

VCL components for advanced communications

Async ProfessionalTM4

REFERENCE GUIDE

The comprehensive guide to using Async Professional

Async Professional 4TM

Reference Guide

TurboPower Software Company
Colorado Springs, CO

www.turbopower.com

© 1998-2001 TurboPower Software Company. All rights reserved.

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

License Agreement

This software and accompanying documentation are protected by United States copyright law and also by International Treaty provisions. Any use of this software in violation of copyright law or the terms of this agreement will be prosecuted to the best of our ability.

Copyright © 1998-2001 by TurboPower Software Company, all rights reserved.

TurboPower Software Company authorizes you to make archival copies of this software for the sole purpose of back-up and protecting your investment from loss. Under no circumstances may you copy this software or documentation for the purposes of distribution to others. Under no conditions may you remove the copyright notices made part of the software or documentation.

You may distribute, without runtime fees or further licenses, your own compiled programs based on any of the source code of Async Professional. You may not distribute any of the Async Professional source code, compiled units, or compiled example programs without written permission from TurboPower Software Company.

Note that the previous restrictions do not prohibit you from distributing your own source code, units, or components that depend upon Async Professional. However, others who receive your source code, units, or components need to purchase their own copies of Async Professional in order to compile the source code or to write programs that use your units or components.

The supplied software may be used by one person on as many computer systems as that person uses. Group programming projects making use of this software must purchase a copy of the software and documentation for each member of the group. Contact TurboPower Software Company for volume discounts and site licensing agreements.

This software and accompanying documentation is deemed to be "commercial software" and "commercial computer software documentation," respectively, pursuant to DFAR Section 227.7202 and FAR 12.212, as applicable. Any use, modification, reproduction, release, performance, display or disclosure of the Software by the US Government or any of its agencies shall be governed solely by the terms of this agreement and shall be prohibited except to the extent expressly permitted by the terms of this agreement. TurboPower Software Company, 15 North Nevada Avenue, Colorado Springs, CO 80903-1708.

With respect to the physical media and documentation provided with Async Professional, TurboPower Software Company warrants the same to be free of defects in materials and workmanship for a period of 60 days from the date of receipt. If you notify us of such a defect within the warranty period, TurboPower Software Company will replace the defective media or documentation at no cost to you.

TurboPower Software Company warrants that the software will function as described in this documentation for a period of 60 days from receipt. If you encounter a bug or deficiency, we will require a problem report detailed enough to allow us to find and fix the problem. If you properly notify us of such a software problem within the warranty period, TurboPower Software Company will update the defective software at no cost to you.

TurboPower Software Company further warrants that the purchaser will remain fully satisfied with the product for a period of 60 days from receipt. If you are dissatisfied for any reason, and TurboPower Software Company cannot correct the problem, contact the party from whom the software was purchased for a return authorization. If you purchased the product directly from TurboPower Software Company, we will refund the full purchase price of the software (not including shipping costs) upon receipt of the original program media and documentation in undamaged condition. TurboPower Software Company honors returns from authorized dealers, but cannot offer refunds directly to anyone who did not purchase a product directly from us.

TURBOPOWER SOFTWARE COMPANY DOES NOT ASSUME ANY LIABILITY FOR THE USE OF ASYNC PROFESSIONAL BEYOND THE ORIGINAL PURCHASE PRICE OF THE SOFTWARE. IN NO EVENT WILL TURBOPOWER SOFTWARE COMPANY BE LIABLE TO YOU FOR ADDITIONAL DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS, OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF OR INABILITY TO USE THESE PROGRAMS, EVEN IF TURBOPOWER SOFTWARE COMPANY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

By using this software, you agree to the terms of this section and to any additional licensing terms contained in the DEPLOY.HLP file. If you do not agree, you should immediately return the entire Async Professional package for a refund.

All TurboPower product names are trademarks or registered trademarks of TurboPower Software Company. Other brand and product names are trademarks or registered trademarks of their respective holders.

Table of Contents

Chapter 1: Introduction	1
Files Supplied	4
The Component Hierarchy	8
Organization of this Manual	17
Technical Support	19
Chapter 2: Port Component	21
TApdComPort Component	22
Chapter 3: Winsock Components	99
TApdSocksServerInfo Class	103
TApdWinsockPort Component	106
TApdSocket Component	115
Chapter 4: Data Packet Component	131
TApdDataPacket Component	132
Chapter 5: Script Component	145
TApdScript Component	146
Chapter 6: State Machine Components	165
TApdStateMachine Component	172
TApdState Component	178
Chapter 7: Status Light Components	187
TApdStatusLight Component	188
TApdSLController Component	191
Chapter 8: The Terminal Components	195
Terminal Design Considerations	196
TAdTerminalBuffer Component	201
The Terminal Parsers	224
TAdTerminalParser Class	229
TAdVT100Parser Class	234
The TAdKeyboardMapping Class	238
The TAdCharSetMapping Class	245
The TAdTerminalEmulator Class	253
The TAdTTYEmulator Class	262
The TAdVT100Emulator Class	263

The TAdTerminal Component	269
Chapter 9: IP Telephony	293
IP Telephony in Async Professional	294
Configuration for VoIP	295
TApdVoIPTerminal Class	298
TApdVoIP Component	301
Chapter 10: SAPI Components	309
SAPI Overview	310
TApdAudioOutDevice Class	314
TApdAudioInDevice Class	324
TApdCustomSapiEngine Component	333
Speech Synthesis Tags	334
TApdSapiEngine Component	337
TApdSapiPhonePrompts	351
TApdCustomSapiPhone Component	356
TApdSapiPhone Component	359
Chapter 11: Remote Access Service (RAS) Components	371
TApdRasDialer Component	372
TApdRasStatus Component	387
Chapter 12: TAPI Components	389
TAPI Device Control from an Application	392
TApdTapiDevice Component	407
TApdAbstractTapiStatus Class	440
TApdTapiStatus Component	443
TApdTapiLog Class	445
Chapter 13: Modem Components	447
modemcap and libmodem	448
TApdLibModem Component	449
TAdModem Component	461
TAdModemStatus Component	478
Chapter 14: File Transfer Protocols	483
General Issues	485
Xmodem	501
Ymodem	504
Zmodem	507
Kermit	513
ASCII	519

FTP	521
TApdProtocol Component	523
TApdFtpClient Component	559
TApdAbstractStatus Class	578
TApdProtocolStatus Component	582
TApdProtocolLog Component	583

Chapter 15: Fax Components 587

Faxmodem Control from an Application	588
Document Conversion	591
TApdFaxConverter Component	594
TApdFaxUnpacker Component	624
TApdFaxViewer Component	649
TApdFaxPrinter Component	674
TApdAbstractFaxPrinterStatus Class	689
TApdFaxPrinterStatus Component	693
TApdFaxPrinterLog Component	695
Sending and Receiving Faxes	698
TAPI/Fax Integration	704
TApdAbstractFax Component	715
TApdSendFax Component	736
TApdReceiveFax Component	754
Fax Server Components	764
TApdFaxJobHandler Component	769
TApdFaxServer Component	779
TApdFaxServerManager Component	809
TApdFaxClient Component	815
TApdAbstractFaxStatus Class	822
TApdFaxStatus Component	826
TApdFaxLog Class	828
Fax Printer Drivers	830
TApdFaxDriverInterface Component	834

Chapter 16: Paging Components 837

Sending Alphanumeric Pages	838
TApdAbstractPager Component	839
TApdTAPPager Component	841
TApdSNPPPager Component	856
TApdPagerLog Component	862
TApdGSMPhone Component	865
TApdSMSMessage Class	872
TApdMessageStore Class	875

TApdGSMPhone Component	878
Chapter 17: Low-level Facilities	887
Timers	888
Name Routines	894
Chapter 18: Appendices	899
Error Handling and Exception Classes	900
Conditional Defines	920
Glossary	921
Debugging Windows Communications Programs	930
Identifier Index	i
Subject Index	xvii

Async Professional is a collection of native Visual Component Library (VCL) components that provide serial communication facilities for programs created with Borland Delphi and C++Builder. It provides optimized components that are fully integrated with Delphi, compile directly into your EXE files, and include complete source code. Async Professional (APRO) provides a wide range of communication components, including:

- A communications port component with standard serial port properties (port number, baud rate, and so on), methods for sending and receiving data, and events for common communications situations (data available, buffer empty, and so on).
- A flexible data packet component that informs you when data that meets your criteria arrives at the communications port.
- New state machine components that let you design and implement protocols.
- New SAPI components to add Speech to your applications. Now your applications can speak (Text to Speech) and listen (Speech to Text).
- New IP Telephony components to implement full streaming audio and video over your network.
- New Non-TAPI modem database using TurboPower's NEW modemcap XML format. Use the TAPI modem definitions (from the INF files) to control your modem when TAPI doesn't cut it.
- New SMS pager component to take advantage of the Short Message System.
- A scripting component that contains properties and methods for automating basic communication operations like logging on and off, file upload and file download.
- A communications port component that provides network and Internet communications using Winsock, in addition to the standard communications port capabilities.
- A RAS dialing component that gives you more control over your Dial-Up Networking via the Remote Access Server API.
- File Transfer Protocol (FTP) components that take care of the FTP protocol details and present a friendly interface, allowing you to transfer huge files from the Internet and support resumable transfers. An FTP logging component automates the process of logging an FTP client-server dialog for auditing FTP activities.

- TAPI components for working with modems in TAPI environments like Windows 95/98, Windows NT 4.0, and Windows 2000.
- A new modem component that provides a simple interface for accessing the most commonly used modem operations. TAdModem integrates the selection of the modem from the new modem database and the dialog to show the current status of the modem.
- An advanced terminal the provides full support for VT100 protocol.
- StatusLight components that react to changes in serial port status and reflect the status of the port.
- A file transfer protocol component for transferring files using an Xmodem, Ymodem, Zmodem, Kermit, or ASCII protocol.
- File transfer status and file transfer logging components to display the progress of a file transfer and create a history file of files sent or received.
- Paging components for sending alphanumeric pages with Telelocator Alphanumeric Protocol (TAP), Personal Entry Terminal Protocol (PET), internet based paging using the Simple Network Paging Protocol (SNPP), and Short Message Services (SMS).
- Fax Client and Server components that make it easy to create a distributed fax server system.
- A fax conversion component that converts color BMP, monochrome PCX, DCX, TIFF and text files to a faxable format, and a fax unpacking component that unpacks received faxes into image files or memory bitmaps. Components for printing and viewing faxes are also included.
- Fax printer drivers and an interface component that provide a print-to-fax feature from any Windows program.
- Fax send and receive components for transmitting and receiving fax files using Class 1, Class 1.0, Class 2 and Class 2.0 faxmodems.

Deprecated components

As Async Professional has matured through the years, several components have become obsolete, or have been replaced by components with greater functionality. Some of these components have been deprecated to allow APRO to evolve, while still maintaining some degree of backwards compatibility.

Previous version of APRO have moved the deprecated components to a separate tab on the component palette, this version of APRO has deprecated even that. The deprecated components are installed on your installation destination folder in the \Bonus folder. The units in this folder contain the component source for several components that can be installed in your palette.

We do not plan to make any enhancements to these components, and technical support for these products will have a very low priority. These components may be completely removed from future versions of APRO. In short, we highly recommend that you do not use these components for new development.

The following components are now deprecated:

- TApdIniDBase, TApdModemDBase, AwModem.ini: These components and files were used for modem configuration and phone book databases. They have been replaced by the TApdLibModem component and the modemcap database.
- TApdModem, TApdSModem: These files were used for non-TAPI modem control using the TApdModemDBase component. They have been replaced by the TAdModem and TApdLibModem components.
- Modem dialer and status components using the TApdModem and TApdSModem components.
- Phonebook and phonebook editor components.
- Terminal window, terminal emulator and keyboard emulator components that allow you to add ANSI, VT52 or VT100 terminals to your application. Replaced by the TAdTerminal and associated components.

Each of the units containing installable deprecated components are duplications of the distributed 3.0x source files. To install these components, you may have to add the registrations methods, or add the units to a custom package.

Documentation for the deprecated components is included in the APRODEP.HLP file, installed in the \Bonus folder.

Files Supplied

Installation information is provided in the Async Professional Developer's Guide.

Async Professional includes Delphi components, demonstration programs, example programs, and a help system. It also includes a few general files, which are described below.

README.HLP

A help file that describes changes to the manual and new features added after the manual was printed. Please read this file before using the product.

APRO.XXX

A text file that summarizes changes between successive versions of Async Professional. "XXX" is replaced with the version number. For example, APRO.4.01 summarizes changes between versions 4.00 and 4.01 of Async Professional.

APRO.HLP or APROBCB.HLP

A Windows help file containing information about Async Professional. The help system is generated from this manual and contains the complete text of the reference sections along with abbreviated versions of each component introduction.

Units supplied

The Async Professional components depend on several low-level units that are not documented in this manual and should never need to be used directly by your program. These include AdFaxCtl, AdMeter, AdPackEd, AdPropEd, AdRasUtl, AdSelCom, AdTSel, AdTUtl, and AdWUtl.

The AdXDial, AdXDown, AdXPort, AdXProt, and AdXUp bonus units provide example dialogs for dialing, downloading, selecting port options, selecting protocol options, and uploading, respectively. Although these units are not documented in this manual, you can easily use the forms in your program or modify them for your needs. Units that will be used in your applications to access the Async Professional VCL components are shown in Table 1.1.

Table 1.1: *Async Professional units*

Unit	Description
AdAbout	Includes the Version property.
AdExcept	Defines all of the exception classes used by Async Professional.

Table 1.1: *Async Professional units (continued)*

AdFax	Includes the TApdAbstractFax, TApdSendFax, TApdReceiveFax, TApdFaxLog, and TApdAbstractFaxStatus components.
AdFaxCvt	Includes the TApdFaxConverter and TApdFaxUnpacker components.
AdFaxPrn	Includes the TApdFaxPrinter, TApdFaxPrinterLog, and TApdAbstractFaxPrinterStatus components.
AdFaxSrv	Includes the TApdFaxJobHandler, TApdFaxServer, TApdFaxServerManager and TApdFaxClient components.
AdFPStat	Includes the TApdFaxPrinterStatus component.
AdFStat	Includes the TApdFaxStatus component.
AdFtp	Includes the TApdFtpClient and TApdFtpServer components.
AdFView	Includes the TApdFaxViewer component.
AdLibMdm	Includes the TApdLibModem modemcap interface component.
AdModem	Includes the TAdModem and TAdAbstractModemStatus components.
AdPacket	Includes the TApdDataPacket component.
AdPager	Includes the TApdTAPPager and TApdSNPPPager components.
AdPort	Includes the TApdComPort component.
AdProtcl	Includes the TApdProtocol, TApdProtocolLog, and TApdAbstractStatus components.
AdPStat	Includes the TApdProtocolStatus component.
AdRas	Includes the TApdRasDialer component.
AdRStat	Includes the TApdRasStatus component.
AdSapiEn	Includes the TApdSapiEngine and TApdSapiPhone components.
AdScript	Includes the TApdScript component.
AdSocket	Includes the TApdSocket component.
AdState	Includes the TApdStateMachne, TApdState and TApdStateWatcher components.
AdStatLt	Includes the TApdStatusLight and TApdSLController components.
AdTapi	Includes the TApdTapi, TApdTapiLog, and TApdAbstractTapiStatus components.
AdTrmBuf	Includes the TAdTerminalBuffer component.
AdTrmEmu	Includes the TAdTerminalEmulator, TAdTTYEmulator, TAdVT100Emulator and the TAdTerminal components.
AdTrmMap	Includes the TAdKeyboardMapping and TAdCharSetMapping components.
AdTrmPsr	Includes the TAdTerminalParser and TAdVT100Parser components.

Table 1.1: *Async Professional units (continued)*

AdTStat	Includes the TApdTapiStatus component.
AdVoIP	Includes the TApdVoIP component and associated classes.
AdWnPort	Includes the TApdWinsockPort component.

AproReg is the unit used to register all of the Async Professional components and to add them to the component palette. Refer to the installation section of the Developer's Guide for more information.

Demonstration and example programs

Async Professional includes many demonstration programs and small example programs. The demonstration programs are intended to be useful as well as instructive and they include many user-interface niceties that sometimes obscure the use of the communications objects, while the example programs typically use minimal code allowing you to focus on the specific component or technique being demonstrated. Generally, you should use the example programs to understand the basics of the communications components and use the demonstration programs for real-life examples of implementation.

Table 1.2 briefly describes the demonstration programs. These programs are described fully in the Developer's Guide.

Table 1.2: *Async Professional demonstration programs*

Name	Description
TermDemo	Demonstrates the terminal component and related components. It provides more features than the introductory example programs but isn't as complex as TCom3.
SendFax	Shows how to send multiple faxes with optional cover pages.
RcvFax	Shows how to wait for and answer incoming fax calls.
Cvt2Fax	Demonstrates how the fax converter component converts text, BMP, PCX, DCX, TIFF files, and the new shell and COM methods.
ViewFax	A fax viewer that allows you to view APF files.
FaxMon	Monitors the Async Professional fax printer driver for print jobs and notifies FaxSrvr. Designed to run with FaxSrvr.
FaxSrvr	Retrieves the recipient's fax number from FaxMon, then sends the fax. Designed to run with FaxMon.
FaxSrvX	Monitors print jobs sent to the Async Professional fax printer driver and send the faxes. FaxSrvX is based on SendFax.

Table 1.2: *Async Professional demonstration programs (continued)*

VoIPDemo	A simple Voice of IP (IP Telephony) program that demonstrates how to establish VoIP connections and conduct conversations.
TTSDemo	A simple SAPI Text-to-Speech demonstration program.
SRDemo	A simple SAPI Speech Recognition demonstration program.
StatDemo	A project demonstrating the protocol wizard components to create an automated logon.
ModemCap	A somewhat detailed example demonstrating the modemcap modem database.
InfParsr	A demonstration program that parses modem INF files and converts them to modemcap entries.
RasDemo	A simple RAS dialer program that can dial and manipulate RAS phonebooks. It is based on the TApdRasDialer component.
FTPDemo	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.
ExPaging	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 example programs are discussed fully in the “Example” section for each of the components throughout the manual. See EXAMPLES.TXT in the installation directory for a list of the example programs included in Async Professional. The list provides the name of the project (DPR) file for each example. The main form/unit file name typically consists of the project name followed by ‘0’.

The Component Hierarchy

In order to provide the user easy access to a product version number, a Version property is associated with the Async Professional's non-visual TApdBaseXxx components. The VCL ancestor is listed in the hierarchy for each TApdBaseXxx component.

Version	read-only property
---------	--------------------

```
property Version : string
```

↩ Show the current version of Async Professional.

Version is provided so you can identify your Async Professional version if you need technical support. You can display the Async Professional about box by double-clicking this property or selecting the dialog button to the right of the property value.

On the following pages are diagrams showing the Async Professional component hierarchy. All of the diagrams show the component or class name and the unit name where the component is implemented. All of the Async Professional components derive ultimately from the TComponent class.

Some of the classes, such as TApdCustomComPort, include the word "Custom" in their names. These classes follow the Borland convention of implementing all of the properties and events of a component, but publishing none of them. You don't design with these classes, but they are useful for deriving your own components that differ in some way from the supplied components. They are not described in the following overview.

ComPort, Winsock, FTP, Data Packet, Socket

TComponent (VCL)

 TApdBaseComponent (OOMisc)

 TApdCustomComPort (AdPort)

 TApdComPort (AdPort)

 TApdCustomWinsockPort (AdWnPort)

 TApdWinsock Port (AdWnPort)

 TApdCustomFtpClient (AdFtp)

 TApdFtpClient (AdFtp)

 TApdFtpLog (AdFtp)

 TApdDataPacket (AdPacket)

 TApdSocket (AdSocket)

The TApdComPort component is the foundation of Async Professional. It allows you to access your PC's serial ports, to set their properties, and to do low level serial communications. Almost all of the other components in Async Professional contain a link to a TApdComPort. The TApdWinsockPort provides a Winsock port for use in network and Internet communications. It provides all of the properties and methods of a TApdComPort, so can perform either as a Winsock port or a standard communications port. The TApdFtpClient is a specialized TApdWinsockPort that implements client-side file transfer protocol (FTP) capabilities. TApdFtpLog can be associated with a TApdFtpClient to provide automatic FTP logging services. TApdDataPacket provides data packets for incoming data. TApdSocket is a low-level component that provides standard Winsock services.

Scripting component

TComponent (VCL)

 TApdBaseComponent (OOMisc)

 TApdCustomScript (AdScript)

 TApdScript (AdScript)

This diagram shows the hierarchy of the TApdScript component which contains properties and methods for automating basic communications sessions.

RAS dialing

TComponent (VCL)

 TApdBaseComponent (OOMisc)

 TApdCustomRasDialer (AdRas)

 TApdRasDialer (AdRas)

 TApdCustomRasStatus (AdRas)

 TApdRasStatus (AdRStat)

This diagram shows the components used to establish and monitor a connection to another computer via Dialup Networking. TApdRasDialer provides an easy to use interface to the Microsoft Remote Access Services API and a set of standard functions for manipulating phonebook entries and displaying dial status information. The TApdRasStatus component provides a standard RAS dialing status dialog for use on machines whose RAS DLL does not provide a status display.

TAPI modem management

TComponent (VCL)

 TApdBaseComponent (OOMisc)

 TApdCustomTapiDevice (AdTapi)

 TApdTapiDevice (AdTapi)

 TApdAbstractTapiStatus (AdTapi)

 TApdTapiStatus (AdTapi)

 TApdTapiLog (AdTapi)

This diagram shows the family of components used for Telephony Application Programming Interface (TAPI) modem management. TApdTapiDevice provides modem dialing, answering, and configuration services using Windows TAPI support. The TApdTapiDevice also provides support for advanced voice modem features like WAV file playing and recording, and DTMF tone detection and generation. TApdAbstractTapiStatus is an abstract class that can be attached to a TApdTapiDevice object to display the TAPI status. TApdTapiStatus is an implementation of this abstract class that displays status in a particular format. TApdTapiLog is a small component that can be attached to a TApdTapiDevice object to keep a log file of TAPI actions.

Modem operations

TComponent (VCL)

 TApdBaseComponent (OOMisc)

 TApdCustomSModem (AdSModem)

 TApdSModem (AdSModem)

This diagram shows the ancestry of the TApdSModem (simple modem) component. It provides a simple interface for accessing the most commonly used modem operations. It integrates the selection of the modem from the modem database and the dialog to show the current status of the modem.

Terminal

TObject (VCL)

 TAdTerminalBuffer (ADTrmBuf)

 TAdKeyboardMapping (ADTrmMap)

 TAdCharSetMapping (ADTrmMap)

 TAdTerminalParser (ADTrmPsr)

 TAdVT100Parser (ADTrmPsr)

 TComponent (VCL)

 TApdBaseComponent (OOMisc)

 TAdTerminalEmulator (ADTrmEmu)

 TAdTTYEmulator (ADTrmEmu)

 TAdVT100Emulator (ADTrmEmu)

 TWinControl (VCL)

 TApdBaseWinControl (OOMisc)

 TAdTerminal (ADTrmEmu)

This diagram shows the family of components and classes used for terminals and emulators. The TAdTerminalBuffer class defines a data structure for maintaining the data required for a communications terminal display. The TAdTerminalParser class is the ancestor class that defines the functionality of a terminal parser. The TAdVT100Parser class defines a parser that understands VT100 terminal data streams.

The TAdKeyboardMapping class provides a simple, convenient method to specify the PC keystrokes that map onto the emulated terminal keystrokes, and also what control sequence those terminal keystrokes are going to send to the host computer. The TAdCharSetMapping class provides a method to emulate the different character sets used by terminals by using glyphs from different fonts.

The TAdTerminalEmulator class is the base class for all terminal emulators. The TAdTTYEmulator class emulates a teletype terminal. The TAdVT100Emulator class emulates a Digital Equipment Corporation (DEC) VT100 terminal and supports some common extensions to the normal VT100 escape sequence set.

