

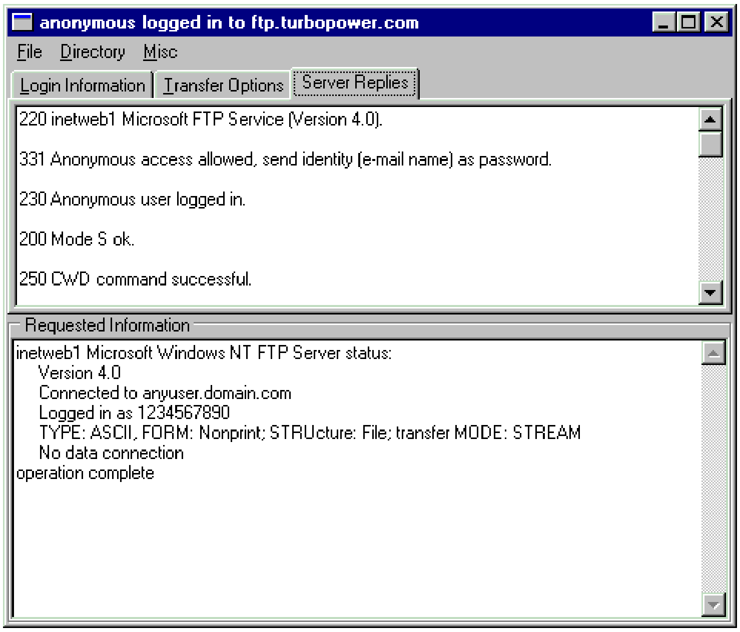
- Select Append to transfer the file data to the end of the existing file.
- Select Replace to overwrite the existing.
- Select Unique to transfer the file data to a unique file name created by the server.

If the FTP server supports resumable transfer, Restart indicates that the Send or Receive file transfer will be restarted at the byte location specified by Restart at.

You can also adjust the maximum time you want to wait for the server to replay to a command. The default value is 540 ticks which is roughly 30 seconds.

Server replies

This page, shown in Figure 6.27, consists of a window that displays replies received from the server. This window is cleared from the Clear Memo menu item.



```
220 inetweb1 Microsoft FTP Service (Version 4.0).

331 Anonymous access allowed, send identity (e-mail name) as password.

230 Anonymous user logged in.

200 Mode S ok.

250 CWD command successful.
```

```
inetweb1 Microsoft Windows NT FTP Server status:
    Version 4.0
    Connected to anyuser.domain.com
    Logged in as 1234567890
    TYPE: ASCII, FORM: Nonprint; STRUcture: File; transfer MODE: STREAM
    No data connection
operation complete
```

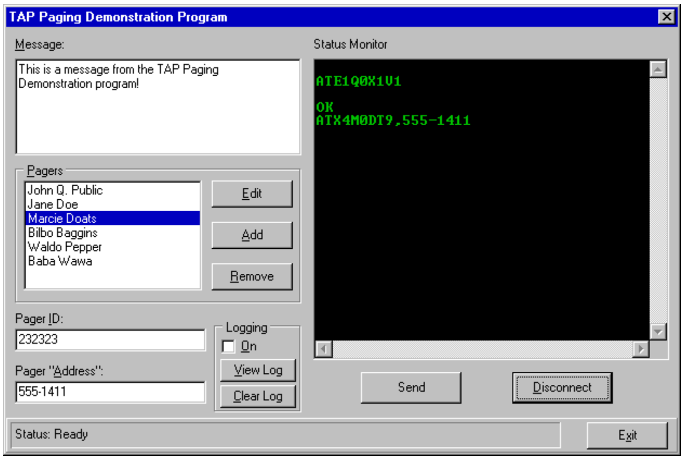
Figure 6.27: Sever Replies page.

Paging Demo

ExPaging is a simple paging program that allows the user to maintain a list of pager IDs and access addresses (TAP Paging Server phone numbers and/or SNPP IP addresses), and to send alphanumeric pages to them individually or in groups.

The main window

ExPaging does not use menus or toolbars. All of the primary options are available from the main form, shown in Figure 6.28, and paging operations are controlled by the buttons beneath the status display.