The TAdTerminal component represents the visual part of a terminal. It is the only visual component in the Async Professional suite of terminal classes. It is responsible for maintaining the window handle and for performing the low-level processing to get all the possible keystrokes available on a PC keyboard.

Modem status lights

TComponent (VCL)

 TApdBaseComponent (OOMisc)

 TApdCustomSLController (AdStatLt)

 TApdSLController (AdStatLt)

 TControl (VCL)

 TGraphicControl (VCL)

 TApdCustomStatusLight (AdStatLt)

 TApdStatusLight (AdStatLt)

This diagram shows the family of components used for modem status lights.

TApdStatusLight is a graphical component used to display the light in a “lit” or “unlit” state. TApdSLController manages a group of TApdStatusLight components and lights them based on events it receives from an associated comport component.

Paging

TComponent (VCL)

 TApdBaseComponent (OOMisc)

 TApdAbstractPager (AdPager)

 TApdCustomModemPager (AdPager)

 TApdTAPPager (AdPager)

 TApdCustomINetPager (AdPager)

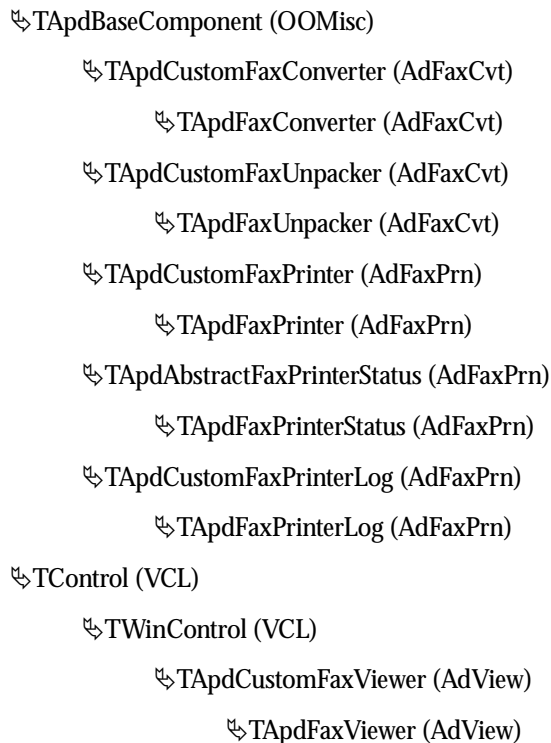
 TApdSNPPPager (AdPager)

This diagram shows the family of components used for sending alphanumeric pages.

TApdAbstractPager provides properties and methods that are common to paging regardless of the transmission medium. TApdTAPPager sends alphanumeric pages to paging services that support Telelocator Alphanumeric Protocol and the Motorola Personal Entry Terminal Protocol. TApdSNPPPager implements Internet based paging using the Simple Network Paging Protocol.

Faxes: conversion, unpacking, viewing, printing

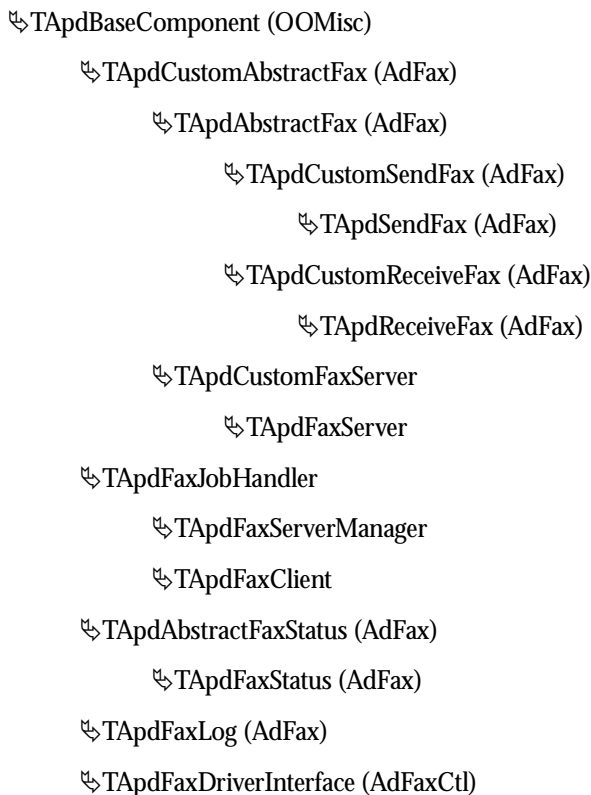
TComponent (VCL)



This diagram shows the family of components used for fax conversion, unpacking, viewing, and printing. TApdFaxConverter converts ASCII text, BMP, PCX, DCX, and TIFF files to Async Professional Fax (APF) format. TApdFaxUnpacker unpacks fax files to memory bitmaps or image files. TApdFaxViewer allows viewing of faxes with scaling, rotation, and white space compression. TApdFaxPrinter prints faxes with scaling and headers and footers. TApdAbstractFaxPrinterStatus is an abstract class that can be attached to a TApdFaxPrinter object to display print status. TApdFaxPrinterStatus is an implementation of this abstract class that displays status in a particular format. TApdFaxPrinterLog is a small component that can be attached to a TApdPrinter object to keep a log file of printer actions.

Faxes: sending, receiving

TComponent (VCL)



This diagram shows the family of components used for fax sending and receiving. TApdAbstractFax provides the set of properties, methods, and events that are common to both sending and receiving. The send and receive components are derived from TApdAbstractFax. TApdSendFax sends single or multiple faxes with optional cover pages. TApdReceiveFax receives faxes. TApdFaxServer is the faxing engine for the Fax Server Components. It 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 TApdSendFax and TApdReceiveFax.

TApdFaxJobHandler provides methods to manipulate the Async Professional Job file format. TApdFaxServerManager component provides fax scheduling and queuing capability. TApdFaxClient provides the ability to create APJ fax job files with a design-time interface.

TApdAbstractFaxStatus is an abstract class that can be attached to a TApdSendFax, TApdReceiveFax or TApdFaxServer object to display fax status. TApdFaxStatus is an implementation of this abstract class that displays status in a particular format. TApdFaxLog is a small component that can be attached to a TApdSendFax, TApdReceiveFax or TApdFaxServer object to keep a log file of fax actions. TApdFaxDriverInterface provides control for the fax printer drivers.

Organization of this Manual

This manual is organized as follows:

- Chapter 1 is an introduction to Async Professional.
- Chapters 2 through 16 describe the Async Professional components.
- Chapter 17 describes a few general non-communication components.
- The appendices provide a discussion of error handling, a description of the Async Professional conditional defines, and a glossary of communications terms.
- An identifier index and a conventional subject index are provided.

Each chapter starts with an overview of the classes and components discussed in that chapter. The overview also includes a hierarchy for those classes and components. Each class and component is then documented individually, in the following format:

Overview

A description of the class or component.

Hierarchy

Shows the ancestors of the class being described, generally stopping at a VCL class. The hierarchy also lists the unit in which each class is declared and the number of the first page of the documentation of each ancestor. Some classes in the hierarchy are identified with a number in a bullet: ❶. This indicates that some of the properties, methods, or events listed for the class being described are inherited from this ancestor and documented in the ancestor class.

Properties, methods, and events lists

The properties, methods, and events for the class or component are listed. Some of these may be identified with a number in a bullet: ❶. In these cases, they are documented in the ancestor class from which they are inherited.

Reference section

Details the properties, methods, and events of the class or component. These descriptions are in alphabetical order. They have the following format:

- Declaration of the property, method, or event.
- Default value for properties, if appropriate.
- A short, one-sentence purpose. A ⚡ symbol is used to mark the purpose to make it easy to skim through these descriptions.
- Description of the property, method, or event. Parameters are also described here.
- Examples are provided in many cases.
- The “See also” section lists other properties, methods, or events that are pertinent to this item.

Throughout the manual, the ⚡ symbol is used to mark a warning or caution. Please pay special attention to these items.

Naming conventions

To avoid class name conflicts with components and classes included with the compiler or from other third party suppliers, all Abbrevia class names begin with “TAb.” The “Ab” stands for Abbrevia.

“Custom” in a component name means that the component is a basis for descendant components. Components with “Custom” as part of the class name do not publish any properties. Instead, descendants publish the properties that are applicable to the derived component. If you create descendant components, use these custom classes instead of descending from the component class itself.

On-line help

Although this manual provides a complete discussion of Abbrevia, keep in mind that there is an alternative source of information available. Once properly installed, help is available from within the IDE. Pressing <F1> with the caret or focus on an Abbrevia property, routine or component displays the help for that item.

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 learn from others' experiences and share ideas in addition to getting answers to questions.

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

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

The TurboPower KnowledgeBase is another excellent support option. It has hundreds of articles about TurboPower products accessible through an easy to use search engine (www.turbopower.com/search). The KnowledgeBase is open 24 hours a day, 7 days a week. 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: Port Component

The Async Professional port component builds the foundation for all communications applications. The simplest application might contain a single `TApdComPort`; a more complex application might contain one or more port components and many other components that rely on the services of a `TApdComPort` (terminal windows, modems, and so on).

This chapter describes the `TApdComPort` component, which contains properties and methods for the following:

- Configuring the serial port hardware and Windows communications driver (buffer sizes, line parameters, flow control).
- Providing information about the state of the serial port (modem signals, line errors).
- Transmitting and receiving data.
- Interfacing with the `TApdDataPacket` component (see page 132) to identify and handle received data.
- Assigning VCL events to handle received data, matched strings, status changes, and timers.

In addition to its support for standard serial ports, the `TApdComPort` includes specialized support for the RS-485 communication standard (see page 31).

If you need to support network or Internet communications using Winsock, you should consider using the `TApdWinsockPort` (see page 106) instead of the `TApdComPort`. Since the `TApdWinsockPort` is a descendant of the `TApdComPort`, it retains all capabilities of the `TApdComPort`, and adds support for Winsock.

TApdComPort Component

An application uses the TApdComPort component to control serial port hardware. All serial port I/O is performed by calling methods of TApdComPort and by writing event handlers that respond to serial events. Higher level communications actions such as dialing a modem or transferring a file use a TApdComPort to interact with the hardware.

Sending and receiving data through the serial port is obviously part of the process, but most communications applications also need to identify and handle data according to a specific need. Async Professional provides many high level components that simplify common tasks such as displaying the data (see “Chapter 11: Status Light Components” on page 373) and transferring data using an error-correcting protocol (see “Chapter 12: File Transfer Protocols” on page 383).

In addition to such common and well-defined tasks, a flexible component is provided that allows you to define the type of data you are looking for, automatically collect the data for you, and notify you when the data has arrived (see “Chapter 3: Data Packet Component” on page 105). Be sure to investigate whether or not these high level components could meet your needs before venturing too far into the low level facilities of the TApdComPort discussed in the following section—you could save yourself a lot of time and effort.

Triggers and trigger handlers

Async Professional uses the term “trigger” for any serial port action that can cause its communications dispatcher to generate a VCL event. There are four types of triggers:

- Data available—received data is available.
- Data match trigger—a particular character or character string was received.
- Status trigger—a status event occurred (details later in this section).
- Timer trigger—a timer expired.

Note: In 32-bit applications, TApdComPort events are synchronized to the thread that sets Open to True (in other words, the thread that instantiates the dispatcher). In most cases, this will end up being the main VCL thread.

The TApdComPort component contains a variety of routines for managing triggers. Triggers can be added, activated, modified, and deactivated.

For example, adding a timer trigger looks something like this:

```
var
    TrigHandle : Word;
...
TrigHandle := ApdComPort.AddTimerTrigger;
ApdComPort.SetTimerTrigger(TrigHandle, 36, True);
```

This code adds a timer trigger and stores the trigger handle in TrigHandle. The timer is activated and set to expire in 36 ticks (about 2 seconds). When the timer expires, an OnTriggerTimer event will be generated. An event handler must be assigned to the OnTriggerTimer property to gain control when the timer expires.

```
procedure Form1.TriggerTimer(CP : TObject; TriggerHandle : Word);
begin
    ...
end;
...
ApdComPort.OnTriggerTimer := TriggerTimer;
```

Here are the TApdComPort event properties and event handler declarations that correspond to the four Async Professional trigger types, and the data they contain.

OnTriggerAvail

```
procedure(CP : TObject; Count : Word) of object;
```

Generated when a certain number of bytes of received serial data are available for processing. Count is the actual number of bytes that have been received at the instant the trigger is generated.

It's likely that more than one byte of data will be available when the message handler is called. The amount of data received in a given event depends on several factors, such as the nature of the data itself (does it get sent one character at a time by the transmitter, or is it being sent in blocks or streams?), and the hardware/driver supplying the data to Async Professional (standard Windows communications drivers typically supply data in small blocks (8 bytes or less), but Winsock can supply data in large blocks (1,000 bytes or more)).

☛ **Caution:** Be sure to process the exact number of bytes passed in the Count parameter of this handler. If you process fewer bytes, you risk losing characters to another component extracting data during the event (such as the terminal). If you process a number of bytes greater than the value of the Count property, you risk receiving events for overlapping data—which may eventually lead to an EBufferIsEmpty exception.

Here's what a typical OnTriggerAvail event handler should look like:

```
procedure Form1.TriggerAvail(CP : TObject; Count : Word);
var
  I : Word;
  C : Char;
begin
  for I := 1 to Count do begin
    C := ApdComPort.GetChar;
    ...
  end;
end;
```

The OnTriggerAvail and OnTriggerData events are generated from the dispatcher thread, and synchronized with the appropriate thread through the use of SendMessageTimeout. The timeout value is 3 seconds. If your OnTriggerAvail or OnTriggerData event handlers take longer than 3 seconds to execute (starting at the time the SendMessageTimeout method was called), you run the risk of losing data. If the timeout period expires, the dispatcher will assume that a deadlock has occurred and will continue processing the serial port notification messages. If you need to perform lengthy processing, you should collect the received data in a buffer, then process the buffer outside the context of the OnTriggerAvail event handler.

OnTriggerData

```
procedure(CP : TObject; TriggerHandle : Word) of object;
```

Generated when the dispatcher finds a match in the received data for a data string previously specified using the AddDataTrigger method. TriggerHandle is the trigger handle returned by AddDataTrigger.

Usually the dispatcher finds the match in the middle of a block of bytes it is examining. In this case the dispatcher first generates an OnTriggerAvail event for all the data leading up to the matched string, then it generates the OnTriggerData event to advertise the match, and finally it generates another OnTriggerAvail event for the data associated with the match.

The data trigger does not guarantee that the notification will be exactly synchronized with the actual occurrence of the data match. In other words, if you set a data trigger for the string "MYDATA", it is not reliable nor was it intended for you to start capturing those characters in an OnTriggerAvail event immediately following the OnTriggerData event for that string. Refer to the APROFAQ.HLP file for example code.

- ☛ **Caution:** Be sure to process the exact number of bytes passed in the Count parameter of this handler. If you process fewer bytes, you risk losing characters to another component extracting data during the event (such as the terminal). If you process a number of bytes greater than the value of the Count property, you risk receiving events for overlapping data—which may eventually lead to an EBufferIsEmpty exception.

In most cases, the `TApdDataPacket` component is the best way to capture a specific string. See “Chapter 3: Data Packet Component” on page 105 for more information on how to use the `TApdDataPacket` component.

OnTriggerTimer

```
procedure(CP : TObject; TriggerHandle : Word) of object;
```

Generated when the dispatcher determines that a timer has expired. `TriggerHandle` is the trigger handle returned when `AddTimerTrigger` was called.

OnTriggerStatus

```
procedure(CP : TObject; TriggerHandle : Word) of object;
```

Generated when a status change occurs. Status types include: changes in modem signals (CTS, DSR, RING, and DCD), changes in line status (line break received and data overrun, parity, and framing errors), output buffer free space reaching a specified level, and output buffer used space reaching a certain level.

`TriggerHandle` is the trigger handle returned when `AddStatusTrigger` was called.

When a status trigger is used to track more than one modem signal, the event handler must check the appropriate modem status properties to determine exactly which modem signal generated the trigger. You can avoid these calls by adding a separate trigger for each modem signal that you need to trap.

Specialized versions of the `OnTriggerStatus` event are also available through several events that are generated only when one particular kind of status change occurs. Event handlers for these events are called after the `OnTriggerStatus` event handler. It is possible to have an `OnTriggerStatus` handler and also one or more specific status event handlers. Here are the event properties and event handler declarations for these specialized events.

OnTriggerLineError

```
procedure(
  CP : TObject; Error : Word; LineBreak : Boolean) of object;
```

Generated when the dispatcher determines that a line error or break signal occurred while receiving data. `Error` contains one of the following values (a subset of all possible values of the `LineError` property):

```
leOverrun = 2;
leParity  = 3;
leFraming = 4;
```

`LineBreak` is `True` if a break signal was received. Note that breaks are often accompanied by framing errors.

OnTriggerModemStatus

```
procedure(CP : TObject) of object;
```

Generated when a monitored modem status signal changes. The signals to monitor are passed in `SetStatusTrigger`. Note that there is no `TriggerHandle` parameter to this event handler. As such, it cannot differentiate among multiple modem status triggers. If you need to add more than one modem status trigger, use an `OnTriggerStatus` event handler instead of this specialized one.

OnTriggerOutbuffFree

```
procedure(CP : TObject) of object;
```

Generated when the number of bytes free in APro's output buffer is greater than or equal to the number passed to `SetStatusTrigger` for this event.

OnTriggerOutbuffUsed

```
procedure(CP : TObject) of object;
```

Generated when the number of bytes used in APro's output buffer is less than or equal to the number passed to `SetStatusTrigger` for this event.

OnTriggerOutSent

```
procedure(CP : TObject) of object;
```

Generated after `PutChar`, `PutBlock`, or `PutString` is called.

The `TApdComPort` component has one more event type that is a superset of all of those just described. This event is generated for every trigger reported by the Async Professional dispatcher to the component.

OnTrigger

```
procedure(
  CP : TObject; Msg, TriggerHandle, Data : Word) of object;
```

Generated for all trigger events. `Msg` is the internal message number that corresponds to the event. For example, the message `APW_TRIGGERAVAIL` corresponds to the `OnTriggerAvail` event (the other message names are documented in the reference section for the `OnTrigger` event on page 69). `TriggerHandle` is the trigger handle returned when the trigger was added. `Data` is the data associated with this trigger.

This event is generated prior to the associated specific event type. For example, when a modem status signal generates a trigger, the `OnTrigger` event is generated with `Msg` set to `APW_TRIGGERSTATUS`, then the `OnTriggerStatus` event is generated with `TriggerHandle` set to the status trigger handle, then the `OnTriggerModemStatus` event is generated.

In most cases you would not want to use an OnTrigger event handler, but instead provide individual event handlers. OnTrigger is provided primarily for script or protocol-oriented processes where it is more efficient to manage all tasks from one central routine.

With the exceptions of OnTriggerAvail, OnTriggerData, and OnTriggerOutSent, all triggers must be reactivated within the event handler. That is, the triggers generate a single message and do not restart themselves automatically. The following example uses triggers:

```
{IFDEF Win32}
{$APPTYPE CONSOLE}
{$ENDIF}
unit Extrig0;

interface

uses
  {$IFDEF Win32}
  WinCrt,
  {$ENDIF}
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, AdPort, AdTerm;
type
  TExTrigTest = class(TForm)
    ApdComPort1: TApdComPort;
    StartTest: TButton;
    Label1: TLabel;
    procedure ApdComPort1TriggerAvail(
      CP : TObject; Count : Word);
    procedure ApdComPort1TriggerData(
      CP : TObject; TriggerHandle : Word);
    procedure ApdComPort1TriggerTimer(
      CP : TObject; TriggerHandle : Word);
    procedure StartTestClick(Sender : TObject);
  private
    TimerHandle : Word;
    DataHandle : Word;
  end;
var
  ExTrigTest: TExTrigTest;
implementation
{$R *.DFM}
```

```

procedure WriteIt(C : Char);
begin
    if Ord(C) > 32 then
        Write(C)
    else
        Write('[', Ord(C), ']');
    end;
end;

procedure TExTrigTest.ApdComPort1TriggerAvail(
    CP : TObject; Count : Word);
var
    I : Word;
    C : Char;
begin
    WriteLn('OnTriggerAvail event: ', Count, ' bytes received');
    for I := 1 to Count do begin
        C := ApdComPort1.GetChar;
        WriteIt(C);
    end;
    WriteLn;
    WriteLn('-----');
end;

procedure TExTrigTest.ApdComPort1TriggerData(
    CP : TObject; TriggerHandle : Word);
var
    I : Word;
    C : Char;
begin
    WriteLn('OnTriggerData event: ', TriggerHandle);
end;

procedure TExTrigTest.ApdComPort1TriggerTimer(
    CP : TObject; TriggerHandle : Word);
begin
    WriteLn('OnTriggerTimer event: ', TriggerHandle);
end;

procedure TExTrigTest.StartTestClick(Sender : TObject);
begin
    TimerHandle := ApdComPort1.AddTimerTrigger;
    ApdComPort1.SetTimerTrigger(TimerHandle, 91, True);
    DataHandle := ApdComPort1.AddDataTrigger('OK', True);
    {send a string to a modem that will hit all triggers}
    ApdComPort1.PutString('ATI'^M);
end;

end.

```

This is the unit containing the form for the example project EXTRIG. It contains two components: StartTest (a TButton) and ApdComPort1 (a TApdComPort). StartButton implements a Click event handler named StartButtonClick that starts the trigger demonstration. StartButtonClick adds and sets a timer for 91 ticks (5 seconds) and adds a data trigger for the string "OK." Finally, it transmits the string 'ATT'^M. If a modem is attached to the serial port, this tells the modem to return its version information, followed by the response "OK."

The form implements three event handlers:

- ApdComPort1TriggerAvail for OnTriggerAvail events.
- ApdComPort1TriggerData for OnTriggerData events.
- ApdComPort1TriggerTimer for OnTriggerTimer events.

As the modem returns its version information the program receives one or more OnTriggerAvail events. The handler for this event, ApdComPort1TriggerAvail, retrieves and displays this data to the WinCrt window (or console window in a 32-bit console application). ApdComPort1TriggerData is called when "OK" is received; it writes a simple message to the WinCrt window. After 5 seconds ApdComPort1TriggerTimer is called, which also writes a short message to the WinCrt window.

ISDN support overview

Integrated Service Digital Network (ISDN) connections are becoming more commonplace in today's communications applications as the associated costs steadily decline.

ISDN introduces a number of features to the world of PC communications. ISDN lines have digital channels as opposed to the analog lines (also called POTS for Plain Old Telephone Service) that standard AT-compatible modems use. Digital channels provide much higher bandwidths that allow faster communication speeds and higher data throughput. ISDN lines consist of multiple channels, whereas POTS consists of a single line. With multiple channels, various types of information (e.g., data, voice, and video) can be transmitted simultaneously.

ISDN is available in a number of configurations. Basic Rate Interface ISDN (BRI-ISDN) comes with two B-channels and a D-channel. B-channels allow for 64 Kbps or 56 Kbps throughput depending on what your local carrier provides. In some configurations, these channels can be utilized as a single 128 Kbps or 112 Kbps line. The D-channel provides a 16 Kbps channel that is normally used to send signaling information and additional control data (e.g., the calling party's name, location, and configuration). Primary Rate Interface ISDN (PRI-ISDN) is much more expensive and provides up to 23 B-channels and one D-channel in the U.S., Canada, and Japan. It provides up to 30 B-channels and two D-channels in Europe.

Async Professional supports basic ISDN services with many AT-compatible adapters through the `TApdTapiDevice` component. In general, non-TAPI and non-AT-compatible ISDN adapters (e.g., those used strictly for PPP or dial-up connections) are not supported by Async Professional for serial communications. The `TApdTapiDevice` treats ISDN adapters as standard AT-compatible modems that control a single channel. A `TApdTapiDevice` cannot control multiple channels. You can use the other channels in your ISDN connection by creating a `TApdTapiDevice` for each one.

Note that Async Professional support ISDN devices through the `TApdTapiDevice` interface. Async Professional does not support the CAPI interface to ISDN devices.

Async Professional supports two types of ISDN adapter/driver configurations: ISDN adapters that are AT-compatible and ISDN adapters that come with AT-compatible analog drivers. Many AT-compatible ISDN adapters work with Async Professional transparently, usually with higher throughput than standard faxmodems. These modems and drivers are used for ISDN to ISDN connections. ISDN adapters that provide analog modem or fax drivers work with Async Professional just like a standard faxmodem (i.e., at the lower speed and throughput of a standard faxmodem). When the driver is the selected device, the ISDN adapter and line can connect to standard POTS devices such as a faxmodem or fax machine. To date, there are relatively few ISDN adapters that provide analog fax drivers.

Async Professional can be used with ISDN adapters that conform to the following specifications:

- 100% AT (command set) compatible.
- TAPI compatible and has a TAPI driver (available in the Modems Applet in the Microsoft Windows Control Panel).
- At least one data channel.

Async Professional also supports the following optional features:

- Analog modem (TAPI) driver support (used when connecting to standard modems on POTS lines).
- Faxmodem (TAPI) driver support (used when connecting to standard faxmodems or fax machines on POTS lines).

Some TAPI service provider drivers do not support the full set of TAPI functions. You might be limited by the driver provided by your ISDN adapter manufacturer.

Using a `TApdTapiDevice` component with an ISDN line is exactly the same as using it with a standard analog line. See “Chapter 8: TAPI Components” on page 203 for more information on how to use a `TApdTapiDevice`.

RS-485 support overview

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

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

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

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

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

- RTS Control
- Automatic
- Other

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


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

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

RTS control

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

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

 **Caution:** The `RTS` line control follows the output buffer. If the output buffer empties while your code is formatting and transmitting a command, the `RTS` line could be lowered and re-raised—which might cause some RS-485 devices to misinterpret the message.

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

For example, use the first code example rather than the second:

```
Message := '!' + Address + MsgLength + Message + '$';
PutString(Message);

PutChar('!');
PutChar(Address);
PutChar(MsgLength);
PutString(Message, MsgLength);
PutChar('$');
```

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

The automatic handling of the RTS line is possible for standard ports in all environments and for all non-standard serial ports that provide the necessary driver support. Part of this support (in Windows 95/98/ME) includes the detection of the serial port hardware's base address. If this address cannot be detected, attempts to transmit in RS-485 mode will raise the `EBaseAddressNotSet` exception. If you know the base address of the port, you can set it manually using the `BaseAddress` run-time property.

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

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

Debugging facilities

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

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

Tracing

Variations of this need for an audit trail can occur in almost any communications program. Rather than force you to add such debugging code to your applications, Async Professional provides a general auditing facility called *tracing*.

Simply put, tracing gives you the ability to keep track of all characters transmitted and received by a program, in the form of a text file that your program creates on request.

Every time an application successfully retrieves a received character (i.e., GetChar returns a character) a trace entry is created. Every time an application successfully sends a character to the Windows communications driver (i.e., PutChar is called successfully) another trace entry is created.

These entries are stored in a circular queue of a specified size. Since the queue is circular, it always contains information about the most recent transmitted or received characters. In Delphi 1, the queue can hold at most 32760 entries. In 32-bit compilers, the queue can hold up to 4 million entries.

The queue can be dumped to a text file at any time. This text file is a report of all data transmitted and received by the application in the following format:

```
Transmit:
**[24]B0100000027fed4[13][138][17]

Receive:
rz[13]**[24]B00800000000dd38[13][138][17]

Transmit:
**[24]B0100000027fed4[13][138][17]

Receive:
[17]*[24]C[4][1][0][0][0][184]6[30][139]a.txt[0]6048
4734111064 0 0 3 18144[0][24]kP[251]B6
```

This report happens to be the first few exchanges of a Zmodem protocol transfer. Printable characters are displayed as ASCII strings; non-printable characters are displayed as decimal values in square brackets (e.g., [13] is a carriage return) or optionally as hex values.

Notice that the data is grouped in blocks of received and transmitted characters representing the sequence in which the program made calls to GetChar and PutChar. A new block is created whenever the program switches from transmitting to receiving or vice versa. For example, if a program transmitted a continuous stream of data without ever stopping to receive data, the report would consist of a single transmit block. Conversely, if a program contained a character-by-character terminal loop, each block might consist of only one character because the program constantly switched between sending and receiving single characters.

The sequence in a trace report is not necessarily the same as the sequence in which data arrived or departed from the serial port. Suppose that a program transmits the string "ABCD" to a remote device that echoes the characters it receives. In a chronological report, the echo characters received by the transmitter would be intermingled with, but slightly behind, the transmitted characters.

That's not what tracing was designed to do. It was designed to show the data in the order it was processed by an application. The major benefit to such ordering is that it permits checking program logic against the data the program is actually processing. For example, tracing was extremely valuable during the development of the modem and file transfer protocol portions of Async Professional. If a particular test went awry, it was a simple matter to review the trace and find out what went wrong (of course, finding out why it went wrong was another matter).

The mechanics of using tracing are quite simple. A typical use would look something like the following:

```
...create comport component
ApdComPort.TraceSize := 1000;
ApdComPort.Tracing := tlOn;
...use comport component
ApdComPort.TraceName := 'TEST.TRC';
ApdComPort.Tracing := tlDump;
...destroy comport component
```

The state of the tracing facility is controlled by setting the Tracing property to one of the following values shown in Table 2.1.

Table 2.1: *Tracing property values*

Value	Effect
tlOff	Turns off tracing without saving the trace data.
tlOn	Turns tracing on or resumes tracing after a pause.
tlDump	Writes the trace data to a new file, turns off tracing.
tlAppend	Appends the trace to an existing file, turns off tracing.
tlClear	Clears the trace buffer but leaves tracing on.
tlPause	Pauses tracing.

When inspected, either at run time or design time, Tracing will always be one of tlOff, tlOn, or tlPause. Setting Tracing to any other value causes an action (such as writing the trace file) then sets Tracing to either tlOff or tlOn. For example, setting Tracing to tlDump writes the trace data to a new file, then turns Tracing off, so inspecting Tracing immediately after setting it to tlDump would return tlOff.

After you create a comport component, set the `TraceSize` property to specify the trace buffer size, then set the `Tracing` property to `tlOn` to begin tracing. From then on, every successful call to `PutXxx` and `GetXxx` is automatically recorded. If more than the specified number of trace entries is generated, the queue always contains the most recent. Set the `TraceName` property to the name of the file where the trace file should be stored. Optionally, you can set the `TraceHex` property to `True` to enable writing of non-printable characters to the trace file in hexadecimal format. When `Tracing` is set to `tlDump`, the current contents of the trace buffer are written to disk.

Dispatch logging

Tracing is a great tool for examining the incoming and outgoing data processed by your program. One of tracing's major strengths is that it shows only the data processed when your program calls the `GetXxx` and `PutXxx` routines. Comparing the resulting trace file to your program logic can often point out logic errors.

In some situations, however, a more appropriate debugging tool is one that shows the true chronology of incoming and outgoing data. It may be more important to see exactly when data arrived at the port rather than seeing how your program processed that data.

The standard Windows communications driver doesn't provide enough information to determine exactly when data arrived. The next best thing is knowing when the Async Professional internal dispatcher got the data, and that's how "dispatch logging" works.

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

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

```

APRO v4.00
Compiler : Delphi 6
Operating System : Windows NT 4.0 Service Pack 4
Time      Type      SubType      Data      OtherData
-----
00000010  TrDatChg  Avail      00000001
00000010  TrgHdAlc  Window     7DDE03CE
00000010  TrgHdAlc  Window     870302A2
00000010  TrDatChg  Avail      00000001
00000010  TrgHdAlc  Procedure   00000000
00000010  TrDatChg  Avail      00000001
00000010  TrigAllc  Data        00000008  rz[0D]
00000010  TrigAllc  Data        00000010  [05]
00000010  TrigAllc  Data        00000018  [10]
00000010  TrigAllc  Data        00000020  [1B]I
00000010  TrigAllc  Status      00000029  (Modem status)

```

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

The first column of the report is a timestamp. It represents the time elapsed from the time dispatch logging was turned on to when the entry was made, measured in milliseconds. The multimedia API TimeGetTime is used to calculate this timestamp, so it should be accurate to the nearest millisecond.

The second column is the major category of log entry, the third column identifies the log entry subtype, the forth column provides additional information related to the event (often a handle or data count), and the remaining column adds any additional information that could be useful for the event being logged.

The following tables identify possible entries in a dispatch log:

Log entry type: Dispatch

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

Subtype - Meaning	Data	Other Data
ReadCom - The Windows comm driver has notified APRO that incoming data is available and APRO's dispatcher has, in turn, read the available data.	The number of bytes read.	The actual data.
WriteCom - APRO has sent data to the Windows comm driver.	The number of bytes sent.	The actual data.
Line status - A line status event has been received from the Windows comm driver.		
Modem status - A modem status event has been received from the Windows comm driver.	The numeric value of the event received.	A translation of the numeric value. (DCTS, DDSR, TERI, DDCD, CTS, DSR, RI, DCD.)

Log entry type: Trigger

A trigger is being dispatched by the dispatcher.

Subtype - Meaning	Data	Other Data
Avail - A data avail event is being dispatched.	The number of bytes ready to be read.	
Timer - A timer event is being dispatched.	The handle of the trigger.	
Data - A data trigger match event is being dispatched.	The handle of the trigger.	
Status - A status trigger is being dispatched.	The handle of the trigger.	

Log entry type: Thread

Occur only if the DebugThreads define is enabled in AWUSER.PAS. They are designed to provide detail about the operation of APRO's threads.

Subtype - Meaning
Start - One of the three background threads associated with each comport (dispatcher thread, communications thread, output thread) is starting.
Exit - One of the three background threads associated with each is terminating.
Sleep - One of the three background threads associated with each is entering a wait state.
Wake - One of the three background threads associated with each is returning from a wait state.

Log entry type: TrigAlloc

A trigger is being allocated.

Subtype - Meaning	Data	Other Data
Data - A data trigger is being allocated.	The handle of the trigger.	What the trigger is being set to trigger on.
Timer - A timer trigger is being allocated.	The handle of the trigger.	
Status - A status trigger is being allocated.	The handle of the trigger.	The type of the status trigger.

Log entry type: TrigDisp

A trigger is being disposed.

Subtype - Meaning	Data
Data - A data trigger is being deleted.	The handle of the trigger.
Timer - A timer trigger is being deleted.	The handle of the trigger.
Status - A status trigger is being deleted.	The handle of the trigger.

Log entry type: TrgHdAlloc

A trigger handler has been allocated.

Subtype - Meaning	Data
Window - A window handle based trigger handler is being registered with the comport.	The window handle.
Procedure - A procedure pointer based trigger handler is being registered with the comport.	
Method - A method pointer based trigger handler is being registered with the comport.	

Log entry type: TrgHdDsp

A trigger handler has been disposed.

Subtype - Meaning	Data
Window - A window handle based trigger handler is being deregistered from the comport.	The window handle.
Procedure - A procedure pointer based trigger handler is being deregistered from the comport.	
Method - A method pointer based trigger handler is being deregistered from the comport.	

Log entry type: TrDatChg

Data associated with the trigger has been changed.

Subtype - Meaning	Data	Other Data
Avail - The data trigger length value is being changed.	The new length.	
Timer - The time-out value for a particular timer trigger is being changed.	The handle of the timer trigger.	The new time.
Status - SetStatusTrigger is being called for a particular status trigger.	The handle of the trigger being changed.	The new value mask for the trigger.

Log entry type: Telnet

Logs the telnet negotiation conversation during a Winsock telnet session. Each SubType shows the type of negotiation being logged.

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

Log entry type: Telnet (cont.)

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

Log entry type: Packet

Events that indicate state changes in packets.

Subtype - Meaning	Other Data
Enable - The packet is being enabled.	The name of the packet component.
If the end condition was a string, the next log entry is another StringPacket event with the value of the end string in the "other data" column. If the end condition was a size event, the next log entry is a SizePacket event.	
Disable - The packet is being enabled.	The name of the packet component.
StringPacket - A string packet event is being dispatched.	The name of the packet component.
If the end condition was a string, the next log entry is another StringPacket event with the value of the end string in the "other data" column. If the end condition was a size event, the next log entry is a SizePacket event.	
SizePacket - Describes the end value for a StringPacket.	The value of the end condition of the previously listed StringPacket.
PcktTimeout - A packet timeout event is being generated.	

Log entry type: Packet (cont.)**Subtype - Meaning****Other Data**

StartStr - Describes the start string for a particular packet.

The value of the start string of an enable sequence.

The event is always generated as a part of the enable sequence for the packet, if it has a start string. See the Enable event above.

EndStr - Describes the current end string for a particular packet.

The end string of an enable sequence.

The event is always generated as a part of the enable sequence for the packet, if it has an end string. See the Enable event above.

Idle - The packet is not currently collecting data and is not waiting for a string.

Waiting - The packet is waiting for its start string to come in.

Collecting - The packet's start condition has been met and thus the packet currently has ownership to the incoming data.

Log entry type: Error

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

Log entry type: Fax

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

Log entry type: XModem

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

Log entry type: YModem

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

Log entry type: ZModem

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

Log entry type: Kermit

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

Log entry type: Ascii

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

Log entry type: BPlus

This type is currently not implemented due to an enumeration capacity limitation in the C++ Builder 3 compiler.

Log entry type: User

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

Logging facility

The state of the logging facility is controlled by setting the Logging property to one of the values shown in Table 2.2.

Table 2.2: *Logging property values*

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

See "Tracing" on page 33, for more information and a related example.

Example

This example shows how to construct and use a comport component. Create a new project, add the following components, and set the property values as indicated in Table 2.3.

Table 2.3: *Example project components and property values*

Component	Property	Value
TApdComPort	ComNumber	<set as needed for your PC>
TButton	Name	'Test'

Double-click on the Test button and modify the generated method to match this:

```
procedure TForm1.TestClick(Sender : TObject);
begin
    ApdComPort1.Output := 'ATZ'^M;
end;
```

This method transmits “ATZ” (the standard modem reset command), followed by a carriage return, to the serial port opened by TApdComPort. If a modem is attached to the serial port it should echo the command, then return “OK.”

Double-click on the TApdComPort OnTriggerAvail event handler in the Object Inspector and modify the generated method to match this:

```
procedure TForm1.ApComPort1TriggerAvail(
    CP : TObject; Count : Word);
var
    I : Word;
    C : Char;
    S : string;
begin
    S := '';
    for I := 1 to Count do begin
        C := ApdComPort1.GetChar;
        case C of
            #0..#31 : {don't display} ;
            else S := S + C;
        end;
    end;
    ShowMessage('Got an OnTriggerAvail event for: ' + S);
end;
```

This method collects all of the received data into the string S, discarding non-printable characters, then displays S using ShowMessage.

To run this program you need to attach a modem to the serial port, then compile and run the project. Clicking on the Test button sends ‘ATZ’^M to the modem. The modem’s responses are shown through one or more message boxes.

Hierarchy

TComponent (VCL)

- ❶ TApdBaseComponent (OOMisc) 8
 - TApdCustomComPort (AdPort)
 - TApdComPort (AdPort)

Properties

AutoOpen	HWFlowOptions	Parity
BaseAddress	InBuffFree	PromptForPort
Baud	InBuffUsed	RI
BufferFull	InSize	RS485Mode
BufferResume	LineBreak	RTS
ComNumber	LineError	StopBits
CTS	LogAllHex	SWFlowOptions
DataBits	Logging	TapiMode
DCD	LogHex	TraceAllHex
DeltaCTS	LogName	TraceHex
DeltaDCD	LogSize	TraceName
DeltaDSR	ModemStatus	TraceSize
DeltaRI	Open	Tracing
DeviceLayer	OutBuffFree	UseEventWord
DSR	OutBuffUsed	❶ Version
DTR	Output	XOffChar
FlowState	OutSize	XOnChar

Methods

ActivateDeviceLayer	FlushInBuffer	PutString
AddDataTrigger	FlushOutBuffer	RemoveAllTriggers
AddStatusTrigger	GetBlock	RemoveTrigger
AddStringToLog	GetChar	SendBreak
AddTimerTrigger	ProcessCommunications	SetBreak
AddTraceEntry	PutBlock	SetStatusTrigger
CharReady	PutChar	SetTimerTrigger

Events

OnPortClose	OnTriggerData	OnTriggerOutbuffUsed
OnPortOpen	OnTriggerLineError	OnTriggerOutSent
OnTrigger	OnTriggerModemStatus	OnTriggerStatus
OnTriggerAvail	OnTriggerOutbuffFree	OnTriggerTimer

Reference Section

ActivateDeviceLayer

virtual method

```
function ActivateDeviceLayer : TApdBaseDispatcher; virtual;  
TApdBaseDispatcher = class;
```

↳ Called when the port is first opened to instantiate a device driver object for the port.

Async Professional includes several device layers. The dlWin32 device layer (the default for 32-bit applications) uses the standard Win32 comms drivers. The dlWin32 device layer have a descendant device layer that is used if TapiMode is set to tmOn. The TAPI device layers rely on TAPI to open and close the serial port, but are otherwise identical to their ancestor device layers. The dlWinsock device layer is available only if you use TApdWinsockPort component (see page 105).

You can create custom device layers by deriving them from TApdBaseDispatcher and creating a new port descendant from TApdCustomComport where you override ActivateDeviceLayer to return the newly defined device layer.

See “Device Independence” on page 725 for more information on device layers.

See also: DeviceLayer, TapiMode

AddDataTrigger

method

```
function AddDataTrigger(  
    const Data : string; const IgnoreCase : Boolean) : Word;
```

↳ Adds a string match trigger to the dispatcher.

Data is the string the dispatcher attempts to match in the received data stream. If IgnoreCase is True, case is not considered when checking for a match.

If the trigger is added successfully, the function returns the handle of the trigger. Otherwise, it generates an exception. No subsequent call is required to activate the trigger.

When the internal dispatcher finds incoming data that matches Data it generates an OnTriggerData event.

The following example tells the internal dispatcher to generate an OnTriggerData event whenever it receives the string “UserID:”. Because False is passed for IgnoreCase the case of the strings must match exactly.

```
ApdComPort.AddDataTrigger('UserID:', False);
```

See also: AddStatusTrigger, AddTimerTrigger, RemoveTrigger

↳ Adds a status trigger of the specified type.

This method adds a status trigger of type `SType`, which is one of the following:

Value	Meaning
<code>stModem</code>	Trigger on modem status change.
<code>stLine</code>	Trigger on line status change.
<code>stOutBuffFree</code>	Trigger on output buffer free above level.
<code>stOutBuffUsed</code>	Trigger on output buffer used below level.
<code>stOutSent</code>	Trigger on any call to <code>PutChar</code> or <code>PutBlock</code> .

See the `SetStatusTrigger` method on page 88 or more information about these status trigger types.

If the trigger is added successfully, the function returns the handle of the trigger; otherwise it generates an exception. The trigger is not activated until a subsequent call to `SetStatusTrigger`. The data associated with the trigger (such as the buffer free level for a `stOutBuffFree` trigger event) is supplied at that time.

The following example adds a status trigger and enables the trigger to generate an `OnTriggerStatus` event as soon as at least 100 bytes become free in the output buffer. Later it deactivates the trigger but does not delete it. Note that status triggers are not self-restarting; the application's message handler must call `SetStatusTrigger` again once an event is generated.

```
var
    StatusHandle : Word;
...
StatusHandle := ApdComPort.AddStatusTrigger(stOutBuffFree);
ApdComPort.SetStatusTrigger(StatusHandle, 100, True);
...
ApdComPort.SetStatusTrigger(StatusHandle, 0, False);
```

See also: `RemoveTrigger`, `SetStatusTrigger`

```
procedure AddStringToLog(S : string);
```

↳ Adds a User-type log entry to the dispatcher log

This procedure can be called to add custom entries to the dispatcher log. These entries will show up in the dispatcher log as “User” entries.

AddStringToLog can be very useful when debugging the application. Custom strings can be added to the log to show when you process events, or start other operations, or just for general logging purposes.

See also: Logging

```
function AddTimerTrigger : Word;
```

↳ Adds a timer trigger.

If the trigger is added successfully, the function returns the handle of the trigger; otherwise, it generates an exception. The timer must be activated subsequently with a call to SetTimerTrigger.

The following example adds a timer trigger and enables it to expire in 36 ticks (2 seconds), when it will generate an OnTriggerTimer event. Note that timer triggers are not self-restarting; the application’s event handler must call SetTimerTrigger again once an event is handled.

```
var  
    Timer : Word;  
...  
Timer := ApdComPort.AddTimerTrigger;  
ApdComPort.SetTimerTrigger(Timer, 36, True);
```

See also: SetTimerTrigger

↳ Adds a trace event to the port's trace queue.

This procedure can be called to add a special entry to the trace buffer. The `CurEntry` parameter indicates the type of entry, normally either 'T' for Transmit or 'R' for Receive. `CurCh` is the character that was transmitted or received.

Although `AddTraceEntry` is intended primarily for internal use by Async Professional routines that send and receive data, you can use it to store additional data in the trace buffer. For example, you might add an entry just before you temporarily suspend tracing.

If `CurEntry` is a character other than 'T' or 'R', it shows up in the tracing report as shown in the following code snippet where 'X' is the `CurEntry` character and 'Y' is `CurCh`:

```
Special-X:  
Y
```

When `AddTraceEntry` is mixed in with a normal set of transmit and receive blocks, it looks something like this:

```
Transmit:  
ATZ[13]  
  
Special-X:  
Y  
  
Receive:  
ATZ[13][13[10]OK[13][10]
```

In this example the program transmitted 'ATZ'<CR>, called `AddTraceEntry('X', 'Y')`, then used one of the `GetXxx` routines to retrieve the received data.

See also: Tracing

```
property AutoOpen : Boolean
```

Default: True

↳ Determines whether the port is automatically opened on demand.

If `AutoOpen` is True and a method or property that requires an open serial port is accessed, the `TAPdComPort` component automatically opens the port. If `AutoOpen` is False, the port must be opened explicitly by setting the `Open` property to True.

See also: Open

BaseAddress**run-time property**`property BaseAddress : Word`

Default: 0

↪ Determines the base address of the port.

Under normal conditions, a program would not need to reference or set this property. The base address has importance only when a program is using RS-485 hardware that uses RTS to control the RS-485 line. In most cases Async Professional is able to determine the base address automatically.

This property is provided for rare cases where Async Professional is not able to automatically determine the base address of the RS-485 port. In these cases, this property must be set manually in order for Async Professional to provide RTS line control for the port.

See “RS-485 support overview” on page 31 for more information on RS-485 support.

See also: RS485Mode

Baud**property**`property Baud : LongInt`

Default: 19200

↪ Determines the baud rate used by the port.

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

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

You can enter a baud rate using the Object Inspector or invoke the SelectBaudRate property editor, which provides a drop-down list box of standard baud rates.

See also: ComNumber, DataBits, Parity, StopBits

BufferFull**property**`property BufferFull : Word`

Default: 0

↪ Determines the input buffer level at which receive flow control is imposed.

When hardware flow control is used, BufferFull should typically be set to 90% of the input buffer size. When software flow control is used, it should typically be set to 75% of the buffer size, since it may take the remote some time to receive the XOff character and stop sending. If flow control is enabled and BufferFull has not been set, or has been set to an invalid value, the level is set to 90% of the input buffer size.

See also: BufferResume, HWFlowOptions, SWFlowOptions

BufferResume**property**`property BufferResume : Word`

Default: 0

↪ Determines the input buffer level at which receive flow control is deactivated.

When hardware flow control is used, BufferResume should typically be set to 10% of the input buffer size. When software flow control is used, it should typically be set to 25% of the buffer size, since it may take the remote some time to receive the XOn character and start sending again. If flow control is enabled and BufferResume has not been set, or has been set to an invalid value, the level is set to 10% of the input buffer size.