This is a message from the TAP Paging Demonstration program!

```
ATE1Q0X1V1

OK
ATX4M0DT9,555-1411
```

John Q. Public
Jane Doe
Marcie Doats
Bilbo Baggins
Waldo Pepper
Baba Wawa

Pager ID: 232323

Pager "Address": 555-1411

Status: Ready

Figure 6.28: ExPaging main window.

The right half of the main form contains the paging status display which shows messages indicating the progress of a page and can aid in diagnosing difficulties with sending to a particular address or pager ID. Use the buttons beneath the status display send a page (or group of pages) and to cancel the current page (or group).

On the left side of the main form, from the top, are an edit area to enter the message to be sent, the list of pagers currently being maintained by ExPaging (along with buttons for managing the list, see “Managing the pager list” on page 240). Multiple pagers may be selected in the list of pagers to have the same page sent to each of them sequentially.

Beneath the list are two edit boxes that automatically update for the currently selected pager in the list. A Pager ID and Access Address may be entered manually in the respective edit boxes for one-shot pages; these are not remembered by the program however.

Next to the Edit boxes are a check box and two buttons for controlling the logging features of ExPaging (see “Page logging” on page 241).

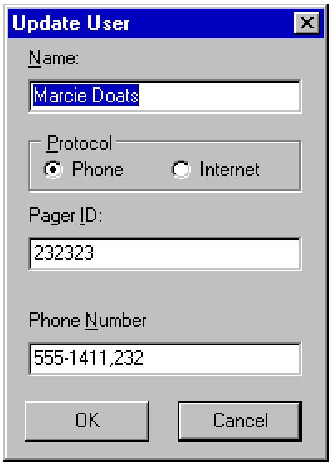
Finally, at the bottom of the screen is a status line with a display showing the current paging state and button to exit the program (exiting the program via this button, or the close button on the title bar cancels the current paging operation just as if you had clicked the Disconnect button before the program exits.

Managing the pager list

ExPaging manages a collection of pager IDs and access addresses. These are maintained in a plain text file, PAGERS.LST, which should reside in the directory/folder where ExPaging itself resides.

Next to the list of pagers on the main form are three buttons: Add, Edit, and Remove. ExPaging can perform all relevant maintenance on the pager list using these buttons, so it should be generally unnecessary to edit the page list file directly.

Add and Edit both bring up a secondary form, the Add/Edit User form (shown in Figure 6.29). On this form are three edit boxes and a Radiogroup.



Name:
Marcie Doats

Protocol
[✓ Phone] Internet

Pager ID:
232323

Phone Number
555-1411,232

Figure 6.29: Update User dialog box.

The first edit box is for an identifier for the pager; generally this will be the user’s name but can be in pretty much anything as long as it is unique among all pager entries.

So, if you ever need to enter more than one pager for an individual (e.g., someone who has both a TAP and an SNPP pager), these must be made distinct: “John Doe (TAP)” vs. “John Doe (SNPP)”.

Next comes the Protocol RadioGroup which allows selection of the paging protocol; either via phone line/modem (TAP) or over an Internet or TCP/IP connection (SNPP). For the entry illustrated above, the default is TAP.

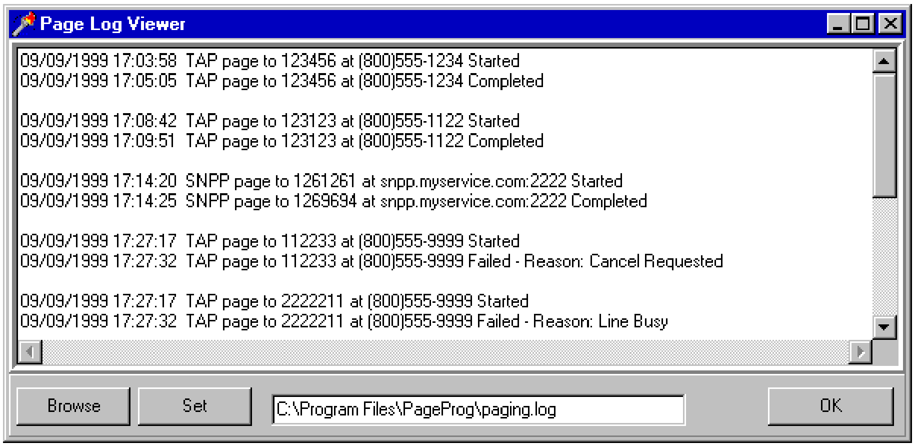
Third is the edit box for the Pager ID and last is a box to enter the paging server “address” (TAP phone number or SNPP IP address and port). The format for an SNPP address is IP Address:Port.

OK closes the form and causes the main form to add or update the entry. Cancel closes the form and no changes are made.

Page logging

ExPaging includes the facility to keep track of sent pages by making notations in a paging log.   This facility is managed by the controls in the Logging group box. First is a check box to determine if logging occurs at all; second is a button to view the log current log, third is a button to clear the current log and start fresh.

Clicking the View button brings up the View Pager Log form, shown in Figure 6.30.



```
09/09/1999 17:03:58  TAP page to 123456 at (800)555-1234 Started
09/09/1999 17:05:05  TAP page to 123456 at (800)555-1234 Completed

09/09/1999 17:08:42  TAP page to 123123 at (800)555-1122 Started
09/09/1999 17:09:51  TAP page to 123123 at (800)555-1122 Completed

09/09/1999 17:14:20  SNPP page to 1261261 at snpp.myservice.com:2222 Started
09/09/1999 17:14:25  SNPP page to 1269694 at snpp.myservice.com:2222 Completed

09/09/1999 17:27:17  TAP page to 112233 at (800)555-9999 Started
09/09/1999 17:27:32  TAP page to 112233 at (800)555-9999 Failed - Reason: Cancel Requested

09/09/1999 17:27:17  TAP page to 2222211 at (800)555-9999 Started
09/09/1999 17:27:32  TAP page to 2222211 at (800)555-9999 Failed - Reason: Line Busy
```

Figure 6.30: Page Log Viewer form.

The main TMemo area shows the contents of the currently selected Paging Log. You can view other Logs (or any text file actually) by clicking Browse and picking another file with the resulting OpenFile dialog box.

The Set button makes the currently selected file the default Log file for ExPaging; any new page logging entries will be appended to that file. If you have entered a name for a file that doesn’t exist, ExPaging will create the file and use it as the log file.

OK closes the Log file viewer and causes ExPaging to update which Log file is used, if that was changed.

Glossary

This glossary contains a combination of industry accepted definitions and, where noted, definitions that are unique to Async Professional.

Alphanumeric paging

An extension of the numerical paging capability. Alphanumeric paging allows transmission of general textual information to paging devices/receivers.

ANSI

American National Standards Institute. In Async Professional, references to ANSI usually refer to the ANSI standard for terminal control as exemplified by the DOS ANSI.SYS device driver.

asynchronous serial communication

Serially transmitted data in which each character is surrounded by start and stop bits. That is, each character can be extracted from the data stream without making assumptions, based on time, about when characters start and stop.

AT commands

An industry-standard set of commands for controlling modems introduced with the Hayes SmartModem.

baud rate

A measure of modulation rate, not communication speed. Technically, baud rate means the number of signal changes per second. At the UART, baud rate is generally equal to bps (bits per second), since each signal change represents one bit. When using modems, however, baud rate is generally different than bps, since the modulation schemes used by modems typically encode more than one bit per signal change. That's why modem speeds are typically rated in terms of bps.

Bell 103

The AT&T modem standard for asynchronous communication at speeds up to 300 bps.

Bell 212A

The AT&T modem standard for asynchronous communication at speeds up to 1200 bps on dial-up telephone lines.

bps

Bits per second, a measure of raw communications speed, which quantifies how fast the bits within a character are being transmitted or received. It is not a measure of overall throughput, but rather a measure of the speed with which a single character can be processed.

break

A signal that can be transmitted or received over serial communication links. A break is not a character, but rather a condition in which the serial line is held in the “0” state for a least one character-time.

client

An application that connects to a server for the purpose of exchanging of data.

CCITT

Comité Consultatif International de Télégraphique et Téléphonique (International Telegraph and Telephone Consultative Committee). A European communications standards committee, which has recently been renamed to ITU-TSS.

character (in terminal emulation)

A binary value, usually byte-sized, the visual representation of which is controlled by font selection and character mapping tables.

character set mapping table

A list of character ranges in character sets and the fonts and glyphs that should be used to display them.

character-time

This term is used to mean the amount of time between the start bit and stop bit of a serial byte (inclusive). This is the smallest period of time between successive received or transmitted characters (of course, the elapsed time between characters can be longer than one character-time).

checksum

A byte, or bytes, appended to the end of a block of data that is used to check the integrity of that block. A checksum is the sum of all the bytes in the block.

comport

In this manual, refers to a TApdComport component, or a component derived from TApdCustomComport (such as TApdWinsockPort). This convention is used to reduce confusion between the physical port and the comport component. Outside of Async Professional's documentation, it is not uncommon to see “comport”; “com port” and “serial port” being used synonymously.