See also: BufferFull, HWFlowOptions, SWFlowOptions

CharReady**method**`function CharReady : Boolean;`

↪ Returns True if at least one character is in the dispatcher buffer.

Don't use CharReady in OnTriggerAvail event handlers. Doing so can interfere with the dispatcher's data tracking ability, leading to errors or lost data. This function is provided for the rare cases where you need to call GetChar outside of an event handler.

See also: GetChar

property ComNumber : Word

Default: 0

- ↪ Determines the serial port number (Com1, Com2, ...) used by the TApdComPort component.

ComNumber does not validate the port number. When the port is opened, the Windows communications driver will determine whether the port number is valid and generate an error if it is not. To validate the port, use the IsPortAvailable method described on page 896.

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

This property is ignored when the TAPI and Winsock device layers are used.

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

```
ApdComPort := TApdComPort.Create(Self);
ApdComPort.ComNumber := 1; {use Com1}
ApdComPort.Baud := 9600;
ApdComPort.Parity := pNone;
ApdComPort.DataBits := 8;
ApdComPort.StopBits := 1;
ApdComPort.Open := True;
```

property CTS : Boolean

- ↪ Returns True if the port's "clear to send" line (CTS) is set.

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

```
if ApdComPort.CTS then
  ApdComPort.PutBlock(BigBlock, 1024);
```

See also: DSR

DataBits**property**`property DataBits : Word`

Default: 8

↪ Determines the number of data bits of the port.

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

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

See also: Baud, ComNumber, Parity, StopBits

DCD**read-only, run-time property**`property DCD : Boolean`

↪ Returns True if the port's "data carrier detect" line (DCD) is set.

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

The following example detects carrier loss and handles the error:

```
if not ApdComPort.DCD then
    {handle unexpected disconnect}
```

See also: DeltaDCD

DeltaCTS**read-only, run-time property**`property DeltaCTS : Boolean`

↪ Returns True if the port's "delta clear to send" bit (DeltaCTS) is set.

DeltaCTS is set only if CTS has changed since the last time the application read the value of DeltaCTS.

See also: CTS

DeltaDCD**read-only, run-time property**

```
property DeltaDCD : Boolean
```

- ↳ Returns True if the port's "delta data carrier detect" bit (DeltaDCD) is set.

DeltaDCD is set only if DCD has changed since the last time the application read the value of DeltaDCD.

See also: DCD

DeltaDSR**read-only, run-time property**

```
property DeltaDSR : Boolean
```

- ↳ Returns True if the port's "delta data set ready" bit (DeltaDSR) is set.

DeltaDSR is set only if DSR has changed since the last time the application read the value of DeltaDSR.

See also: DSR

DeltaRI**read-only, run-time property**

```
property DeltaRI : Boolean
```

- ↳ Returns True if the port's "delta ring indicator" bit (DeltaRI) is set.

DeltaRI is set only if RI (the ring indicator) has changed since the last time the application called CheckDeltaRI.

The formal name for this bit is "trailing edge ring indicator" or TERI. DeltaRI is used for consistency with other Async Professional naming conventions.

It is generally better to use DeltaRI to detect incoming calls than to use RI. RI toggles rapidly as rings are detected, making it easy for your application to miss the brief periods that it returns True. By contrast, DeltaRI returns True if any ring was detected since the last call to the routine.

The following example calls the Answer method of TApdModem to connect a modem after a ring is detected:

```
if ApdComPort.DeltaRI then
  Modem.Answer;
```

See also: RI

```
property DeviceLayer : TDeviceLayer
```

```
TDeviceLayer = (dlWin16, dlFossil, dlWin32, dlWinsock);
```

Default: dlWin16 for 16-bit, dlWin32 for 32-bit

↪ Determines the hardware interface used by the port.

Async Professional includes several device layers. The dlWin32 device layer (the default for 32-bit applications) uses the standard Win32 comms drivers. The dlWin32 device layer each has a descendant device layer that is used if TapiMode is set to tmOn. The TAPI device layers rely on TAPI to open and close the serial port, but are otherwise identical to their ancestor device layers. The dlWinsock device layer is available only if you use TApdWinsockPort component (see page 105).

You can create custom device layers by deriving them from TApdBaseDispatcher and creating a new port descendant from TApdCustomComport where you override ActivateDeviceLayer to return the newly defined device layer.

If desired, each individual TApdComPort component can use a different device layer.

See “Device Independence” on page 725 for more information on device layers.

See also: ActivateDeviceLayer

DSR

read-only, run-time property

```
property DSR : Boolean
```

↪ Returns True if the port’s “data set ready” line (DSR) is set.

DSR is a signal that the remote device sets to indicate that it is attached and active. It may be a good idea to check this signal before transmitting and periodically thereafter.

See also: DeltaDSR, CTS

```
property DTR : Boolean
```

Default: True

↪ Determines the current state of the “data terminal ready” signal (DTR).

Some types of remote devices require that this signal be raised before they transmit. For example, the default configuration of many modems is not to transmit data unless the PC has raised the DTR signal.

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

```
ApdComPort := TApdComPort.Create(Self);
ApdComPort.Open := True;
ApdComPort.DTR := False;
...
ApdComPort.DTR := True;
```

See also: RTS

FlowState

read-only, run-time property

```
property FlowState : TFlowControlState
```

```
TFlowControlState = (fcOff, fcOn, fcDsrHold, fcCtsHold,
    fcDcdHold, fcXOutHold, fcXInHold, fcXBothHold);
```

↪ Returns the state of hardware or software flow control.

fcOff indicates that flow control is not in use. fcOn indicates that flow control is enabled, but that blocking is not currently imposed in either direction.

fcDsrHold, fcCtsHold and fcDcdHold indicate that the application cannot transmit because the other side has lowered DSR, CTS, or DCD respectively. Note that Async Professional doesn't currently provide DCD flow control.

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

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

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

See also: `HWFlowOptions`, `SWFlowOptions`

FlushInBuffer

method

```
procedure FlushInBuffer;
```

- ✎ Clears the input buffers used by both the Windows device driver and the Async Professional internal dispatcher.

It also resets all data triggers so as to disregard any cleared data.

The following example flushes all data currently in the input buffer if a line error is detected. You probably shouldn't do this routinely after each line error. Logic like this is usually appropriate only before trying to resynchronize with the transmitter in a file transfer protocol.

```
if ApdComPort.LineError <> leNoError then begin
    ...error handling
    ApdComPort.FlushInBuffer;
end;
```

See also: `FlushOutBuffer`

FlushOutBuffer

method

```
procedure FlushOutBuffer;
```

- ✎ Clears the output buffers used by both the Windows device driver and the Async Professional internal dispatcher.

Any data pending in the output buffer is not transmitted.

The following example discards any data that hasn't yet been transmitted whenever an application function named `ErrorDetected` returns `True` after a remote device reports an error:

```
if ErrorDetected then begin
    ApdComPort.FlushOutBuffer;
    ...resync with remote
end;
```

See also: `FlushInBuffer`

```
procedure GetBlock(var Block; Len : Word);
```

↳ Returns a block of received characters and removes them from the dispatcher buffer.

This routine makes a request to return the next Len received bytes. The data is moved into the buffer referenced by Block. If Block is not large enough to hold Len bytes this will result in a memory overwrite. If fewer than Len bytes are available, none are returned and an EBufferIsEmpty exception is generated. The returned bytes are removed from the Async Professional dispatcher buffer.

To determine if line errors occurred while the communications driver was receiving this data, check the LineError property after calling GetBlock.

The following example calls GetBlock to remove the next 128 bytes from the dispatcher buffer, and handles the various possible outcomes:

```
var
  Block : array[0..127] of Char;
...
try
  ApdComPort.GetBlock(Block, 128);
except
  on E : EAPDException do
    if E is EBufferIsEmpty then begin
      ...protocol error, 128 bytes expected
      raise;
    end;
  end;
end;
```

See also: CharReady, GetChar, InBuffUsed, PeekBlock


```
function GetChar : Char;
```

↳ Returns the next character from the dispatcher buffer.

If at least one character is available in the dispatcher buffer, GetChar returns the first available one.

To determine if line errors occurred while the communications driver was receiving this data, check the LineError property after calling GetChar.

The following example returns the next available character in C:

```
var
  C : Char;
...
if ApdComPort.CharReady then
  C := ApdComPort.GetChar;
```

See also: CharReady, GetBlock, PeekChar

HWFlowOptions

property

```
property HWFlowOptions : THWFlowOptionSet
THWFlowOptionSet = set of THWFlowOptions;
THWFlowOptions = (
  hwfUseDTR, hwfUseRTS, hwfRequireDSR, hwfRequireCTS);
```

Default: Empty set

↳ Determines the hardware flow control options for the port.

When the options are an empty set, as they are by default, there is no hardware flow control. The options can be combined to enable hardware flow control.

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

Receive flow control is enabled by including the hwfUseRTS and/or hwfUseDTR elements in the options set. When enabled, the corresponding modem control signals (RTS and/or DTR) are lowered when the input buffer reaches the level set by the BufferFull property. The remote must recognize these signals and stop sending data while they are held low. Because there is usually little delay before the remote reacts (as there is with software flow control), BufferFull can be set close to the input buffer size, perhaps at the 90% level.

As the application processes received characters, buffer usage eventually drops below the value set by the `BufferResume` property. At that point, the corresponding modem control signals are raised again. The remote must recognize these signals and start sending data again. Again, because there is usually little delay you can set `BufferResume` close to zero, perhaps at 10% of the input buffer size.

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

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

See “Flow Control” on page 711 for more information.

The following example enables bi-directional hardware flow control with limits at the 10% and 90% levels of the buffer. RTS is lowered for receive flow control and CTS is checked for transmit flow control. Later in the application, hardware flow control is disabled.

```
ApdComPort.HWFlowOptions := [hwfUserTS, hwfRequireCTS];
ApdComPort.BufferFull := Trunc(0.9*ApdComPort.InSize);
ApdComPort.BufferResume := Trunc(0.1*ApdComPort.InSize);
...
ApdComPort.HWFlowOptions := [];
```

See also: `DTR`, `FlowState`, `RTS`, `SWFlowOptions`

InBuffFree

read-only, run-time property

property `InBuffFree` : Word

↳ Returns the number of bytes free in the dispatcher buffer.

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

Because the dispatcher automatically drains the Windows buffer using timer and notification messages, its status is rarely relevant to the program.

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

```
if ApdComPort.InBuffFree > 128 then
    ...perform a time-consuming operation
```

See also: `InBuffUsed`

InBuffUsed**read-only, run-time property**

```
property InBuffUsed : Word
```

↳ Returns the number of bytes currently available for reading from the dispatcher buffer.

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

Because the dispatcher automatically drains the Windows buffer using timer and notification messages, this buffer's status is rarely relevant to the program.

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

```
if ApdComPort.InBuffUsed <> 0 then
  ...process data
```

See also: CharReady, InBuffFree

InSize**property**

```
property InSize : Word
```

Default: 4096

↳ Determines the size, in bytes, of the Window communications driver's input buffer.

InSize should always be fairly large, perhaps 4096 or 8192. The larger this size, the less likely the driver loses data if an ill-behaved program takes control of Windows foreground processing for an extended time.

The Windows communication API does not permit changing the buffer size when a serial port is open. When InSize is changed for an open port, the port is closed and re-opened with the new size.

See also: Open, OutSize

LineBreak**read-only, run-time property**

```
property LineBreak : Boolean
```

↳ Returns True if a line break signal was received since the last call to LineBreak.

See also: OnTriggerStatus

property LineError : Word

↳ Returns a non-zero value if line errors have occurred since the last call to LineError.

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

Value	Error Number	Meaning
leBuffer	1	Buffer overrun in COMM.DRV.
leOverrun	2	UART receiver overrun.
leParity	3	UART receiver parity error.
leFraming	4	UART receiver framing error.
leCTSTO	5	Transmit timeout waiting for CTS.
leDSRTO	6	Transmit timeout waiting for DSR.
leDCDTO	7	Transmit timeout waiting for RLSD.
leTxFull	8	Transmit queue is full.

Line errors can occur during calls to any GetXxx or PutXxx method of the port. If your application must detect line errors, it should check LineError after each such call or group of calls, or it should install an OnTriggerLineError event handler.

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

```
ApdComPort.GetBlock(DataBlock, DataLen);  
if ApdComPort.LineError <> 0 then  
    ...error handling
```

See also: OnTriggerLineError

property LogAllHex : Boolean

Default: False

↳ Determines whether all characters in the dispatcher log are written as hex or decimal.

This property is useful when you are processing raw data, instead of a mixture of printable text and raw data.

If `LogAllHex` is `False` (the default), a received string of “123” will be written to the log as literal, printable chars:

```
0000.824 Dispatch ReadCom      0000000A 123
```

If `LogAllHex` is `True`, the same received string will be written to the log in their hexadecimal notation:

```
0000.829 Dispatch ReadCom      0000000A [31][32][33]
```

See also: `LogHex`, `TraceAllHex`

Logging

property

```
property Logging : TTraceLogState
```

```
TTraceLogState = (tlOff, tlOn, tlDump, tlAppend, tlClear, tlPause);
```

Default: `tlOff`

↪ Determines the current logging state.

When `Logging` is set to `tlOff`, as it is by default, no logging is performed.

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

To write the contents of the logging buffer to disk, set `Logging` to `tlDump` (which overwrites any existing file named `LogName`, or creates a new file) or `tlAppend` (which appends to an existing file, or creates a new file). After the component writes to the file it sets `Logging` to `tlOff`.

To clear the contents of the logging buffer and continue logging, set `Logging` to `tlClear`. After the component clears the buffer, it sets `Logging` to `tlOn`.

To temporarily pause logging, set `Logging` to `tlPause`. To resume logging, set `Logging` to `tlOn`.

See “Dispatch logging” on page 36 for more information.

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

```
ApdComPort.Logging := tlOn;
...
ApdComPort.LogName := 'APRO.LOG';
ApdComPort.Logging := tlDump;
```

See also: `AddStringToLog`, `LogHex`, `LogName`, `LogSize`, `Tracing`

LogHex**property**

```
property LogHex : Boolean
```

Default: True

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

See also: LogAllHex, Logging, LogName, LogSize

LogName**property**

```
property LogName : ShortString
```

Default: "APRO.LOG"

- ↳ Determines the name of the file used to store a dispatch log.

The dispatcher log file is written when the Logging property is changed to tlDump or tlAppend. If a path is not provided, the log will be written to whatever directory is the current directory, which may not be where you expect it to be. Specify an explicit path and filename to ensure the log file is stored in the correct location.

See also: Logging

LogSize**property**

```
property LogSize : Word
```

Default: 10000

- ↳ Determines the number of bytes allocated for the dispatch logging buffer.

The assigned value limit is 16 million. Each dispatch entry consumes at least 10 bytes. Many entries use additional buffer space to store a sequence of received or transmitted characters.

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

See also: Logging

```
property ModemStatus : Byte
```

2

↳ Returns the modem status byte and clears all delta bits.

The status is returned in the byte format used by the UART's modem status register. The returned value can be used with the following bit masks to isolate the desired bits:

Mnemonic	Value	Description
DeltaCTSMask	01h	CTS changed since last read.
DeltaDSRMask	02h	DSR changed since last read.
DeltaRIMask	04h	RI changed since last read.
DeltaDCDMask	08h	DCD changed since last read.
CTSMask	10h	Clear to send.
DSRMask	20h	Data set ready.
RIMask	40h	Ring indicator.
DCDMask	80h	Data carrier detect.

You'll probably find it easier to use the CTS, DSR, RI, DCD, and related DeltaXxx properties instead of ModemStatus, but ModemStatus is more efficient when you need to check several signals at once.

ModemStatus also clears the internal delta bits used to indicate changes in the CTS, DSR, RI, and DCD signals. This affects the results of subsequent checks of DeltaCTS, etc.

The following example uses ModemStatus to check for dropped carrier. It would have been simpler in this case to check the DCD property directly.

```
Status := ApdComPort.ModemStatus;
if Status and DCDMask = 0 then
    ...port dropped carrier
```

See also: CTS, DCD, DeltaCTS, DeltaDCD, DeltaDSR, DeltaRI, DSR, RI

OnPortClose

event

property OnPortClose : TNotifyEvent

- ↳ OnPortClose is generated when the serial port associated with the TApdComPort is physically closed.

Setting the Open property to False does not close the associated serial port immediately. Buffers and structures need to be reset, and the physical hardware needs to be closed before the port can be considered closed. This event is generated when the physical port is actually closed and available to other processes.

See also: OnPortOpen, Open

OnPortOpen

event

property OnPortOpen : TNotifyEvent

- ↳ OnPortOpen is generated when the serial port associated with the TApdComPort is physically opened.

Setting the Open property to True begins a series of actions, ending with the physical serial port being opened and ready for use. This includes creating the buffers, data structures and the three threads required by the component. This process may take a few milliseconds, depending on the responsiveness of the Windows serial port drivers, unconventional configurations, etc. This event is generated when the physical serial port is opened and ready for use.

See also: OnPortClose, Open

OnTrigger

event

property OnTrigger : TTriggerEvent

```
TTriggerEvent = procedure(
  CP : TObject; Msg, TriggerHandle, Data : Word) of object;
```

- ↳ Defines an event handler that is called whenever any serial data trigger occurs.

OnTrigger can be used to handle all kinds of trigger events in one location. Normally it is easier to use the more specific kinds of serial data triggers. The OnTrigger event handler is always called first, then the more specific event handlers are also called if they are assigned.

CP is the TApdComPort component that generated the trigger. Msg is the Windows message that specifies the kind of trigger:

Trigger	Description
APW_TRIGGERAVAIL	Corresponds to OnTriggerAvail.
APW_TRIGGERDATA	Corresponds to OnTriggerData.
APW_TRIGGERTIMER	Corresponds to OnTriggerTimer.
APW_TRIGGERSTATUS	Corresponds to OnTriggerStatus.

TriggerHandle is the handle number returned when the trigger was added. Data is a numeric value that is relevant for the APW_TRIGGERAVAIL, APW_TRIGGERDATA, and APW_TRIGGERSTATUS events. See the corresponding event handlers for more information.

The following example waits for and responds to a login prompt, processing APW_TRIGGERDATA, APW_TRIGGERAVAIL, and APW_TRIGGERTIMER messages in a single routine:

```
DataTrig := ApdComPort.AddDataTrigger('login:', True);
TimerTrig := ApdComPort.AddTimerTrigger;
ApdComPort.SetTimerTrigger(TimerTrig, 182, True);
...

procedure TMyForm.ApdComPortTrigger(
  CP : TObject; Msg, TriggerHandle, Data : Word);
var
  I : Word;
  C : Char;
```

```

begin
  case Msg of
    APW_TRIGGERDATA :
      {got 'login', send response}
      ApdComPort.PutString('myname');
    APW_TRIGGERAVAIL :
      {extract and display/process the data}
      for I := 1 to Data do begin
        C := ApdComPort.GetChar;
        ...process data
      end;
    APW_TRIGGERTIMER :
      {timed out waiting for login prompt, handle error}
      ...
  end;
end;

```

See also: OnTriggerAvail, OnTriggerData, OnTriggerStatus, OnTriggerTimer

OnTriggerAvail

event

```
property OnTriggerAvail : TTriggerAvailEvent
```

```
TTriggerAvailEvent = procedure(
  CP : TObject; Count : Word) of object;
```

↳ Defines an event handler that is called whenever a certain amount of serial input data is available for processing.

This event is generated when data has been transferred into the dispatcher buffer.

CP is the TApdComPort component that generated the trigger. Count is the actual number of bytes that are available to read at the instant the event is generated. Your event handler should process exactly the number of bytes of input equal to the value of the Count parameter (by calling GetChar or GetBlock). It should not use CharReady in a loop to process all available bytes since additional bytes may have transferred into the buffer while the event handler is active, and removing those bytes could interfere with the dispatcher's data tracking mechanism.

If several parts of the same application are using the same comports component and each part installs its own OnTriggerAvail handler (as would occur if a terminal window and a file transfer protocol were both in use), the Async Professional dispatcher takes pains to ensure that all of them can read the same data. Even if the first handler to get control calls GetChar to read Count bytes of data, the second and subsequent handlers will also be able to read the same bytes of data by calling GetChar. After all of the OnTriggerAvail handlers have returned, the dispatcher determines the largest number of characters read by any of the handlers and removes those characters from the dispatcher buffer. If any data remains in the

buffer, the dispatcher immediately generates another OnTriggerAvail event. Hence, if one handler reads fewer characters than the other handlers, it will miss seeing the characters it did not read on its first opportunity.

☛ **Caution:** Be sure to process the exact number of bytes passed in the Count parameter of this handler. If you process fewer bytes, you risk losing characters to another component extracting data during the event (such as the terminal). If you process more than Count bytes, you risk receiving events for overlapping data – which may eventually lead to an EBufferIsEmpty exception.

The following example collects incoming data until it finds a carriage return character (ASCII 13). If the incoming data stream contained “TurboPower Software” <CR>, ApdComPortTriggerAvail would be called one or more times until the entire string except <CR> was received. ApdComPortTriggerData would then be called and could process the complete string. ApdComPortTriggerAvail would then be called again with the <CR> and any other data that followed it. ApdComPortTriggerData would not be called again in this example, because the handler disables the data trigger.

```
const
  S : string = '';
...
  CRTrig := ApdComPort.AddDataTrigger(#13, False);
...
procedure TMyForm.ApdComPortTriggerData(
  CP : TObject; TriggerHandle : Word);
begin
  if TriggerHandle = CRTrig then begin
    ...do something with S
    ApdComPort.RemoveTrigger(TriggerHandle);
  end;
end;

procedure TMyForm.ApdComPortTriggerAvail(
  CP : TObject; Count : Word);
var
  I : Word;
begin
  for I := 1 to Count do
    S := S + ApdComPort.GetChar;
  end;
```

See also: OnTrigger, OnTriggerData

```
property OnTriggerData : TTriggerDataEvent  
  
TTriggerDataEvent = procedure(  
    CP : TObject; TriggerHandle : Word) of object;
```

- ↳ Defines an event handler that is called whenever a string matching a predefined goal is detected in the input buffer.

The event is generated as a result of adding a match string using `AddDataTrigger`. When the dispatcher finds a matching string in the input buffer, it generates an `OnTriggerAvail` event for the bytes leading up to the match, then generates an `OnTriggerData` event for the match. Finally, it generates another `OnTriggerAvail` event for the matched data itself.

CP is the `TApdComPort` component that generated the trigger. `TriggerHandle` is the handle number returned when the trigger was added.

Note that data match triggers remain active until explicitly removed. The event handler can call `RemoveTrigger`, passing `TriggerHandle` as the parameter, to remove the trigger that just generated the event.

See also: `AddDataTrigger`, `OnTrigger`, `OnTriggerAvail`

OnTriggerLineError**event**

```
property OnTriggerLineError : TTriggerLineErrorEvent  
  
TTriggerLineErrorEvent = procedure(  
    CP : TObject; Error : Word; LineBreak : Boolean) of object;
```

- ↳ Defines an event handler that is called whenever the dispatcher detects a line error or line break in the received data.

This event handler is called in a subset of the cases where the more general `OnTriggerStatus` handler is called. `OnTriggerStatus` is called first when a line error is detected, even if an `OnTriggerLineError` handler is installed.

CP is the `TApdComPort` component that generated the trigger. `Error` is a numeric code that indicates the most severe line error detected. See the `LineError` property for details. The `LineBreak` parameter is `True` if a line break was detected.

Note that status triggers are not self-restarting. The event handler must call `SetStatusTrigger` again to reactivate the trigger as needed.