COMM.DRV

The Windows device driver that performs all of the low-level work required to send and receive using the PC's UART chip in 16-bit Windows.

CRC

A byte, or bytes, appended to the end of a block of data that is used to check the integrity of the data. CRC is short for cyclical redundancy check, a data checking algorithm that provides a much higher level of protection than a simple checksum.

CTS

Clear to send. This is a modem control signal that is raised by the modem when it is ready to accept characters. The modem may lower this line when it cannot accept any more characters (this usually means that its receive buffer is nearly full). This behavior is called hardware handshaking or hardware flow control.

data bits

The bits in a serial stream of data that hold data as opposed to control information. The number of data bits is one of the line parameters needed to describe a serial port configuration. The acceptable values are 5 through 8.

data compression

Refers to the ability of some modems to compress data before passing it to the remote modem. There are two standards that describe data compression methods, MNP and V.42bis.

DCB

Device control block. A structure passed from a Windows program to the communications driver. It contains the line parameters and other configuration information that the communication driver uses to configure the UART.

DCD

Data carrier detect. A signal provided by a modem to indicate that it is currently connected to a remote modem.

DCE

Data communications equipment. Generally, this refers to a modem.

device layer

This layer of Async Professional provides the physical connection between the software and the hardware.

DNS

A remote database that contains a list of host names and their corresponding IP addresses.

dot notation

A way of specifying an IP address (e.g., 165.212.210.12).

DSR

Data set ready. This is a modem control input signal to a UART that tells the UART that the remote device (usually a modem) is active and ready to transmit data.

DTE

Data terminal equipment. Generally, this refers to a terminal or a PC emulating a terminal.

DTR

Data terminal ready. This is a modem control signal raised by a UART to notify the remote (usually a modem) that it is active and ready to transmit.

emulation

Refers to a PC program that mimics the “appearance” and functionality of a terminal in such a fashion that the server computer is not aware that there is no real terminal present.

error correction

Refers to the ability of some modems to check the integrity of data received from a remote modem. There are two standards that describe error correction protocols, MNP and V.42.

escape sequences

Terminal control sequences in the stream of data coming into the terminal.

FIFO mode

A mode of operation for 16550 UARTs that takes advantage of the UART’s first-in-first-out buffers.

flow control

A facility that allows either side of a serial communication link to request a temporary pause in data transfer. Typically, such pauses are required when data is being transferred faster than the receiver can process it. Hardware flow control is implemented via changes in the CTS and RTS signals. Software flow control is implemented via the exchange of XOn and XOff characters.

full duplex

1. A mode of communication in which the receiving computer automatically echoes all data it receives back to the transmitter. 2. A communications link that can pass data both directions (receive and transmit) at the same time.

glyph

The visual representation of a character.

half duplex

1. A mode of communication in which the receiving computer does not echo any data back to the transmitter. 2. A communications link that can pass data in only one direction at a time.

handshaking

Refers to the initial transfers of data between two processes. Usually this term is used to describe the start of a protocol file transfer or the exchange of data that occurs when two modems first connect.

host name

The text description of an IP address (e.g., joeb.turbopower.com).

interface layer

The layer of Async Professional that contains the majority of the application programming interface (API). This layer is implemented by the TApdComPort component.

IP address

The 32-bit address of a network computer. All IP addresses are unique.

IRQ

One of the lines on the PC or PS/2 bus that is used to request a hardware interrupt. Any device that needs to interrupt the CPU (such as a UART) does so via an IRQ line.

ITU-TSS

International Telecommunications Union-Telecommunications Standardization Sector. A European communications standards committee, formerly known as CCITT.

LAP M

An error-correction protocol included with the most recent CCITT communications standard V.42.

line error

Refers to one of the following errors: UART overrun, parity error, or framing error. Such errors are due either to interference picked up by the physical connection (cable, phone line, etc.) or to a mismatch in line parameters between the two ends of a serial link.

lookup

An action that Winsock performs to retrieve the IP address for a host name, or to retrieve the port number for a service name (and vice versa).