The following example adds a status trigger for line errors and line breaks. The events are handled using an `OnTriggerLineError` event handler.

```

TrigLE : Word;
...
TrigLE := ApdComPort.AddStatusTrigger(stLine);
ApdComPort.SetStatusTrigger(
    TrigLE, lsParity or lsFraming or lsOverrun or lsBreak, True);
...
procedure TMyForm.ApdComPortTriggerLineError(
    CP : TObject; Error : Word; LineBreak : Boolean);
begin
    if Error <> leNone then
        ...process line error
    if LineBreak then
        ...process line break
    {reactivate trigger}
    ApdComPort1.SetStatusTrigger(
        TrigLE, lsParity or lsFraming or lsOverrun or lsBreak, True);
end;

```

See also: `LineError`, `OnTriggerStatus`, `SetStatusTrigger`

OnTriggerModemStatus

event

property `OnTriggerModemStatus` : `TNotifyEvent`

- ↳ Defines an event handler that is called whenever the dispatcher detects that modem status signals have changed.

This event handler is called in a subset of the cases where the more general `OnTriggerStatus` handler is called. `OnTriggerStatus` is called first when a modem status change is detected, even if an `OnTriggerModemStatus` handler is installed.

The parameter passed to the `TNotifyEvent` is the `TApdComPort` component that generated the trigger. The `TApdComPort`'s modem status properties can be checked by the event handler to determine the exact reason for the event. No `TriggerHandle` is passed to the event handler, so it is not possible to distinguish between multiple modem status triggers in this event handler. If you need to do so, use the `OnTriggerStatus` event instead.

Note that status triggers are not self-restarting. The event handler must call `SetStatusTrigger` again to reactivate the trigger as needed.

The following example adds and activates a modem status trigger for ring indicators and changes in DSR. Modem status changes are handled using an OnTriggerModemStatus event handler.

```

TrigMS : Word;
...
TrigMS := ApdComPort.AddStatusTrigger(stModem);
ApdComPort.SetStatusTrigger(
    TrigMS, msRingDelta or msDSRDelta, True);
...
procedure TMyForm.ApdComPortTriggerModemStatus(CP : TObject);
begin
    if ApdComPort.DeltaRI then
        ...handle ring
    if ApdComPort.DeltaDSR then
        ...handle change in DSR
        {reactivate trigger}
    ApdComPort.SetStatusTrigger(
        TrigMS, msRingDelta or msDSRDelta, True);
end;
```

See also: CTS, DCD, DeltaCTS, DeltaDCD, DeltaDSR, DeltaRI, DSR, ModemStatus, RI

OnTriggerOutbuffFree

event

property OnTriggerOutbuffFree : TNotifyEvent

- ↳ Defines an event handler that is called whenever the dispatcher detects that free space in its output buffer has reached a certain level.

This event handler is called in a subset of the cases where the more general OnTriggerStatus handler is called. OnTriggerStatus is called first when sufficient free space is detected, even if an OnTriggerOutbuffFree handler is installed. If the output buffer level is already below the specified level when the trigger is activated, an OnTriggerOutbuffFree event is generated the next time the internal dispatcher gains control.

The parameter passed to the TNotifyEvent is the TApdComPort component that generated the trigger.

Note that status triggers are not self-restarting. The event handler must call SetStatusTrigger again to reactivate the trigger as needed.

The following example adds and activates a status trigger for OnTriggerOutbuffFree events; when the output buffer free level reaches 255 or greater ApdComPortTriggerOutbuffFree transmits BigString:

```
TrigOBF : Word;
...
TrigOBF := ApdComPort.AddStatusTrigger(stOutbuffFree);
ApdComPort.SetStatusTrigger(TrigOBF, 255, True);
...
procedure TMyForm.ApdComPortTriggerOutbuffFree(CP : TObject);
begin
    {buffer has at least 255 bytes free, transmit a big string}
    ApdComPort.Output := BigString;
end;
```

See also: OnTriggerStatus, OnTriggerOutbuffUsed

OnTriggerOutbuffUsed

event

property OnTriggerOutbuffUsed : TNotifyEvent

- ↳ Defines an event handler that is called whenever the dispatcher detects that used space in its output buffer has dropped below a certain level.

This event handler is called in a subset of the cases where the more general OnTriggerStatus handler is called. OnTriggerStatus is called first when used space drops below a particular level, even if an OnTriggerOutbuffUsed handler is installed. If the output buffer already contains fewer bytes than the specified level when the trigger is activated, an OnTriggerOutbuffUsed event is generated as soon as the internal dispatcher gains control.

The parameter passed to the TNotifyEvent is the TApdComPort component that generated the trigger.

Note that status triggers are not self-restarting. The event handler must call SetStatusTrigger again to reactivate the trigger as needed.

The following example adds and activates a status trigger for an OnTriggerOutbuffUsed event; when the output buffer used level drops below 100 bytes ApdComPortTriggerOutbuffUsed is called and transmits additional data:

```
TrigOBU : Word;
...
TrigOBU := ApdComPort.AddStatusTrigger(stOutBuffUsed);
ApdComPort.SetStatusTrigger(TrigOBU, 100, True);
...
procedure TMyForm.ApdComPortTriggerOutbuffUsed(CP : TObject);
```

```

begin
    {buffer almost empty, start filling up again}
    ApdComPort.Output := Stuff;
    ApdComPort.Output := MoreStuff;
    ApdComPort.Output := EvenMoreStuff;
    ...
end;

```

See also: OnTriggerStatus, OnTriggerOutbuffFree

OnTriggerOutSent

event

```
property OnTriggerOutSent : TNotifyEvent
```

↳ Defines an event handler that is called whenever the dispatcher gets a request to send one or more characters.

This event handler is called in a subset of the cases where the more general OnTriggerStatus handler is called. OnTriggerStatus is called first when an output request occurs, even if an OnTriggerOutSent handler is installed. The parameter passed to the TNotifyEvent is the TApdComPort component that generated the trigger.

Unlike most triggers, OnTriggerOutSent does not need to be reset. The event is always generated for output events until the trigger is deactivated.

The following example adds a status trigger for OnTriggerOutSent; thereafter, each time the program calls any transmit method or property ApdComPort.TriggerOutSent is called to update a status display:

```

TrigOS : Word;
...
TrigOS := ApdComPort.AddStatusTrigger(stOutSent);
...
procedure TMyForm.ApdComPort.TriggerOutSent(CP : TObject);
begin
    ...update display to indicate data was transmitted
end;

```

See also: OnTriggerStatus


```
property OnTriggerStatus : TTriggerStatusEvent
```

```
TTriggerStatusEvent = procedure(
  CP : TObject; TriggerHandle : Word) of object;
```

↳ Defines an event handler that is called whenever a line status change of some kind is detected.

This event handler combines the events described under OnTriggerLineError, OnTriggerModemStatus, OnTriggerOutbuffFree, OnTriggerOutbuffUsed, and OnTriggerOutSent.

CP is the TApdComPort component that generated the trigger. TriggerHandle is the handle number returned when the trigger was added.

With the exception of OnTriggerOutSent, status triggers are not self-restarting. The event handler must call SetStatusTrigger again to reactivate the trigger as needed.

The following example adds and activates status triggers for line errors and modem status changes. Subsequent status events are handled by ApdComPortTriggerStatus.

```
TrigLE : Word;
TrigMS : Word;
...
TrigLE := ApdComPort.AddStatusTrigger(stLine);
TrigMS := ApdComPort.AddStatusTrigger(stModem);
ApdComPort.SetStatusTrigger(
  TrigLE, lsParity or lsFraming or lsOverrun or lsBreak, True);
ApdComPort.SetStatusTrigger(
  TrigMS, msRingDelta or msDSRDelta, True);
...
procedure TMyForm.ApdComPortTriggerStatus(
  CP : TObject; TriggerHandle : Word);
begin
  if TriggerHandle = TrigLE then begin
    ...handle line error or break
    ...reset line error trigger
  end else if TriggerHandle = TrigMS then begin
    ...handle modem status change
    ...reset modem status trigger
  end;
end;
```

See also: OnTriggerLineError, OnTriggerModemStatus, OnTriggerOutbuffFree, OnTriggerOutbuffUsed, OnTriggerOutSent

```
property OnTriggerTimer : TTriggerTimerEvent
TTriggerTimerEvent = procedure(
    CP : TObject; TriggerHandle : Word) of object;
```

↳ Defines an event handler that is called when an Async Professional timer expires.

CP is the TApdComPort component that generated the trigger. TriggerHandle is the handle number returned when the trigger was added.

Note that timer triggers are not self-restarting. The event handler must call SetTimerTrigger again to reactivate the trigger as needed.

The following example adds and activates two timer triggers. After 10 seconds and 60 seconds elapse, events are generated and handled by ApdComPortTriggerTimer.

```
Timer1, Timer2 : Word;
...
Timer1 := ApdComPort.AddTimerTrigger;
Timer2 := ApdComPort.AddTimerTrigger;
ApdComPort.SetTimerTrigger(Timer1, 182, True);
ApdComPort.SetTimerTrigger(Timer2, 1092, True);
...
procedure TMyForm.ApdComPortTriggerTimer(
    CP : TObject; TriggerHandle : Word);
begin
    if TriggerHandle = Timer1 then begin
        ...handle 10 second timeout condition
        {restart timer}
        ApdComPort.SetTimerTrigger(Timer1, 182, True);
    end else begin
        ...handle 60 second timeout condition
        {restart timer}
        ApdComPort.SetTimerTrigger(Timer2, 1092, True);
    end;
end;
```

See also: AddTimerTrigger, SetTimerTrigger

property Open : Boolean

Default: False

- ↪ Determines whether the physical port is opened and initialized with all current port properties.

Open must be set to True before a comPort component can send or receive characters. If the AutoOpen property is set to True, the comPort component will open itself automatically under many conditions: calling any I/O method or property or when a component that uses a TApdComPort is loaded.

When Open is set to True, the TApdComPort uses all current property settings to allocate input and output buffers, open the physical port, initialize the line settings and flow control settings, and enable or disable tracing and logging. It then registers a low-level trigger for the port, which gets the first look at all trigger events and passes control on to the appropriate OnTriggerXxx event handlers.

When Open is set to False, the TApdComPort turns off tracing and logging (by setting the associated properties to tlDump, which creates an output file if any information has been buffered), closes the physical port, and deallocates input and output buffers.

There is no harm done by setting Open to True when it is already True, or setting it to False when it is already False.

See also: AutoOpen, OnPortClose, OnPortOpen

OutBuffFree**read-only, run-time property**

property OutBuffFree : Word

- ↪ Returns the number of bytes free in the output buffer.

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

The following example checks for sufficient output buffer space to transmit a block of NeededSpace bytes. If enough space is available the block is transmitted. Otherwise, a status trigger is added to detect the required free space. The code assumes that an OnTriggerStatus event handler has already been activated.

```
if ApdComPort.OutBuffFree >= NeededSpace then
  ApdComPort.PutBlock(Data, NeededSpace)
else begin
  MyHandle := ApdComPort.AddStatusTrigger(stOutBuffFree);
  ApdComPort.SetStatusTrigger(MyHandle, NeededSpace, True);
end;
```

See also: OutBuffUsed

OutBuffUsed

read-only, run-time property

```
property OutBuffUsed : Word
```

↳ Returns the number of bytes currently in the output buffer.

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

The following example checks to see if any outgoing data is still in the output buffer. If so, it sets a status trigger to go off once the buffer is completely empty. The code assumes that an OnTriggerStatus event handler has already been activated.

```
if ApdComPort.OutBuffUsed <> 0 then begin
  MyHandle := ApdComPort.AddStatusTrigger(stOutBuffUsed);
  ApdComPort.SetStatusTrigger(MyHandle, 0, True);
end;
```

See also: OutBuffFree

Output

write-only, run-time property

```
property Output : string
```

↳ Transmits its assigned value through the port.

Assigning a value to Output is equivalent to calling the PutString method with that same string.

The following example sends a dial string out the port:

```
ApdComPort.Output := 'ATDT555-1212'^M;
```

See also: PutChar, PutString

OutSize**property**

```
property OutSize : Word
```

Default: 4096

- ↳ Determines the size, in bytes, of the output buffer used by the Windows communications driver.

OutSize must be large enough to hold the largest block of data that you might transmit at one time (using PutBlock, for example). For file transfer protocols OutSize must be at least 2078 bytes. The recommended setting is 4096, which is large enough to work with all protocols but not so large that it is wasteful.

To obtain a non-default buffer size, OutSize must be set before the port is opened.

See also: Open, InSize

Parity**property**

```
property Parity : TParity
```

```
TParity = (pNone, pOdd, pEven, pMark, pSpace);
```

Default: pNone

- ↳ Determines the parity checking mode of the port.

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

See also: Baud, ComNumber, DataBits, StopBits

ProcessCommunications**method**

```
procedure ProcessCommunications;
```

- ↳ Calls the internal dispatcher one time.

This method is used by the Winsock device layer.

An application should call this routine if it needs to receive data during lengthy processing where the application's message loop isn't running, or the dispatcher thread is otherwise blocked.

The internal dispatcher, which is responsible for retrieving data from the Windows communication driver, is normally called from an application's message loop. If an application isn't calling its message loop then no new received data will be retrieved. To retrieve new data the application must call `ProcessCommunications`, usually in a loop, until it receives its data or times out.

Note that `ProcessCommunications` is provided for those cases where an application must wait (for timing reasons) for a particular response. Normally, an application would use `OnTriggerAvail` and `OnTriggerData` event handlers to wait for data.

The following example sends a string and waits for a response:

```
ET : EventTimer;
S : string;
...
S := '';
ApdComPort.Output := 'login: ';
NewTimer(ET, 182);
repeat
  ApdComPort.ProcessCommunications;
  if ApdComPort.CharReady then
    S := S + ApdComPort.GetChar;
until (S = 'ABC') or TimerExpired(ET);
```

See also: `AddDataTrigger`

PromptForPort

property

property `PromptForPort` : Boolean

Default: True

↪ Indicates whether the user should be prompted for the serial port number.

If `PromptForPort` is True and `ComNumber` is zero, a dialog is displayed to prompt the user for the serial port when the port is opened:

If `PromptForPort` is False and `ComPort` is zero, an `ENoPortSelected` exception is raised when the port is opened. This is the same behavior as older versions of Async Professional, which do not have a `PromptForPort` property.

See also: `ComPort`

PutBlock**method**

```
function PutBlock(const Block; const Len : Word) : Integer;
```

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

The communications driver then transmits the block byte-by-byte as fast as possible.

When there is insufficient free space in the output buffer, the documented behavior of the Windows communications driver is to delete old data from the buffer. To avoid this behavior, programs should always check `OutBuffFree` before calling any `PutXxx` method or assigning a value to the `Output` property.

`Block` refers to the block of data and `Len` is the number of bytes to transmit. `Len` must be smaller than the current value of the `OutSize` property.

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

```
if ApdComPort.OutBuffFree >= 20 then
  ApdComPort.PutBlock(Block, 20);
```

See also: `OutBuffFree`, `PutChar`, `PutString`

PutChar**method**

```
procedure PutChar(const C : Char);
```

↳ Copies a single character to the output buffer of the Windows communications driver.

The communications driver then transmits the character as soon as possible.

The following example transmits one character after assuring that space is available:

```
if ApdComPort.OutBuffFree >= 1 then
  ApdComPort.PutChar(C);
```

See also: `OutBuffFree`, `PutBlock`, `PutString`

PutString**method**

```
procedure PutString(const S : string);
```

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

The communications driver then transmits the string as soon as possible. The length byte of the string is not transmitted.

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

```
S := 'Guinness Stout';
if ApdComPort.OutBuffFree >= Length(S) then
  ApdComPort.PutString(S);
```

See also: OutBuffFree, Output, PutBlock, PutChar

RemoveAllTriggers

method

```
procedure RemoveAllTriggers;
```

↪ Deactivates all triggers added to this port.

Use this routine when your program changes modes and requires completely new triggers. Destroying or closing a port automatically removes all of its triggers.

⚠ **Caution:** Calling this method effectively disables the comport component since it removes all triggers, including the ones that Async Professional requires internally for normal operation of the comport and associated components.

Normally, it's best to keep track of the triggers you add and remove them individually using RemoveTrigger when they are no longer needed.

See also: Open, RemoveTrigger

RemoveTrigger

method

```
procedure RemoveTrigger(Handle : Word);
```

↪ Deactivates a specified trigger.

Handle is the handle returned when the trigger was added. If no matching trigger handle is found, no error is generated. If Handle is a valid trigger, Handle is set to zero when removed.

The following example adds and uses a timer trigger, and later removes it:

```
var
  MyHandle : Word;
...
MyHandle := ApdComPort.AddTimerTrigger;
ApdComPort.SetTimerTrigger(MyHandle, 36, True);
...
ApdComPort.RemoveTrigger(MyHandle);
```

See also: RemoveAllTriggers

RI**read-only, run-time property**`property RI : Boolean`

↳ Returns True if the port's "ring indicator" line (RI) is set.

Because the ring indicator line fluctuates rapidly as rings occur, the DeltaRI property is much more reliable for detecting an incoming call.

See also: DeltaRI

RS485Mode**property**`property RS485Mode : Boolean`

Default: False

↳ Determines whether the RTS line should be raised or lowered automatically when transmitting data.

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

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

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

See "RS-485 support overview" on page 31 for more information on RS-485 support.

See also: BaseAddress

RTS**property**`property RTS : Boolean`

Default: True

↳ Determines the current state of the "request to send" signal (RTS).

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

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

```
ApdComPort := TApdComPort.Create(Self);
ApdComPort.Open := True;
ApdComPort.RTS := False;
...
ApdComPort.RTS := True;
```

See also: DTR, HWFlowOptions

SendBreak

method

```
procedure SendBreak(Ticks : Word; Yield : Boolean);
```

↳ Transmits a break signal.

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

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

SetBreak

method

```
procedure SetBreak(BreakOn : Boolean);
```

↳ Raises or lowers the break signal.

This method will begin transmission of the break signal if BreakOn is True; or stops transmission of the break signal if BreakOn is False. Use this method if you need to transmit the break signal for an undetermined time period.

See also: SendBreak

```
procedure SetStatusTrigger(const Handle : Word;  
    const Value : Word; const Activate : Boolean);
```

↳ Activates or deactivates a status trigger.

Status triggers are activated in two steps. The trigger is added using `AddStatusTrigger`, then the trigger is activated using `SetStatusTrigger`. The trigger type is specified when the trigger is added, and the exact trigger condition is specified when the trigger is activated.

`Handle` is the value that was returned by the call to `AddStatusTrigger`. The interpretation of `Value` varies between the trigger types, as described below. `Activate` is `True` to activate the trigger, `False` to deactivate it. When `Activate` is `False` the `Value` parameter is ignored.

For triggers of type `stModem`, `Value` is a bit mask that contains one or more of the following options:

Option	Description
<code>msCTSDelta</code>	Trigger when CTS changes.
<code>msDSRDelta</code>	Trigger when DSR changes.
<code>msRingDelta</code>	Trigger when a ring is detected.
<code>msDCDDelta</code>	Trigger when DCD changes.

For the `msCTSDelta`, `msDSRDelta`, and `msDCDDelta` options `SetStatusTrigger` saves the current state of the corresponding modem signals and checks for changes to those signals. When a change from the original state is detected an `OnTriggerStatus` event is generated. If a single trigger is used to monitor multiple signals, the message response routine must check the appropriate modem status properties to determine which signal actually changed state. Alternatively, a separate trigger can be added for each modem signal. Once a trigger message is sent the trigger is disabled, even if some of the monitored signals did not change state.

The `msRingDelta` option triggers an `OnTriggerStatus` event at the end of the next incoming ring signal, immediately after the audible termination of the ring. In order to receive another `OnTriggerStatus` event using `msRingDelta`, the application must not only call `SetStatusTrigger` again, but it must also read the `DeltaRI` property to clear the ring condition in the modem status register.

An `stModem` trigger also generates an `OnTriggerModemStatus` event. Note, however, that no trigger handle is passed to the `OnTriggerModemStatus` event handler, so it cannot distinguish among multiple different triggers. If you need to do this, use an `OnTriggerStatus` event handler instead.

For triggers of type `stLine`, `Value` is a bit mask that contains one or more of the following options:

Option	Description
<code>lsOverrun</code>	Trigger on UART overrun errors.
<code>lsParity</code>	Trigger on parity errors.
<code>lsFraming</code>	Trigger on framing errors.
<code>lsBreak</code>	Trigger on a received line break signal.

If a single trigger is used to monitor multiple line status signals, the `OnTriggerStatus` event handler must read the `LineError` property to determine the most severe error. When `lsBreak` is combined with other options the response routine must read both `LineError` and `LineBreak` to determine whether the trigger was caused by an error or by a received line break.

An `stLine` trigger also generates an `OnTriggerLineError` event, which passes the current values of `LineError` and `LineBreak` as parameters to its handler.

For triggers of type `stOutBuffFree`, an `OnTriggerStatus` event is generated when the number of bytes free in the output buffer is greater than or equal to `Value`. An `OnTriggerOutbuffFree` event is also generated by the trigger.

For triggers of type `stOutBuffUsed`, an `OnTriggerStatus` event is generated when the number of bytes used in the output buffer is less than or equal to `Value`. An `OnTriggerOutbuffUsed` event is also generated by the trigger.

For triggers of type `stOutSent`, `Value` is not used. Here, an `OnTriggerStatus` event is generated whenever `PutChar`, `PutString`, or `PutBlock` is called. However, the event is not generated directly from these routines, but is instead generated the next time the dispatcher gains control. Only one event is generated even if multiple `PutXxx` calls were made or the `Output` property was assigned since the last time the dispatcher ran.

All status triggers except `stOutSent` must be restarted within the message handler. That is, the triggers generate a single message and do not restart themselves automatically.

The following example adds an `stOutBuffFree` status trigger and activates it to send a message when at least 100 bytes are free in the output buffer:

```
var
  MyHandle : Word;
...
MyHandle := ApdComPort.AddStatusTrigger(stOutBuffFree);
ApdComPort.SetStatusTrigger(MyHandle, 100, True);
```

The following example adds an stModem trigger and activates it to send a message when either the DSR or CTS signal changes from its current state:

```
var
    MyHandle : Word;
...
MyHandle := ApdComPort.AddStatusTrigger(stModem);
ApdComPort.SetStatusTrigger(
    MyHandle, msDSRDelta or msCTSDelta, True);
```

See also: AddStatusTrigger, OnTriggerStatus

SetTimerTrigger

method

```
procedure SetTimerTrigger(const Handle : Word;
    const Ticks : LongInt; const Activate : Boolean);
```

↪ Activates or deactivates a timer trigger.

Timer triggers are activated in two steps. The trigger is added using AddTimerTrigger, then the trigger is activated using SetTimerTrigger. The duration of the timer is specified when the trigger is activated.

Handle is the handle returned when the trigger was added. Ticks is the duration of the timer in BIOS clock ticks (a tick is approximately 55 milliseconds).

Activate is True to activate the trigger, False to deactivate it. When Activate is False the Ticks parameter is ignored.

After the specified time elapses the internal dispatcher generates an OnTriggerTimer event. The trigger handle is passed to the event handler so that it can detect which timer expired.

Timer triggers generate a single OnTriggerTimer event. The timer is automatically disabled after it triggers once. To reuse the timer your program must call SetTimerTrigger again.

The following example adds a timer trigger and activates it with a 36 tick (2 second) timeout:

```
var
    MyHandle : Word;
...
MyHandle := ApdComPort.AddTimerTrigger;
ApdComPort.SetTimerTrigger(MyHandle, 36, True);
```

See also: AddTimerTrigger

```
property StopBits : Word
```

Default: 1

↪ Determines the number of stop bits of the port.

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

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

See also: Baud, ComNumber, DataBits, Parity

SWFlowOptions

```
property SWFlowOptions : TSWFlowOptions
```

```
TSWFlowOptions = (swfNone, swfReceive, swfTransmit, swfBoth);
```

Default: swfNone

↪ Determines the software flow control options for the port.