MNP

Microcom Networking Protocol. A communications protocol designed by Microcom, Inc. and placed in the public domain. MNP defines several service levels that provide error control and data compression facilities between two modems. MNP is of interest only if you are using modems that support it. See the modem manual for more information about the details of MNP.

modem

A device that facilitates serial communication over phone lines. The term is derived from the phrase MOdulation/DEModulation device.

network shared-modem pool

A collection of modems in a network that are available to any PC in the network. In a typical situation, several modems are attached to one PC (a “modem server”) and other PCs on the network use a network protocol to access these modems.

paging

Originally, simply a means of notifying someone to call back to an answering service; and later a means of transmitting numeric data to devices with limited display capabilities; the definition of paging has been broadened to include the transmission of general textual information to portable electronic devices.

paging device

Also “paging receiver” or "pager". The usually small electronic device capable of receiving paging transmissions, originally restricted to dedicated “pagers”, many modern cell phones and other personal electronic devices can also serve as paging receivers. Each has an identifier (often called a PAN: “Personal Access Number”) which allows the device to be uniquely identified and contacted.

paging receiver

See “paging device”

paging server

A device and/or software that manages requests for pages and transmits them to the appropriate paging devices/receivers.

parity

A bit that is used to check the integrity of a byte. The parity bit is set by the transmitter and checked by the receiver. If present, the parity bit is set so that the sum of the bits in the character is always odd or always even. The parity bit can also be set to a constant value (always on or always off).

port (Winsock)

A number from 0 to 32767 that, along with the IP address, is used to create a socket.

protocol

Generally, an agreed upon set of rules that both sides of a communications link follow. This term crops up in two places in Async Professional: file transfer protocols and modem protocols. A file transfer protocol is a set of rules that two computers use to transfer one or more files. A modem protocol describes the modulation technique as well as the error control and data compression rules.

remote device

In Async Professional, this term is used to describe what's attached to your serial port. Since it can be another PC, a different kind of computer, a modem, an instrument, or another device, we often just say "remote" or "remote device."

RI

Ring indicator. A signal provided by the modem to indicate that a call is coming in (i.e., the phone is ringing).

RS-232

An EIA (Electronic Industries Association) standard that provides a physical description (voltages, connectors, pin names, and purposes) of a serial asynchronous communications link. This is the standard used by the IBM PC's Asynchronous Communications Adapter (and compatibles). The original intent of RS-232 was to describe the link between a computer and a modem. However, many devices other than modems (printers, plotters, laboratory instruments, and so on) have adopted some of the conventions of RS-232.

RTS

Request to send. This is a modem control signal that the UART uses to tell the modem that it is ready to receive data.

S-registers

A register in a Hayes-compatible modem that stores configuration information. Lower numbered S-registers are somewhat standardized, but higher numbered S-registers are generally used for different purposes by different modem manufacturers.

script

A list or file containing communications commands. Script languages are often provided by general-purpose communications programs (such as Kermit, Telix and Procomm Plus) to automate standard operations.

scrollback view

When a terminal is in scrollback view, it shows a history of data that have scrolled off the top of the display view.

scrolling region

A range of lines within which writes to the screen and scrolling are restricted.

serial data

Refers to data transmitted over a single wire where bits are represented as either high or low signals over a specified period of time. This is in contrast to parallel data, where each bit is represented by its own "wire."

server

An application that listens on a socket for client connection attempts.

SNPP

Simple Network Paging Protocol, a formal specification of transmitting alphanumeric page requests over TCP/IP networks (e.g. Internet). A paging service must provide an SNPP server as a facility before a pager may be contacted via SNPP.

socket

A Windows object that is created using a combination of an IP address and port number. A socket is used to make a network connection between two computers.

start bit

The bit in a serial stream that indicates a data byte follows. This value cannot be changed; UART communications always uses one start bit.

stop bits

The bits in a serial stream that indicate all data bits were sent. One or two stop bits can be used. The number of stop bits is one of the line parameters needed to describe a serial link.

streaming protocol

A file transfer protocol that doesn't require an acknowledgement for each block. Such protocols are usually much faster than non-streaming protocols because the transmitter never pauses to wait for an acknowledgement.

TAP

Telelocator Alphanumeric Protocol, a formal specification for transmitting alphanumeric page requests over a telephone line, typically using a modem. TAP includes means of entering pages manually via the phone keypad or a terminal program, however these procedures are awkward and error-prone. APRO's support of TAP is directed at its specification for direct computer transmission of paging requests. TAP is also known as the “IXO” protocol or the Motorola “PET” (Personal Entry Terminal) protocol.

Telnet

A network protocol designed to allow two network computers to communicate via a terminal screen.

terminal emulator

Software that interprets special sequences of characters as video control information (for setting colors, positioning the cursor, etc.) rather than data. This process is referred to as “emulation” because it emulates the behavior built into serial terminals (such as the DEC VT100 terminal).

terminal

A device (or software) that displays received data to a CRT and transmits keyboard characters to a host computer. A “dumb terminal” is one that does no local processing of the data it receives from the host. A “smart terminal” is capable of interpreting special “escape sequences,” allowing the host to move the terminal’s cursor, change the colors used to display text, etc.

trigger

An Async Professional term describing an event or condition noted by the internal dispatcher and passed to an application through a VCL event handler.

UART

An acronym for Universal Asynchronous Receiver Transmitter. This is the device (usually one integrated circuit) that serializes and deserializes data between the CPU and the serial data line.

V.17

CCITT 7200, 9600, 12000, and 14400 bps faxmodem standard.

V.21

CCITT 300 bps faxmodem standard.

V.22

CCITT 1200 bps modem standard.

V.22bis

CCITT 2400 bps modem standard.

V.25bis

CCITT communications command set. Frequently implemented in addition to the AT command set.

V.27, V.27 ter

CCITT 2400 and 4800 bps faxmodem standard.

V.29

CCITT 7200 and 9600 bps faxmodem standard.

V.32

CCITT 9600 bps communications standard which describes a standard modem modulation technique. Any 9600 baud modem that complies with V.32 can connect to any other V.32 compliant modem (this is an improvement from the early days of 9600 bps communication when only modems from the same manufacturer could connect to each other).

V.32bis

CCITT standard for data modem modulation rates up to 14400 bits per second.

V.34

CCITT 28800 bps communication standard which describes a standard modem modulation technique. V.34 includes several advanced features designed to get as much performance as possible out of a given telephone connection. The top speed of 28800 bps occurs only under optimal conditions; normal telephone conditions usually yield lower throughput, but still substantially higher than V.32.

V.42

CCITT error correcting protocol standard. Includes both MNP-4 and LAP-M error correction protocols.

V.42bis

CCITT 4:1 data compression protocol. This data compression scheme generally achieves a much higher degree of compression than is possible with MNP.

V.FC/V.Fast

An early unratified version of the V.34 specification. V.34 modems can usually connect to V.FC and V.Fast modems, but usually at lower rates than with other V.34 modems.

Glossary

Index

For over fifteen years you've depended on TurboPower to provide the best tools and libraries for your development tasks. Now try Sleuth QA Suite 3 and Orpheus 4—two of TurboPower's best selling products—risk free. Both are fully compatible with Borland Delphi and C++Builder, and are backed with our expert support and 60-day money back guarantee.

2001 READERS CHOICE Delphi INFORMANT MAGAZINE COMPANY OF THE YEAR

The TurboPower family of tools-
Winners of 6 *Delphi Informant* Readers' Choice Awards for 2001! Company of the Year in 2000 and 2001.

DEVELOP, DEBUG, OPTIMIZE

FROM START TO FINISH, TURBOPOWER

HELPS YOU BUILD YOUR BEST

SLEUTH QA SUITE 3™

Want to make your code the best it can be? More than just a debugger and profiler, Sleuth QA Suite 3 is a full-cycle debug/optimization/test solution that helps you meet quality assurance requirements from coding through product deployment and beyond. No expensive add-ons are required—everything you need is in one box.

ORPHEUS 4™

Make the first impression count with new Orpheus 4, the award-winning user interface construction kit from TurboPower Software Company. Your customers will feel comfortable the minute your application starts, resulting in them spending less time learning how to use your program and more time getting to know what makes it special.

Try the full range of TurboPower products. Download free Trial-Run Editions from our Web site.

www.turbopower.com

Async Professional 4 requires Microsoft Windows (9x, Me, NT, 2000 or XP) and Borland Delphi 3 and above, or C++Builder 3 and above

TURBOPOWER® Software Company