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

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

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

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

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

The following example enables bi-directional software flow control with limits at the 25% and 75% levels of the buffer. The default characters are used for XOn and XOff. Later in the application, software flow control is disabled.

```

ApdComPort.BufferFull := Trunc(0.75*ApdComPort.InSize);
ApdComPort.BufferResume := Trunc(0.25*ApdComPort.InSize);
ApdComPort.SWFlowOptions := swfBoth;
...
ApdComPort.SWFlowOptions := swfNone;

```

See also: FlowState, HWFlowOptions

TapiMode

property

```

property TapiMode : TTapiMode
TTapiMode = (tmNone, tmAuto, tmOn, tmOff);

```

Default: tmAuto

↪ Determines whether a TApdComPort can be controlled by a TApdTapiDevice.

A TApdTapiDevice cannot work by itself; it must work in conjunction with a TApdComPort. When a TApdTapiDevice is created, it searches the form for a TApdComPort. If it finds one, it checks the comport component's TapiMode property to determine whether it can be used by the TApdTapiDevice.

If TapiMode is tmAuto (the default), the TApdComPort is available for TAPI use. The TApdTapiDevice saves a pointer to the TApdComPort and sets these property values:

```

ApdComPort.TapiMode := tmOn;
ApdComPort.AutoOpen := False;
ApdComPort.Open := False;

```

TapiMode is changed to tmOn to indicate that the TApdComPort is being controlled by the associated TApdTapiDevice. AutoOpen and Open are both set to False because the TApdComPort should no longer control when it is opened or closed—that is now done by TAPI.

To turn off TAPI mode, or to prevent a TAPI device from taking control of the TApdComPort, set TapiMode to tmOff. To re-enable TAPI mode later, set TapiMode back to tmAuto or tmOn. You must also set AutoOpen and Open to False because the TApdTapiDevice automatically sets these properties only when either the TApdTapiDevice or TApdComPort are first created.

The value tmNone isn't used.

See the ADXPORT form/unit in the TERMDemo demonstration program (see the Async Professional Developer's Guide) for an example of a program that uses both TAPI devices and direct serial port access. It modifies TapiMode accordingly as the user selects either TAPI devices or direct serial ports.

See "Chapter 8: TAPI Components" on page 203 for more information on TAPI.

See also: AutoOpen, Open

TraceAllHex **property**

property TraceAllHex : Boolean

Default: False

- ↪ Determines when the trace log will contain literal printable characters, or if all characters will be written in hexadecimal notation.

See also: LogAllHex, Tracing

TraceHex **property**

property TraceHex : Boolean

Default: True

- ↪ Determines whether non-printable characters stored in a trace file are written using hexadecimal or decimal notation.

See also: TraceAllHex, TraceName, TraceSize, Tracing

TraceName **property**

property TraceName : ShortString

Default: APRO.TRC

- ↪ Determines the name of the file used to store a trace.

See also: Tracing

TraceSize**property**

```
property TraceSize : Word
```

Default: 10000

↳ Determines the number of entries allocated in the trace buffer.

The value may be as large as 4 million. Each entry consumes 2 bytes.

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

See also: Tracing

Tracing**property**

```
property Tracing : TTraceLogState
```

```
TTraceLogState = (tlOff, tlOn, tlDump, tlAppend, tlClear, tlPause);
```

Default: `tlOff`

↳ Determines the current tracing state.

When `Tracing` is set to `tlOff`, as it is by default, no tracing is performed.

To enable tracing, set `Tracing` to `tlOn`. This allocates an internal buffer of $2 * \text{TraceSize}$ bytes and informs the dispatcher to start using this buffer. To disable tracing without writing the contents of the buffer to a disk file, set `Tracing` to `tlOff`. This also frees the internal buffer.

To write the contents of the tracing buffer to disk, set `Tracing` to `tlDump` (which overwrites any existing file named `TraceName`, or creates a new file) or `tlAppend` (which appends to an existing file, or creates a new file). After the component writes to the file it sets `Tracing` to `tlOff`.

Note that `Tracing` is usually not as useful as the dispatcher log. The trace file will contain groupings of transmitted and received characters, which may not be in chronological order. Dispatcher logging will also show most internal state machine states, which tracing does not provide.

To clear the contents of the tracing buffer and continue tracing, set `Tracing` to `tlClear`. After the component clears the buffer, it sets `Tracing` to `tlOn`.

To temporarily pause tracing, set `Tracing` to `tlPause`. To resume, set `Tracing` to `tlOn`.

See “Tracing” on page 33 for more information.

The following example turns on tracing and later dumps the tracing buffer to APRO.TRC:

```
ApdComPort.Tracing := tlOn;
...
ApdComPort.TraceName := 'APRO.TRC';
ApdComPort.Tracing := tlDump;
```

See also: Logging, TraceHex, TraceName, TraceSize

UseEventWord

property


property UseEventWord : Boolean

Default: True

 Determines how the dispatcher checks for received data.

The Windows communication API provides two methods to check for received data and line/modem status changes: API calls and an event word. The event word is maintained by the Windows communications driver. As data is received or line/modem status changes occur, the driver sets bits in the event word. The application can check the bits to determine if any communication events occurred. If so, the application can make the appropriate API call to clear the event word and retrieve the data or the new line/modem status values.

Windows also provides API calls to retrieve the same status information provided by the event word but the API calls are slower. Async Professional uses the event word by default for the fastest possible performance. Unfortunately, there is at least one communication driver (WRPI.DRV, included with some U.S. Robotics modems) that doesn't appear to support the event word. For this and similar drivers, UseEventWord must be set to False before Async Professional will receive data.

 **Caution:** Yielding introduces the possibility of reentrancy, which your application must anticipate and prevent. For example, if WaitForString is called from within a button's OnClick event handler with Yield set to True, the user is able to navigate back to the button and click on it again. Although WaitForString would work in this situation (and would thereby start a wait within a wait) you probably do not want this to happen. It's up to the application to prevent this by disabling the button or the screen containing the button or by checking for reentrancy within the OnClick event handler.

The reentrancy issue also applies to other parts of the application since most applications provide menu options and dialog boxes for changing port parameters, starting file transfers, dialing the modem, and so on. The application must prevent the user from instigating any actions that interfere with WaitForString until WaitForString returns.

None of these problems apply when `Yield` is `False` because `WaitForString` won't allow other message processing while it is waiting. However, you should use this approach only for brief periods of just a few ticks since it prevents Windows from processing other applications and, worse yet, worries your user since the machine appears frozen until `WaitForString` returns.

Note that `WaitForString` uses `GetChar` to retrieve data, which may prevent this data from being seen by any trigger handlers for the same comport component (unless `WaitForString` is called from within a trigger handler itself). However, each received character generates an `OnWaitChar` event, so an event handler can be implemented to pass the data to other processes.

Note that `WaitForString` is depreciated and maintained for backward compatibility. In most cases, other alternatives, such as using the `TApdDataPacket` component or data triggers, are more appropriate for Windows applications. Data triggers avoid the reentrancy problems while still allowing Windows to process messages for other applications and other windows in the current application. Data triggers are more complex to use than `WaitForString` but are well worth the effort in the long run.

The following example shows an `OnWaitChar` event handler that manually stuffs received data into a terminal window:

```
procedure TForm.ApdComPort1WaitChar(CP : TObject; C : Char);
begin
    ApdTerminal1.StuffChar(C);
    ApdTerminal1.ForcePaint;
end;
```

The following example is the `OnClick` event handler from a “Login” button that waits for and responds to “login” and “password” prompts from a remote host:

```
procedure TForm1.LoginClick(Sender : TObject);
begin
    ApdComPort.Output := 'ATDT260-9726'^M;
    if not ApdComPort.WaitForString('login', 1092, True, True) then
        ...handle timeout error
    ApdComPort.Output := 'myname';
    if not ApdComPort.WaitForString(
        'password', 182, True, True) then
        ...handle timeout error
    ApdComPort.Output := 'secret';
    ...
end;
```

See also: `AddDataTrigger`, `OnTriggerData`, `OnWaitChar`, `WaitForMultiString`

XOffChar**property**

```
property XOffChar : Char
```

Default: #19 (^S)

- ↳ Determines the character that is sent to disable remote sending when software flow control is active.

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

See also: SWFlowOptions, XOnChar

XOnChar**property**

```
property XOnChar : Char
```

Default: DefXOnChar (#17, ^Q)


- ↳ Determines the character that is sent to enable remote sending when software flow control is active.

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

Chapter 3: Winsock Components

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

Before a connection can be established, Winsock needs to know how to find the host computer. Each network computer has an address associated with it. This address, called the *IP address*, is a 32-bit value that uniquely identifies the machine. Since a number like 32,147,265 is difficult to remember, network addresses are often displayed in dot notation. Dot notation specifies an IP address as a series of four bytes, each separated by a dot. For example, the TurboPower Web site address can be specified in dot notation as 165.212.210.12. Network software translates the address specified in dot notation to a real 32-bit value.

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

While expressing a network address in dot notation is a little better than dealing with a raw 32-bit value, it is still not particularly easy to remember. For that reason, a global database gives you the capability to specify an IP address in plain text. This database, called the *Domain Name Service (DNS)*, has text entries that correspond to IP address values. For example, the TurboPower Web site DNS entry is `www.turbopower.com`. If Winsock does a lookup for the host name `www.turbopower.com`, it gets the IP address 165.212.210.12.

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

Most software allows you to specify either the host name or the IP address in dot notation when attempting to connect to a server. To illustrate, start your favorite Web browser and type “`www.turbopower.com`” at the address prompt. When you hit Enter, your browser displays the home page of the TurboPower Web site. Now try again, but this time type “`165.212.210.12`” at the address prompt. Once again the browser takes you to the TurboPower Web site.

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

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

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

The telnet protocol describes option negotiation (typically at the beginning of a session) and escaping of certain characters during the entire communication session. This processing is enabled by the WsTelnet property (which is True by default). If the client or server you are communicating with does not support telnet processing, you should set WsTelnet to False prior to opening the port.

Note: If WsTelnet is True, and the client or server to which you are connecting does not support telnet processing, it may appear that your data is being corrupted because telnet processing modifies the data stream.

Sockets in Async Professional

Async Professional includes a device layer, `dlWinsock`, that utilizes Winsock for network and Internet communications.

The Async Professional implementation of Winsock consists of two components. `TApdWinsockPort` is a component that replaces the `TApdComPort` component and can be placed on a form at design time. `TApdWinsockPort` includes properties to allow you to set the network address, the port number, and the mode of the socket (server mode or client mode).

`TApdWinsockPort` is derived from `TApdCustomComPort` and therefore inherits all of its properties and methods. Many of these properties and methods are not applicable to `TApdWinsockPort` when operating in Winsock mode, but are retained in the descendent component so that `TApdWinsockPort` behaves exactly like `TApdComPort` when the `DeviceLayer` property is not set to `dlWinsock`. The properties and methods that do not apply to Winsock operation (e.g., Baud, DataBits, Parity, and StopBits) are simply ignored when `DeviceLayer` is set to `dlWinsock`. Certain Async Professional components (e.g., the modem and TAPI components) are not applicable when using the `dlWinsock` device layer. Faxing over the Internet is not supported because Internet faxing uses a different protocol than faxmodems.

`TApdSocket` is a low-level component that provides access to most standard Winsock services. This component is used internally by Async Professional. A global instance of this component, `ApdSocket`, is created for use by the Winsock device layer in the initialization code of the `AwWinsock` unit. In most cases you won't need to, but you can create your own instance of this class.

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

Winsock NIC selection

Some systems are configured with multiple IP addresses, perhaps a physical network card for local intranet access and a RAS network for Internet access. Some peripherals (i.e., IR ports under Windows 2000) install themselves as a separate network. The `WsLocalAddresses` and `WsLocalAddressIndex` property allow the developer to select which network to use.

Winsock proxy/firewall support

Some systems must go through a firewall/proxy to access remote systems. In this case, the `TApdSocksServerInfo` class (via the `WsSocksServerInfo` property) is used to specify the proxy server to connect through.

TApdSocksServerInfo Class

TApdSocksServerInformation contains the location of the proxy server. If a proxy is to be used with the TApdCustomWinsockPort, these values must be properly configured before opening the connection. When a connection attempt is started (Open property set to true), the TApdCustomWinsockPort will connect to the proxy server and negotiate the connection to the address specified by the WsAddress and WsPort properties. When the connection to the destination is made, the OnWsConnect event will be generated.

If the connection attempt fails, the OnWsError event is generated and the Open property is set to False. The developer can test the ErrorCode parameter of the OnWsError event to determine whether the failure was due to the SOCKS server or due to some other type of failure.

Hierarchy

TPersistent (VCL)

TApdSocksServerInfo (AdWnPort)

Properties

Address

SocksVersion

UserCode

Password

Port

Reference Section

3

Address

run-time property

```
property Address : string
```

↳ Specifies the address of the proxy server.

Password

run-time property

```
property Password : string
```

↳ Specifies the password needed to access the proxy server.

Port

run-time property

```
property Port : Word
```

↳ Specifies the port number of the proxy server.

SocksVersion

run-time property

```
property SocksVersion : TApdSocksVersion
```

```
TApdSocksVersion = (svNone, svSocks4, svSocks5);
```

↳ Specifies the version of the proxy server.

The SocksVersion property can be set to the following values:

Value	Meaning
svNone	No proxy support is required. Opening the port will establish a connection by using the WsAddress and WsPort properties directly.
svSocks4	Specifies Socks4.
svSocks5	Specifies Sock5.

Socks4a has extended Socks4 by adding DNS lookup by the server. Both Socks4 and Socks4a are supported. If SocksVersion is set to svSocks4, Socks4 will be attempted if the WsAddress property contains a dotted-quad IP address, or Socks4a will be used if the WsAddress property contains a domain name.

More information on Socks4 can be found on
<http://www.socks.nec.com/protocol/socks4.protocol> and
<http://www.socks.nec.com/protocol/socks4a.protocol>

More information on Socks5 can be found on <http://www.eborder.nec.com/index2.htm>,
<http://www.socks.nec.com/rfc/rfc1928.txt> and <http://www.socks.nec.com/rfc/rfc1929.txt>.

UserCode

run-time property

```
property UserCode : string
```

↪ Specifies the user name or code needed to access the proxy server.

TApdWinsockPort Component

The TApdWinsockPort component provides a Winsock port that can be used to establish a TCP/IP connection. In addition, it provides all of the services of the standard TApdComPort component. For a description of the properties, events, and methods of TApdComPort, see “TApdComPort Component” on page 22. To put the TApdWinsockPort in Winsock mode, simply set the DeviceLayer property to dlWinsock.

If you use the TApdWinsockPort in Winsock mode, it cannot be used with the TAdModem because in this case Async Professional is not directly controlling a modem. It also cannot be used with TApdSendFax or TApdReceiveFax because in this case Async Professional is not directly communicating with a faxmodem or fax machine.

The TApdWinsockPort component is an implementation of the Winsock version 1.1 API.

Example

This example shows how to connect to the Library of Congress via telnet. Create a new project, add the following components, and set the property values as indicated in Table 3.1.

Table 3.1: Example components and property values

Component	Property	Value
TApdWinsockPort	DeviceLayer	dlWinsock
	WsAddress	locis.loc.gov
	AutoOpen	False
TAdEmulator		
TAdTerminal	Columns	80
	Rows	25
TButton	Caption	Open
TButton	Caption	Close

Double-click on the Open button's OnClick event handler in the Object Inspector and modify the generated method to match this:

```
procedure TForm1.OpenClick(Sender : TObject);
begin
    ApdWinsockPort1.Open := True;
end;
```

Double-click on the Close button's OnClick event handler in the Object Inspector and modify the generated method to match the following code:

```
procedure TForm1.CloseClick(Sender : TObject);
begin
    ApdWinsockPort1.Open := False;
end;
```

Establish a connection to the Internet (e.g., using Windows Dialup Networking).

Compile and run the example. Of course, this is a bare-bones application—but it demonstrates the potential of the TApdWinsockPort.

Hierarchy

TComponent (VCL)

❶ TApdBaseComponent (OOMisc)	8
❷ TApdComPort (AdPort)	22
TApdWinsockPort (AdWnPort)	

Properties

❷ AutoOpen	❷ DSR	Open
❷ BaseAddress	❷ DTR	❷ OutBuffFree
❷ Baud	❷ FlowState	❷ OutBuffUsed
❷ BufferFull	❷ HWFlowOptions	❷ Output
❷ BufferResume	❷ InBuffFree	❷ OutSize
❷ ComHandle	❷ InBuffUsed	❷ Parity
❷ ComNumber	❷ InSize	❷ RI
❷ CTS	❷ LineBreak	❷ RS485Mode
❷ DataBits	❷ LineError	❷ RTS
❷ DCD	❷ LogAllHex	❷ StopBits
❷ DeltaCTS	❷ Logging	❷ SWFlowOptions
❷ DeltaDCD	❷ LogHex	❷ TapiMode
❷ DeltaDSR	❷ LogName	❷ TraceAllHex
❷ DeltaRI	❷ LogSize	❷ TraceHex
DeviceLayer	❷ ModemStatus	❷ TraceName

- ② TraceSize
- ② Tracing
- ② UseEventWord
- ① Version
- WsAddress

- WsLocalAddresses
- WsLocalAddressIndex
- WsMode
- WsPort
- WsSocksServerInfo

- WsTelnet
- ② XOffChar
- ② XOnChar

Methods

- ② ActivateDeviceLayer
- ② AddDataTrigger
- ② AddStatusTrigger
- ② AddTimerTrigger
- ② AddTraceEntry
- ② CharReady
- ② CheckForString
- ② FlushInBuffer
- ② FlushOutBuffer
- ② ForcePortOpen
- ② GetBlock
- ② GetChar
- ② InitPort
- ② PeekBlock
- ② PeekChar
- ② ProcessCommunications
- ② PutBlock
- ② PutChar
- ② PutString
- ② RemoveAllTriggers
- ② RemoveTrigger
- ② SendBreak
- ② SetBreak
- ② SetStatusTrigger
- ② SetTimerTrigger

Events

- ② OnPortClose
- ② OnPortOpen
- ② OnTrigger
- ② OnTriggerAvail
- ② OnTriggerData
- ② OnTriggerLineError
- ② OnTriggerModemStatus
- ② OnTriggerOutbuffFree
- ② OnTriggerOutbuffUsed
- ② OnTriggerOutSent
- ② OnTriggerStatus
- ② OnTriggerTimer
- OnWsAccept
- OnWsConnect
- OnWsDisconnect
- OnWsError

Reference Section

DeviceLayer

property

```
property DeviceLayer : TDeviceLayer  
TDeviceLayer = (dlWin16, dlFossil, dlWin32, dlWinsock);
```

Default: dlWinsock

↳ Determines the hardware interface used by the port.

The DeviceLayer property determines whether the TApdWinsockPort is acting as a Winsock port (dlWinsock) or as a serial port (dlWin32). Since the TApdWinsockPort is a descendent of the TApdCustomComPort, this component can be used almost interchangeably with the TApdComPort component. To switch between Winsock and serial ports, change the DeviceLayer property. DeviceLayer must be dlWinsock to connect via Winsock.

You can create custom device layers by deriving them from TApdBaseDispatcher and creating a new port descendant from TApdCustomComPort where you override ActivateDeviceLayer to return the newly defined device layer.

OnWsAccept

event

```
property OnWsAccept : TWsAcceptEvent  
TWsAcceptEvent = procedure (  
    Sender : TObject; Addr : TInAddr; var Accept : Boolean) of object;
```

↳ Defines an event handler that is called when a client attempts to connect to a server.

This event is generated when an application is acting as a server (WsMode equals WsServer) and a client application attempts a connection. Addr is the network address of the client. To accept the connection, set Accept to True. To refuse the connection, set Accept to False. OnWsAccept is not generated when an application is acting as a client.

The following example calls a user-supplied function named `GoodAddress` to determine whether the network address is one for which connection will be accepted. If `GoodAddress` returns `True`, `Accept` is set to `True` and the connection is made. If `GoodAddress` returns `False`, `Accept` is set to `False` and the connection is refused.

```
procedure TForm1.WsPortWsAccept(
  Sender : TObject; Addr : TInAddr; var Accept : Boolean);
begin
  if GoodAddress(Addr) then begin
    Status.Caption := 'Accepted!';
    Accept := True;
  end else begin
    Status.Caption := 'Connection Denied';
    Accept := False;
  end;
end;
```

See also: `OnWsConnect`, `WsMode`

OnWsConnect

event

```
property OnWsConnect : TNotifyEvent
```

↳ Defines an event handler that is called when a Winsock connection is established.

When an application is operating as a client (`WsMode` equals `WsClient`), it usually attempts to connect to a server. This event is generated when the server accepts the connection. This event is not generated when an application is acting as a server.

The following example illustrates a client application receiving notification that a connection to the server was established and accepted by the server:

```
procedure TForm1.WsPortWsConnect(Sender : TObject);
begin
  Status.Caption := 'Connected';
  { do some processing... }
end;
```

See also: `OnWsAccept`, `OnWsDisconnect`, `WsMode`

```
property OnWsDisconnect : TNotifyEvent
```

↳ Defines an event handler that is called when a Winsock connection is dropped.

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

If WsMode equals WsServer, OnDisconnect is generated when the client is disconnected. The Open property stays True and the TApdWinsockPort continues to listen for other clients attempting to connect. If WsMode equals WsClient, OnDisconnect is generated when the connection is lost. Async Professional then sets the Open property to False.

The following example illustrates a server application receiving notification that the client has disconnected:

```
procedure TForm1.WsPortWsDisconnect(Sender : TObject);  
begin  
    Status.Caption := 'Bye!';  
end;
```

See also: OnWsConnect, Open, WsMode

OnWsError**event**

```
property OnWsError : TWSErrorEvent
```

```
TWSErrorEvent = procedure(  
    Sender : TObject; ErrorCode : Integer) of object;
```

↳ Defines an event handler that is generated when a Winsock error occurs.

ErrorCode contains the Winsock error code. See “Error Handling and Exception Classes” on page 900 for a list of error codes.

```
property Open : Boolean
```

Default: False

3

↪ Determines whether the Winsock port is open and initialized.

Open must be set to True before a Winsock port can send or receive characters. When Open is set to True, the TApdWinsockPort uses all current property settings to allocate input and output buffers, create a socket, open the Winsock port, and enable or disable tracing and logging. It then registers a low-level trigger handler, which gets the first look at all trigger events and passes control on to the appropriate OnTriggerXxx event handlers.

When Open is set to False, the TApdWinsockPort sets the tracing and logging properties to tIDump (which creates output files if any information was buffered), closes the Winsock port, and deallocates input and output buffers.

There is no harm done by setting Open to True when it is already True, or setting it to False when it is already False. If WsMode equals WsServer and you set Open to True, a socket is created and it listens at the port designated by the WsPort property. If WsMode equals WsClient and you set Open to True, the component attempts to connect to a server at the designated WsAddress and WsPort.

WsAddress

property

```
property WsAddress : string
```

↪ The network address used to make a Winsock connection.

WsAddress accepts the IP address in dot notation (165.212.210.10) or as a host name (telnet.turbopower.com). If a host name is used, Async Professional does a DNS lookup to determine whether a DNS entry exists for the host name. If an IP address can be found, the port is opened. If an IP address cannot be found, a EApdSocketException is raised.

⚠ **Caution:** Do not add leading zeros in dot notation addresses (e.g., 165.212.210.010). Leading zeros will cause the number to be interpreted as an octal value.

WsLocalAddresses**read-only, run-time property**

```
property WsLocalAddresses : TStringList
```

- ↳ Lists the IP addresses for each network interface that is installed.

WsLocalAddresses is a read-only TStringList containing the IP address for each network interface installed on the computer. This property is populated when the TApdCustomWinsockPort is created. The network interface to use is specified by the WsLocalAddressIndex property.

See also: WsLocalAddressIndex

WsLocalAddressIndex**run-time property**

```
property WsLocalAddressIndex : Integer
```

- ↳ Determines the network interface to use.

To select a network interface, set WsLocalAddressIndex to the index of the network interface listed in the WsLocalAddresses property.

See also: WsLocalAddresses

WsMode**property**

```
property WsMode : TWsMode
TWsMode = (wsClient, wsServer);
```

Default: WsClient

- ↳ Determines whether the application operates as a server or a client.

If WsMode is WsServer, the application acts as a server. When the Open property is set to True, the application begins listening for possible connections on the port specified by WsPort.

If WsMode is WsClient, the application operates as a client. When the Open property is set to True, the client attempts to connect to the server at the address specified by WsAddress. When Open is set to False, the client disconnects from the server and the socket is closed.

See also: Open, WsAddress, WsMode, WsPort

WsPort**property**

```
property WsPort : string
```

Default: "telnet"

↳ The Winsock port used to establish a network connection.

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

See also: WsAddress

WsSocksServerInfo**run-time property**

```
property WsSocksServerInfo : TApdSocksServerInfo
```

↳ Contains the Firewall/Proxy configuration.

WsSocksServerInfo contains the configuration of the proxy server and the type of proxy server in use. If a proxy server is in use, this must be properly configured before opening the port.

WsTelnet**property**

```
property WsTelnet : Boolean
```

Default: True

↳ Indicates whether telnet processing is enabled.

For most uses of the TApdWinsockPort (such as connecting to telnet servers or communication between TApdWinsockPort components) the default value of True is appropriate. However, if you communicate with a server or client that does not support telnet processing you should set WsTelnet to False.

WsTelnet cannot be changed while the port is open. The value of WsTelnet when the port is opened is used by the device layer for the duration of that communication session.

TApdSocket Component

The TApdSocket component is a low-level class that provides many standard Winsock services. It is essentially a thin wrapper around the Winsock API and transparently handles tasks such as loading, starting, and shutting down Winsock. It is used internally by the Async Professional Winsock device layer and the TApdWinsockPort component. In most cases you won't need to make use of the TApdSocket component directly, but it is documented here in case you do.

A global instance of the TApdSocket class (ApdSocket) is created in the initialization code of the Winsock device layer unit (AwWnsock), and is available for use in your application. To use its services, simply add AwWnsock to your unit's uses clause. Because it is a low-level class, the Winsock services it provides access to are not documented in detail here. There are many Winsock references available to consult. The following were useful in the development of TApdSocket:

- Microsoft Winsock API help file
- Microsoft Developer Network CD
- Dumas, *Programming Winsock*, Sam's Publishing, ISBN 0-672-30594-1
- Quinn and Shute, *Windows Sockets Network Programming*, Addison-Wesley, ISBN 0-201-63372-8
- Roberts, *Developing for the Internet with Winsock*, Coriolis Group Books, ISBN 1-883577-42-X
- Chapman, *Building Internet Applications with Delphi 2*, Que, ISBN 0-7897-0732-2

Hierarchy

TComponent (VCL)	
❶ TAPdBaseComponent (OOMisc)	8
TAPdSocket (AdSocket)	

Properties

Description	LocalAddress	❶ Version
Handle	LocalHost	WSVersion
HighVersion	MaxSockets	
LastError	SystemStatus	

Methods

AcceptSocket	htons	ntohl
BindSocket	ListenSocket	ntohs
CheckLoaded	LookupAddress	ReadSocket
CloseSocket	LookupName	SetAsyncStyles
ConnectSocket	LookupPort	String2NetAddr
CreateSocket	LookupService	WriteSocket
htonl	NetAddr2String	

Events

OnWsAccept	OnWsDisconnect	OnWsRead
OnWsConnect	OnWsError	OnWsWrite

Reference Section

AcceptSocket

method

```
function AcceptSocket(  
    Socket : TSocket; var Address : TSockAddrIn) : TSocket;  
  
TSockAddrIn = packed record  
    case Integer of  
        0: (sin_family : Word;  
            sin_port    : Word;  
            sin_addr    : TInAddr;  
            sin_zero    : array[0..7] of AnsiChar);  
        1: (sa_family  : Word;  
            sa_data     : array[0..13] of AnsiChar)  
    end;
```

↳ Accepts a client that is trying to attach to a listening socket.

Socket is the handle of the socket. Address is a structure that contains information used by Winsock.

See also: ListenSocket

BindSocket

method

```
function BindSocket(  
    Socket : TSocket; Address : TSockAddrIn) : Integer;  
  
TSockAddrIn = packed record  
    case Integer of  
        0: (sin_family : Word;  
            sin_port    : Word;  
            sin_addr    : TInAddr;  
            sin_zero    : array[0..7] of AnsiChar);  
        1: (sa_family  : Word;  
            sa_data     : array[0..13] of AnsiChar)  
    end;
```

↳ Associates a local network address and port number with a socket.

Socket is the handle of the socket. Address is a structure that contains information used by Winsock. Typically a server application calls BindSocket with a specified address and port number prior to calling ListenSocket. A client application does not typically bind to a specific address and port.

See also: CreateSocket, ListenSocket

CheckLoaded**method**

```
procedure CheckLoaded;
```

↳ Determines whether the Winsock DLL is loaded and initialized.

Call `CheckLoaded` to see if Winsock is ready for use. If Winsock is not initialized, `CheckLoaded` raises an `EApdSocketException`. This exception can be caught in your application and responded to accordingly.

CloseSocket**method**

```
function CloseSocket(Socket : TSocket) : Integer;
```

↳ Closes a socket.

`CloseSocket` closes a socket and frees the memory allocated for it. `Socket` is the handle of the socket to close.

See also: `CreateSocket`

ConnectSocket**method**

```
function ConnectSocket(
    Socket : TSocket; Address : TSocketAddrIn) : Integer;
```

```
TSocketAddrIn = packed record
    case Integer of
        0: (sin_family : Word;
            sin_port    : Word;
            sin_addr    : TInAddr;
            sin_zero    : array[0..7] of AnsiChar);
        1: (sa_family  : Word;
            sa_data     : array[0..13] of AnsiChar)
    end;
```

↳ Establishes a network connection.

`ConnectSocket` is used by a client to connect the socket specified by `Socket` to a remote host. `Address` is a structure that contains information used by Winsock to find the remote host.

See also: `BindSocket`, `CreateSocket`

CreateSocket**method**

```
function CreateSocket : TSocket;
```

↳ Creates a socket.

If the socket is created successfully, `CreateSocket` returns a unique socket descriptor that is used to refer to this socket in subsequent Winsock operations. If the socket cannot be created, `CreateSocket` raises an `EApdSocketException`.

See also: `BindSocket`, `CloseSocket`

Description**read-only, run-time property**

```
property Description : string
```

↳ Contains a string that describes the Winsock DLL.

`Description` is a read-only property that contains a textual description of the current Winsock DLL. Since there are so many different Winsock DLLs, the string returned depends on the Winsock vendor. For the standard Windows NT 4.0/2000 Winsock DLL, the string is "Winsock 2.0." For the standard Windows 95/98/ME Winsock DLL, the string is "Microsoft Windows Sockets version 1.1."

See also: `HighVersion`, `SystemStatus`, `Version`

Handle**read-only, run-time property**

```
property Handle : HWND
```

↳ The window handle for the `TApdSocket` class.

Winsock uses the window handle to send messages to the `TApdSocket` object. The Winsock messages received by the object generate the `OnWsAccept`, `OnWsConnect`, `OnWsDisconnect`, `OnWsError`, `OnWsRead`, and `OnWsWrite` events.

HighVersion**read-only, run-time property**

```
property HighVersion : Word
```

↳ Contains the highest version of the Winsock specification supported by the current Winsock DLL.

For the Windows NT 4.0 and 2000 Winsock, `HighVersion` is 2.2, which indicates that it can support version 2.2 of the Winsock specification. For the Windows 95/98/ME Winsock, `HighVersion` is 1.1.

See also: `WSVersion`

htonl**method**

```
function htonl(HostLong : LongInt) : LongInt;
```

↳ Translates a 32-bit value from host byte order to network byte order.

IBM-compatible computers typically store data in memory in little-endian byte order (the least significant byte stored first followed by the most significant byte) or host byte order. TCP/IP stipulates that data should be sent in big-endian byte order (most significant byte followed by least significant byte) or network byte order.

See also: htons, ntohl, ntohs

htons**method**

```
function htons(HostShort : Word) : Word;
```

↳ Translates a 16-bit value from host byte order to network byte order.

IBM-compatible computers typically store data in memory in little-endian byte order (the least significant byte stored first followed by the most significant byte) or host byte order. TCP/IP stipulates that data should be sent in big-endian byte order (most significant byte followed by least significant byte) or network byte order.

See also: htonl, ntohl, ntohs

LastError**read-only, run-time property**

```
property LastError : Integer;
```

↳ Contains the error code of the last Winsock error.

If a Winsock operation fails, you can use LastError to get the Winsock error code. See “Error Handling and Exception Classes” on page 900 for a list of the error codes.

See also: OnWsError

ListenSocket**method**

```
function ListenSocket(
    Socket : TSocket; Backlog : Integer) : Integer;
```

↳ Tells a socket to listen for a connection attempt.

ListenSocket is used by a server application to enter listening mode. Socket is the socket on which to listen. Backlog is the maximum length of the queue for waiting connection attempts. If ListenSocket is successful, 0 is returned. If ListenSocket is not successful, it raises the EApdSocketException.

See also: BindSocket, CreateSocket

LocalAddress**read-only, run-time property**

```
property LocalAddress : string
```

↳ Contains the local machine's network address.

LocalAddress contains a text string of the local machine's network address in dot notation (e.g., "165.212.210.12").

See also: LocalHost

LocalHost**read-only, run-time property**

```
property LocalHost : string
```

↳ Contains the local machine's network name.

LocalHost contains a textual description of the local machine's network name (e.g., "garyf-testmachine").

See also: LocalAddress

LookupAddress**method**

```
function LookupAddress(InAddr : TInAddr) : string;

TInAddr = packed record
  case Integer of
    0 : (S_un_b : SunB);
    1 : (S_un_w : SunW);
    2 : (S_addr : LongInt);
  end;
```

↳ Gets a host name for the Internet address specified by InAddr.

The following example uses String2NetAddr to fill in a TInAddr structure from a text string containing an Internet address. It then calls LookupAddress to get the host name for the address.

```
var
  MyAddr : TInAddr;
with TAPdSocket.Create(self) do try
  MyAddr := String2NetAddr('165.212.210.12');
  HostLabel.Caption := LookupAddress(MyAddr);
finally
  Free;
end;
```

See also: LookupName, String2NetAddr

LookupName**method**

```
function LookupName(const Name : string) : TInAddr;

TInAddr = packed record
  case Integer of
    0 : (S_un_b : SunB);
    1 : (S_un_w : SunW);
    2 : (S_addr : LongInt);
  end;
```

↳ LookupName gets an Internet address for the host name specified by Name.

The Internet address is returned as a TInAddr structure.

The following example gets an Internet address from the host name “www.turbopower.com” and then uses the NetAddr2String method to display the address in a label:

```
var
  MyAddr : TInAddr;
with TAPdSocket.Create(Self) do try
  MyAddr := LookupName('www.turbopower.com');
  AddressLabel.Caption := NetAddr2String(MyAddr);
finally
  Free;
end;
```

See also: LookupHost, NetAddr2String

LookupPort

method

```
function LookupPort(Port : Integer) : string;
```

↳ Gets a text string of the service name for the port specified by Port.

There are certain well-known ports used in Winsock. For example, port 25 is typically used for SMTP (mail), port 23 is used for telnet, and port 119 is used for NNTP (news):

```
with TAPdSocket.Create(Self) do try
  ServiceLabel.Caption := LookupPort(25);
finally
  Free;
end;
```

See also: LookupService

LookupService**method**

```
function LookupService(const Service : string) : Integer;
```

↳ Gets the port number for the service name specified by Service.

The service name should be one of the Winsock well-known services (such as “SMTP”). If the service cannot be found, LookupService returns an empty string.

```
var
  MyPort : Integer;
with TApdSocket.Create(Self) do try
  MyPort := LookupService('smtp');
finally
  Free;
end;
```

See also: LookupPort

MaxSockets**read-only, run-time property**

```
property MaxSockets : Word
```

↳ The maximum number of sockets available for the current version of Winsock.

NetAddr2String**method**

```
function NetAddr2String(InAddr : TInAddr) : string;
```

```
TInAddr = packed record
  case Integer of
    0 : (S_un_b : SunB);
    1 : (S_un_w : SunW);
    2 : (S_addr : LongInt);
  end;
```

↳ Translates the 32-bit network address in InAddr to a string.

The string is in dot notation (e.g., “165.212.210.12”).

The following example converts an Internet address to a string. The string is then displayed in a label component.

```
var
  MyAddr : TInAddr;
with TAPdSocket.Create(Self) do try
  MyAddr := LookupName('www.turbopower.com');
  AddressLabel.Caption := NetAddr2String(MyAddr);
finally
  Free;
end;
```

See also: String2NetAddr

ntohl

method

```
function ntohl(NetLong : LongInt) : LongInt;
```

↪ Translates a 32-bit value from network byte order to host byte order.

IBM-compatible computers typically store data in memory in little-endian byte order (the least significant byte stored first followed by the most significant byte) or host byte order. TCP/IP stipulates that data should be sent in big-endian byte order (most significant byte followed by least significant byte) or network byte order.

See also: htonl, htons, ntohs

ntohs

method

```
function ntohs(NetShort : Word) : Word;
```

↪ Translates a 16-bit value from network byte order to host byte order.

IBM-compatible computers typically store data in memory in little-endian byte order (the least significant byte stored first followed by the most significant byte) or host byte order. TCP/IP stipulates that data should be sent in big-endian byte order (most significant byte followed by least significant byte) or network byte order.

See also: htonl, htons, ntohl

OnWsAccept**event**

```
property OnWsAccept : TWsNotifyEvent

TWsNotifyEvent = procedure(
    Sender: TObject; Socket: TSocket) of object;
```

↳ Defines an event handler that is called when the server accepts a connection.

This event is primarily used when an application is operating as a server. The server application listens on a specific port for possible connections. When a client socket tries to connect, the OnWsAccept event is generated.

See also: OnWsConnect

OnWsConnect**event**

```
property OnWsConnect : TWsNotifyEvent

TWsNotifyEvent = procedure(
    Sender: TObject; Socket: TSocket) of object;
```

↳ Defines an event handler that is called when a Winsock connection is established.

When a server application accepts a connection, the OnWsConnect event is generated to notify the client application.

See also: OnWsDisconnect

OnWsDisconnect**event**

```
property OnWsDisconnect : TWsNotifyEvent

TWsNotifyEvent = procedure(
    Sender: TObject; Socket: TSocket) of object;
```

↳ Defines an event handler that is called when a Winsock connection is dropped.

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

See also: OnWsConnect

OnWsError**event**

```
property OnWsError : TWsSocketErrorEvent

TWsSocketErrorEvent = procedure(Sender : TObject;
    Socket : TSocket; ErrorCode : Integer) of object;
```

↳ Defines an event handler that is called when a Winsock error occurs.

Socket identifies the socket for which the error occurred. ErrorCode contains the Winsock error code. See “Error Handling and Exception Classes” on page 900 for a list of the error codes.

See Also: LastError

OnWsRead**event**

```
property OnWsRead : TWsNotifyEvent

TWsNotifyEvent = procedure(
    Sender : TObject; Socket : TSocket) of object;
```

↳ Defines an event handler that is called when data is available to be read on a socket.

See also: OnWsWrite

OnWsWrite**event**

```
property OnWsWrite : TWsNotifyEvent

TWsNotifyEvent = procedure(
    Sender : TObject; Socket : TSocket) of object;
```

↳ Defines an event handler that is called when Winsock can accept more data from a socket.

See also: OnWsRead

ReadSocket**method**

```
function ReadSocket(
    Socket : TSocket; var Buf; BufSize, Flags : Integer) : Integer;
```

↳ Reads data from a socket.

Socket is the socket from which to receive data. Buf is the buffer where the data is stored. BufSize is the size of Buf in bytes. Flags determines how the receive operates. Set Flags to zero for normal operation, or see your Winsock documentation for other possible values. ReadSocket returns the number of bytes read.

See also: WriteSocket

```
function SetAsyncStyles(
    Socket : TSocket; lEvent : LongInt) : Integer;
```

↳ Tells Winsock to send notification of certain network events.

3

Socket is the socket for which events should be reported. lEvent is the event or events that should be reported. The notification event constants for which you can receive notification are FD_READ, FD_WRITE, FD_CONNECT, and FD_ACCEPT. See the WSAAsyncSelect function in your Winsock documentation for more information.

When an event that notification is requested for occurs, Winsock sends a CM_APDSOCKETMESSAGE message to the TApdSocket class. The low word of lParam is the network event that occurred and the high word is an error code if an error occurred.

String2NetAddr

method

```
function String2NetAddr(const S : string) : TInAddr;

TInAddr = packed record
    case Integer of
        0 : (S_un_b : SunB);
        1 : (S_un_w : SunW);
        2 : (S_addr : LongInt);
    end;
```

↳ Translates a string to a network address.

String2NetAddr translates S into a 32-bit value in network byte order. It is returned in the form of a TInAddr structure. S should be in dot notation (e.g., “165.212.210.12”).

The following example creates a socket and then turns the string address “165.212.210.12” into an network address. The network address could then be used to connect to a server.

```
var
    MyAddr : TInAddr;
with TApdSocket.Create(Self) do try
    MyAddr := String2NetAddr('165.212.210.12');
finally
    Free;
end;
```

See also: Net2StringAddr

SystemStatus**read-only, run-time property**

```
property SystemStatus : string
```

↳ Contains the current status of the Winsock DLL.

SystemStatus usually returns “Running under” Windows 95/98/ME or Windows NT 4.0/2000.

See also: Description

WriteSocket**method**

```
function WriteSocket(  
    Socket : TSocket; var Buf; BufSize, Flags : Integer) : Integer;
```

↳ Sends data to a socket.

Socket is the socket on which to send data. Buf is the buffer that contains the data. BufSize is the size of Buf in bytes. Flags determines the send method. Set Flags to zero for normal operation or see your Winsock documentation for other possible values.

WriteSocket does not send the data directly to the receiving end. Winsock queues the data and sends it when possible. The return value from WriteSocket is the number of bytes queued for transmission.

See also: ReadSocket

WsVersion**read-only, run-time property**

```
property WsVersion : Word
```

↳ Contains the version number of the current Winsock DLL.

WsVersion is a 16-bit value. The high-order byte contains the major version number and the low-order byte contains the minor version number.

The following example gets the version number, translates it into a text string, and displays it in a label component:

```
var
  MyVer : Word;
with TApdSocket.Create(Self) do try
  MyVer := WsVersion;
  VerLabel.Caption := Format('%d.%d', [LoByte(MyVer),
HiByte(MyVer)]);
finally
  Free;
end;
```

See also: [HighVersion](#)

Chapter 4: Data Packet Component

The purpose of the data packet component is to provide a simple solution to the common task of looking for a particular sequence of bytes in the incoming data. Data packet components collect data that has certain properties and pass that data as a complete unit to the client application.

TApdDataPacket Component

The TApdDataPacket component provides automatic data packet delivery from the incoming data stream based on simple properties set in the component.

A data packet can be thought of as an advanced data trigger. Packets automatically collect data from the incoming data stream based on criteria specified in the properties of the data packet component, and deliver the data when the criteria have been met. As opposed to traditional data triggers, data packets do their own buffering. This means that data packets do not have the same limitation as data triggers (that data may no longer be available in the input buffer for processing when the data trigger fires).

You would typically use data packets in place of data triggers when the data you are looking for has a fixed length or starts or ends with a known string of data. These conditions can be set in the data packet component at design time or run time.

Data as packets

Most data arriving at the serial port can be described as a packet. It will have a start condition (something that defines the beginning of the data) and an end condition (something that defines the end of the data). The TApdDataPacket component supports start conditions of a character or characters, or any data. The TApdDataPacket component supports end conditions of a character or characters, timeout, or a specific number of characters. The simplest data packet is a single string, such as “hello”, and can progress through more complicated packets that define other packets within the collected data. In broad strokes, packets can be categorized as follows:

- A specific character/string: The StartString defines the packet in its entirety. Set StartCond to scString (the default) and StartString to the string to detect:

```
...
ApdDataPacket1.StartCond := scString;
ApdDataPacket1.StartString := 'hello';
...
```

- A bracketed packet: This is the most common usage, where the beginning and ending of the data is defined by known characters/strings. For example, you may be expecting data starting with an <STX> (#2) character, followed by a chunk of data,

terminated by an <ETX> (#3) character. Set StartCond to scString, StartString to the “start of packet” character/string, EndCond to [ecString] and EndString to the “end of packet” character/string:

```
...
ApdDataPacket1.StartCond := scString;
ApdDataPacket1.StartString := #2; // STX char
ApdDataPacket1.EndCond := [ecString];
ApdDataPacket1.EndString := #3; // ETX char
...
```

Note: C++Builder uses a more difficult implementation of sets than Delphi does, use the following syntax to set the EndCond property:

```
...
ApdDataPacket1->EndCond.Clear();
ApdDataPacket1->EndCond << ecString;
...
```

- A known start character/string followed by data of known length: An example might be an <STX> (#2) followed by 18 characters without a terminating character/string. To detect this packet, set StartCond to scString, StartString to the “start of packet” character/string, EndCond to [ecPacketSize] and PacketSize to the length of the data. If IncludeStrings is True, add the length of your StartString to PacketSize.

```
...
ApdDataPacket1.StartCond := scString
ApdDataPacket1.StartString := #2; // STX char
ApdDataPacket1.EndCond := [ecPacketSize];
ApdDataPacket1.IncludeStrings := True;
ApdDataPacket1.PacketLength := 19; // 18 data chars,
    1 start char
...
```

- A known number of data chars, with a terminating character/string: An example of this type of packet could be a log in sequence, where you would prompt for a user name and want to collect everything up to a <CR>.

```
...
ApdDataPacket1.StartCond := scAnyData;
ApdDataPacket1.EndCond := [ecString];
ApdDataPacket1.EndString := #13; // CR char
...
```

These are only a few of the possibilities. Your data may vary. You may have a start string, followed by a character indicating the length of the data (use two data packets, one to collect the start string and length char, the other to capture the next “length” number of chars), or

you may have something that needs a more liberal packet interpretation. If you can conceptualize the expansibility of the packet format, you can usually work something up that works for your conditions.

Data ownership

There is no limit on the number of data packet components for a port, however, any incoming character can be part of only one data packet. The first enabled data packet that has its start condition met takes ownership of all incoming data until the packet is complete. If a data packet times out, the data it has collected up to that point is made available to any other enabled data packets for the port.

The TApdDataPacket component has a component editor, shown in Figure 4.1, where all properties can conveniently be set at once. You can invoke it by right clicking on the context-menu of the component.

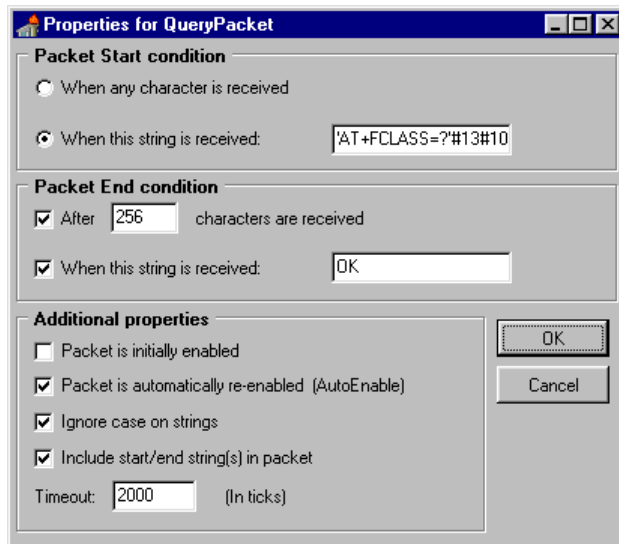


Figure 4.1: TApdDataPacket component editor.

Packet Start Condition

The Packet Start Condition defines the start of the packet. You have the option to start the packet as soon as any data is received or you can start data collection when a particular string is received.

Refer to the StartCond and StartString properties in the reference section for more information on starting a packet.

Packet End Condition

The Packet End Condition defines when the packet is complete. Packet completion can either occur after a certain number of characters have been received, or when a particular string is received to terminate the packet. If both types of conditions are defined, the first condition met will cause the packet delivery event to fire.

Refer to the `EndCond`, `EndString` and `PacketSize` properties in the reference section for more information on terminating a packet.

Additional properties

The additional properties define details about how the packet should operate: Whether it should be initially enabled, whether it should automatically re-enable after having been received (the default is that it re-enables itself), whether case should be ignored on the start and end strings, whether or not the start and end strings should be included in the packet delivered in the `OnPacket` events and whether the packet collection logic can time out for this packet and what the time-out period should be.

Refer to the `AutoEnable`, `IgnoreCase`, `IncludeStrings` and `TimeOut` properties in the reference section for more information on these additional properties.

Non-printable characters and wildcards in the packet

The `TApdDataPacket` supports some translations in the `StartString` and `EndString` properties. These include support for non-printable characters (control chars) and wildcards. To specify a control character, you can use either caret (^) or decimal notation, or you can enter the literal char at run time. For example, to add the <ACK> character, you can enter “^F” or “#6” at design time, or just #6 at run time. Since a caret (^) and pound (#) are control character escapes, they must be enclosed in quotes to detect a literal ‘^’ in the data stream.

Wildcards in the packet definition can be very useful. The wildcard character is a single question mark (?). This is interpreted as any single character. Since a single ‘?’ is now considered a wildcard, use “\?” to detect a literal ‘?’ in the data stream.

For example, a relatively common packet will contain a block of data terminated by a character that is followed by a checksum character. This can be captured by setting the `EndString` to the terminating character and a ‘?’:

```
...
ApdDataPacket1.EndString := #3 + '?'; // ETX and the next char
...
```

If `IncludeStrings` is `True`, the last char in the collected data will be the checksum.

Example

This example demonstrates the use of a `TApdDataPacket` component to retrieve a modem's response to the `ATI3` command. Create a new project, add the following components, and set the property values as indicated in Table 4.1.

Table 4.1: *Example components and property settings*

Component	Property	Value
TApdComport		
TApdDataPacket	StartString	'ATI3' #13
	EndString	'OK' #13
	EndCond	[ecString]
	IncludeStrings	False
TButton		

Double-click on the button's `OnClick` event handler in the Object Inspector and modify the generated source code to match this:

```
procedure TForm1.Button1Click(Sender : TObject);
begin
    ApdComPort1.PutString('ATI3' #13);
end;
```

Double-click on the `TApdDataPacket` component `OnStringPacket` event handler in the Object Inspector and modify the generated source code to match this:

```
procedure TForm1.ApdDataPacket1StringPacket(
    Sender : TObject; Data : String);
begin
    Caption := trim(Data);
end;
```

Compile and run the application. When prompted, select a serial port that has a modem attached. When you press the button, you should see the caption change to the data set name (the response to `ATI3`) reported by the modem.

Hierarchy

TComponent (VCL)

❶ TApdBaseComponent (OOMisc)	8
TApdDataPacket (AdPacket)	

Properties

AutoEnable	EndString	StartCond
ComPort	IgnoreCase	StartString
Enabled	IncludeStrings	TimeOut
EndCond	PacketSize	❶ Version

Events

OnPacket	OnTimeout	OnStringPacket
----------	-----------	----------------

Reference Section

AutoEnable

property

```
property AutoEnable : Boolean
```

Default: True

↳ Determines whether a data packet is automatically re-enabled.

AutoEnable controls what happens after the packet is received (the start string and end condition for the packet are met in the data stream). If AutoEnable is True, the data packet is enabled again and TApdDataPacket starts watching for the start string again. If AutoEnable is False, the data packet is disabled.

ComPort

property

```
property ComPort : TApdCustomComPort
```

↳ Determines the port used by the data packet.

Enabled

property

```
property Enabled : Boolean
```

Default: True

↳ Determines whether a packet is allowed to collect data.

You can have as many TApdDataPacket components as you like. You can conveniently turn them on or off using the Enabled property. Changing the Enabled property is allowed from within event handlers. This lets you chain packets together. See the QRYMDM example program for an example of how to do this.

```
property EndCond : TPacketEndSet
TPacketEndSet = set of TPacketEndCond;
TPacketEndCond = (ecString, ecPacketSize);
```

Default: []

↪ Determines when a complete packet has been received.

The valid values for EndCond are:

Value	Result
[]	The string specified by StartString is. considered to be the entire packet (this is equivalent to a traditional data trigger).
[EcString]	The packet ends when the string specified by EndString is received.
[EcPacketSize]	The packet ends when the number of characters specified by PacketSize is received.
[EcString and ecPacketSize]	The packet ends when either the string specified by EndString is received or the number of characters specified by PacketSize is received.

Note: C++Builder uses a more difficult implementation of sets than Delphi does, use the following syntax to set the EndCond property:

```
...
  ApdDataPacket1->EndCond.Clear();
  ApdDataPacket1->EndCond << ecString;
...
```

See also: EndString, PacketSize, StartString

EndString**property**`property EndString : string`

↪ The string that completes a data packet.

If `EndCond` contains `ecString`, the packet stops collecting data when the string specified by `EndString` is received.

See the `StartString` property on page 143 for information about specifying characters and using fixed-length wildcards in the string.

See also: `EndCond`, `StartString`

IgnoreCase**property**`property IgnoreCase : Boolean`

Default: `True`

↪ Determines whether the `StartString` and `EndString` properties are case-sensitive.

See also: `EndString`, `StartString`

IncludeStrings**property**`property IncludeStrings : Boolean`

Default: `True`

↪ Determines whether the strings that define a packet are made available to the event handler.

For example, assume that `StartString` is “Async”, `EndString` is “al”, and the string “Async Professional” arrives at the port. If `IncludeStrings` is `True`, the packet will contain “Async Professional.” If `IncludeStrings` is `False`, the packet will contain “Profession.”

See also: `EndString`, `StartString`

```
property OnPacket : TPacketNotifyEvent

TPacketNotifyEvent = procedure(
    Sender : TObject; Data : pointer; Size : Integer) of object;
```

↳ Defines an event handler that is called when a complete data packet is available.

Data is a pointer to the actual collected data. Size is the length of the collected data. The data at Data is only valid for the duration of this event. Since Data is temporary, you can move the collected data into your own buffer for storage or further processing outside of this event. The following snippet demonstrates one technique of doing this:

```
var
    Buffer: array[0..255] of byte;

procedure TForm1.ApDataPacket1Packet(
    Sender: TObject; Data: Pointer; Size: Integer);
begin
    Move(Data^, Buffer[0], Size);
end;

void __fastcall TForm1::ApDataPacket1Packet(
    TObject *Sender, Pointer Data, int Size)
{
    char* MyData = new char[Size];
    Move(Data, MyData, Size);
}
```

```
property OnTimeout : TNotifyEvent
```

↳ Defines an event handler that is called when a timeout occurs during receipt of a packet.

The OnTimeout event is generated when a packet is in data collection mode but hasn't completed within the number of ticks specified by TimeOut. By default, packets are disabled when they time out, but they can be re-enabled from within the event handler if desired.

The data collected up to the point of the timeout is available through the GetCollectedString and GetCollectedData methods.

The timeout timer does not start until the start condition has been met. If StartCond = scString, the timer starts once the string defined by StartString has been received. If StartCond = scAnyData, the timer starts once the data packet has been enabled. If you need to start the timer once the packet starts actually collecting data, set the StartCond to scString and StartString to the wildcard char ('?').


```
property OnStringPacket : TStringPacketNotifyEvent  
TStringPacketNotifyEvent = procedure(  
    Sender : TObject; Data : string) of object;
```

↳ Defines an event handler that is called when a complete data packet is available.

Data is the actual data in the packet. The data packet is only available for the duration of the event.

Note that a null character (#0) in the collected data may terminate the Data string prematurely. This is due to the way that Delphi and C++Builder implement huge strings. If you are expecting null characters in the collected data, use the OnPacket event instead.

See also: OnPacket

PacketSize**property**

```
property PacketSize : Integer  
Default: 0
```

↳ Determines the size of a packet.

If EndCond contains ecCharCount, PacketSize determines the size of the data packet.

If IncludeStrings is True, PacketSize will not compensate for the length of the start and end strings. For example, assume that the StartString is “Async”, PacketSize is 13 and the string “Async Professional” arrives at the port. If IncludeStrings is True, the collected data will contain “Async Profess” (13 characters); if IncludeStrings is False, the collected data will contain “Professional.”

See also: EndCond

```
property StartCond : TPacketStartCond
TPacketStartCond = (scString, scAnyData);
```

Default: scString

↳ Determines when a packet should start collecting data.

The valid values for StartCond are:

Value	Result
scString	The packet starts collecting data when the string specified by StartString is received.
scAnyData	The packet starts collecting data as soon as data is available in the input queue.

See also: StartString

```
property StartString : string
```

↳ The string that causes a packet to start collecting data.

If StartCond is scString, the packet starts collecting data when the string specified by StartString is received.

To specify a control character in the string, use a caret '^' symbol (e.g. ^L^M). To specify a character with ordinal value greater than 127, use the #nnn notation, where nnn is an integer in the range 128 to 255. Since '^' and '#' are used in this special way as escape characters, if you want a '^' or '#' as a printable character in the string, it must be enclosed in quotes. To mix printable and non-printable characters in a string, enclose the printable characters in quotes.

The following example sets the StartString to "123 #", followed by a <Ctrl C>, followed by "Sample ^ ^", followed by the unprintable character 255:

```
ApdDataPacket.StartString := '123 # '^C'Sample ^ ^ '#255;
```

StartString also supports fixed-length wildcards. The character ? within a string is interpreted as a wildcard character place-holder which will match any character in the input stream. Wildcards can occur anywhere in the StartString and EndString properties, including at the beginning or end of the strings. For example:

```
'ATI?'^M^J will match 'ATI0^M^J', 'ATI1^M^J...',
'END??' will match 'END12', 'END99'...,
'??BEGIN' will match 'AABEGIN', 'BBBEGIN'..., etc.
```

Since ? is now interpreted as a wildcard, an actual ? in the packet must be escaped by \ (backslash). To specify an actual \, use \\. For example:

```
'+FMFR\?' really means '+FMFR?' where the '?' is a literal '?'.
'\\ASC' really means '\ASC' where the '\' is a literal '\'.
```

See also: EndString, StartCond

TimeOut **property**

property TimeOut : Integer

Default: 2184 (~ 2 minutes)

↳ Determines how long a data packet waits for completion of a data stream.

If TimeOut is non-zero, it determines how long (in ticks) a data packet is allowed to wait for completion after it has started collecting data. After TimeOut ticks, the data packet relinquishes ownership of the data stream. If TimeOut is zero, the data packet holds ownership of the data stream until the EndString is received.

The timeout timer does not start until the start condition has been met. If StartCond = scString, the timer starts once the string defined by StartString has been received. If StartCond = scAnyData, the timer starts once the data packet has been enabled. If you need to start the timer once the packet starts actually collecting data, set the StartCond to scString and StartString to the wildcard char (?).

See “Data ownership” on page 134 for more information.

See also: EndString, OnTimeOut

Chapter 5: Script Component

This chapter describes the TApdScript component, which contains properties and methods for automating basic communications sessions.

A *script* is a list or file containing communications commands. Script languages are often provided by general-purpose communications programs to automate standard operations like logging on and off, file upload, and file download. The scripting support in TApdScript provides similar, though much simpler, script facilities for Async Professional applications.

TApdScript Component

The AdScript unit provides a single documented component: TApdScript. TApdScript implements a script language of about a dozen commands. While you wouldn't want to build a complete BBS using just these commands, they provide enough features to automate and simplify many standard tasks.

The script language

The basic syntax of the script language is shown in the following line of code:

```
<command> <data1> <data2>;<comment>
```

In this line of code, <command> describes the action to perform, <data1> and <data2> are optional arguments, and <comment> is an optional comment. The format of the arguments vary among commands. The various components of each line must be separated by at least one space or a comma. Additional spaces are permitted, but ignored. Commands are not case-sensitive.

The following is a list of supported commands followed by brief descriptions and discussions of the relationships between various commands:

:<label>	GOTO <label>
DISPLAY 'XX XX'	;<comment>
SEENDBREAK <duration in ms>	INITPORT <1..99>
DELAY <duration in ms>	IF CONNECTED <label>
SET <option> <data>	DONEPORT
UPLOAD <protocol>	SEND 'XXXXXX'
DOWNLOAD <protocol>	CHDIR <pathname>
DELETE <filemask>	RUN <command> <wait>
EXIT <exitcode>	IF SUCCESS <label>
WAIT 'XXXX' <timeout in ms>	IF TIMEOUT <label>
IF FAIL <label>	IF 1,2,3...127
WAITMULT 'XXX YYY ZZZ' <timeout in ms>	

:<label>

A point in the script file that can be jumped to via a GOTO or IF instruction. A label name can be any type of string without embedded spaces. For example “:TopOfLoop”, “:TOP_OF_LOOP” are both acceptable; “:top of loop” is not.

;<comment>

Any line that starts with a semicolon is considered a comment. Blank lines are also considered comments and may be freely added for readability.

INITPORT <Com1..Com99>

Opens the specified port. Only one port at a time may be opened. This number directly correlates with the ComNumber property of the TApdComPort component.

DONEPORT

Closes a port previously opened with INITPORT.

SEND 'XXXXXX'

Transmits the string “XXXXXX”. Control characters may be transmitted by preceding a character with ‘^’. For example, a control C character is represented by “^C”. You’ll use this feature most often when sending carriage returns. For example, SEND “myname^M” might be an appropriate response to a logon prompt where you would normally type your name and press Enter.

Note: Unlike Object Pascal, control characters must be inside the quote marks, if quote marks are necessary.

If the string does not contain any embedded blanks the beginning and ending quotes can be omitted. The quotes are required if the string has embedded blanks. For example:

SEND ABC sends ABC

SEND 'ABC' sends ABC

SEND A B C sends only the A ('B C' is considered a comment)

SEND 'A B C' sends A B C

WAIT 'XXXXX' <timeout in ms>

Waits up to <timeout in ms> milliseconds for a particular received string. The string comparison is always case insensitive. However, the string comparison need not be complete. If, for example, a host returns the string “Host XXXX ready” where XXXX might vary from session to session, the WAIT command should wait for “ready” only. As with the SEND command, beginning and ending quotes are only required if the string contains embedded blanks.

This command sets one of three conditions: SUCCESS, FAIL or TIMEOUT, which can be tested with the IF command. SUCCESS is set if the string is received before the timeout. TIMEOUT is set if the timeout expires before the string is received. FAIL is set if the timeout expires and all retries are exhausted.

IF SUCCESS/TIMEOUT/FAIL <label>

Tests the condition set by the last command and, if the tested condition is True, script execution jumps to <label>. If the condition is not True then execution continues with the next statement.

WAITMULTI 'XXX|ZZZ|YYY', <timeout in ms>

Waits up to <timeout in ms> milliseconds for one of several substrings. The bar character (|) separates the substrings. The comparisons are always case insensitive. The maximum length of the entire string is 255 characters. As with the SEND command, beginning and ending quotes are only required if the string contains embedded blanks.

This command sets a numeric condition result based on the substring received: '1' is set if the first substring is received, '2' is set if the second substring is received, and so on. If none of the strings are received, then TIMEOUT is set. If all retries have been exhausted, then FAIL is set.

IF 1,2,3...127 <label>

Tests the condition set by the last WAITMULTI command and, if the tested condition is True, script execution jumps to <label>. If the condition is not True, then execution continues with the next statement.

The following example sends a modem dial command, then waits for one of CONNECT, NO CARRIER, or BUSY responses. If none of the responses are received then control falls through to the GOTO statement:

```
send 'atdt260-9726^m'
waitmulti 'connect|no carrier|busy' 60000
if 1 HandleConnect
if 2 HandleNoConnect
if 3 HandleBusy
goto HandleTimeout
:HandleConnect
...proceed with session
:HandleNoConnect
...handle noconnect error
:HandleBusy
...handle busy error
...
```

GOTO <label>

Unconditionally jumps to <label>.

DISPLAY 'Just did something'

Generates a call to the TAPdScript component's OnScriptDisplay event handler. If the DisplayToTerminal property is True and a terminal component exists on the form, then the string is also displayed to the terminal. This can be used to monitor the progress of the script and to aid in debugging.

SENDERBREAK <duration in ms>

Transmits a break of <duration in ms> milliseconds.

DELAY <duration in ms>

Delays for <duration in ms> milliseconds. The script doesn't yield during delays so you should keep the delays as short as possible.

SET <option> <data>

Sets or resets a variety of port, script and protocol options. Some options require an additional argument, others do not. Table 5.1 shows a list of all options.

Table 5.1: *SET options*

BAUD <number>	Sets the Baud property of the associated TApdCustomComPort component. For further information regarding the allowable values, refer to the Baud property of TApdComPort.
DATABITS <5,6,7,8>	Sets the DataBits property of the associated TApdCustomComPort component. Allowable values are 5, 6, 7 or 8.
FLOW <RTS/CTS,XON/XOFF,NONE>	Sets flow control options for the associated TApdCustomComPort component. Allowable values are RTS/CTS for hardware flow control, XON/XOFF for software flow control, and NONE to turn off all flow control.
PARITY <NONE,ODD,EVEN,MARK,SPACE>	Sets the Parity property of the associated TApdCustomComPort component. Allowable values are NONE, ODD, EVEN, MARK or SPACE.
STOPBITS <1,2>	Sets the StopBits property of the associated TApdCustomComPort component. Allowable values are 1 and 2.

Table 5.1: SET options (continued)

RETRY <data>	Sets an internal retry count that is incremented whenever WAIT or WAITMULTI result in a TIMEOUT condition. When <retry count> TIMEOUTs have occurred the FAIL condition is set. The default is 1, meaning no retries are attempted.
DIRECTORY <pathname>	Sets the destination directory used during protocol receives.
FILEMASK <filemask>	Sets the file mask used during protocol file transfers. For non-batch protocols this must be a specific file name rather than a mask.
FILENAME <filename>	Sets the received file name for protocols that do not transfer the file name (all Xmodem protocols).
WRITEFAIL	Sets the WriteFailAction property to wfWriteFail for all protocols except Zmodem. This means that if an incoming file already exists the incoming file is skipped.
WRITERENAME	Sets the WriteFailAction property to wfWriteRename for all protocols except Zmodem. This means that if an incoming file already exists the incoming file is renamed (the first character of the file name is replaced with \$).
WRITEANYWAY	Sets the WriteFailAction property to wfWriteAnyway for all protocols except Zmodem. This means that if an incoming file already exists the existing file is overwritten.
ZWRITECLOBBER	Sets the ZmodemFileOption property to zfoWriteClobber. This means that if an incoming file already exists the existing file is overwritten.

Table 5.1: *SET options (continued)*

ZWRITEPROTECT	Sets the ZmodemFileOption property to zfWriteProtect option. This means that if an incoming file already exists the incoming file is skipped.
ZWRITENEWER	Sets the ZmodemFileOption property to zfWriteNewer option. This means that if an incoming file already exists the existing file is overwritten only if the incoming file is newer.
ZSKIPNOFILE <True/False>	Sets the ZmodemSkipNoFile property to True or False. When this option is True incoming files are skipped if they do not already exist on the receiving machine.

UPLOAD <protocol>

Starts transmitting files using <protocol>. <protocol> must be one of the following: XMODEM, XMODEM1K, XMODEM1KG, YMODEM, YMODEMG, ZMODEM or KERMIT. All files matching the mask previously specified by SET FILEMASK are transmitted.

DOWNLOAD <protocol>

Starts receiving files using <protocol>. <protocol> must be one of the following: XMODEM, XMODEM1K, XMODEM1KG, YMODEM, YMODEMG, ZMODEM or KERMIT. When using any of the Xmodem protocols, you must call SET FILENAME before DOWNLOAD to specify the name of the received file.

CHDIR <pathname>

Changes the current directory to the one specified by <pathname>. If the directory does not exist, the FAIL condition is set.

DELETE <filemask>

Deletes all files matching <filemask>. If no path is specified the current directory is used.

RUN <command> <wait>

Executes the specified command, batch file or program. <wait> can be True or False and determines whether the script waits for the command to complete its execution. <wait> is True by default.

Following is an example script showing how these commands might be used to log on to a host or terminal server:

```
SET RETRY 10                ;Try 10 times
:Again
SEND ^C                    ;Send an attention character
WAIT 'READY' 182           ;Wait 10 seconds for response
IF SUCCESS Logon           ;Got prompt, continue with logon
IF TIMEOUT Again           ;Try again if we timed out
IF FAIL, Done              ;Give up after 10 tries
:Logon
SEND 'Name, password^M'    ;Send name and password
...
:Done
SEND 'Bye^M'
```

EXIT <exitcode>

This will terminate the script and return the exit code as the Condition parameter in the OnScriptFinish event. If the exitcode is not specified, a Condition of SUCCESS will be passed to the OnScriptFinish event.

The exitcode parameter can be SUCCESS, TIMEOUT, FAIL or an integer value.

User functions and variables

User functions and variables are designed to provide a means by which your scripts can interact with the host application. When a user function or user variable are encountered, the script will fire events to either handle the event or to supply the value for the user variable.

User functions

User functions are indicated by a '&' as the first character in the name of the function. User functions can take a single optional parameter.

When a user function is encountered in the script, it will generate the OnScriptUserFunction event. This event will pass the name of the function converted to upper case and with the leading '&' removed in the Command parameter of the event. If a parameter has been specified for the user function, this will be passed in the Parameter parameter.

Following are examples of calls to user functions. In all cases, the OnScriptUserFunction event will be called with a Command parameter of “MYFUNCTION.” In the first example, the value of Parameter is empty. In the second and third examples, the value of Parameter is the string “Parameter” and “1234” respectively:

```
&MyFunction
&MyFunction 'Parameter'
&MyFunction 1234
```

User variables

User variables are indicated by a ‘\$’ as the first character in the name of the variable. User variables can appear in any place where a parameter is expected and they can appear as the parameter to a user function. In the latter case, the user variable will be evaluated before the user function.

When a user variable is encountered in the script, it will generate the OnScriptUserVariable event. The name of the user variable will be passed in the Variable parameter of the OnScriptUserVariable event. The name of the variable will be exactly as it appears in the script (including the leading ‘\$’). You will need to specify the value of the variable in the NewValue parameter of the event.

Following are examples of using user variables:

```
DISPLAY $MyVariable
&MyFunction $MyVariable
```

Executing scripts

A script is a list of commands in the format described in the previous section. The script can be an external ASCII text file or can be contained within a TStringList component. Scripts must be prepared with a call to PrepareScript before they can be executed. Preparing the script translates (compiles) it into memory. Syntax errors cause the EApxScriptError exception to be raised with one of the following error messages:

```
Not a valid script command. Line #
Bad format for first parameter. Line #
Bad format for second parameter. Line #
Label is referenced but never defined. Line #
Bad option in SET command. Line #
Error XXX while processing script. Line #
```

Scripts are always executed in the background in a fashion similar to file transfer protocols. Scripts are started with a call to `StartScript`, which returns immediately. If no script commands have been prepared (or can be prepared) `StartScript` raises the `EApXScriptError` exception. If the script was started successfully it continues in the background until completion (either successfully or due to an error). When the script is finished, it generates the `OnScriptFinish` event.

Other components

The script component always needs a `TApdCustomComPort` component descendent (i.e., `TApdComPort` or a user created descendent). When a `TApdScript` component is created, it searches the form for a `TApdCustomComPort` component and uses the first one it finds. If a `TApdCustomComPort` isn't found, `StartScript` creates one with all default values. In many cases it's best to create and customize the `TApdCustomComPort` yourself before starting the script.

A program using a script may also use a terminal window (`TAdTerminal`). No special action is required to associate the terminal with the `TApdScript` component. However, if the `TApdScript` component finds a terminal component it does two things. One, it deactivates the terminal during file transfers; and two, it sends the strings from `DISPLAY` commands to the terminal if the script property `DisplayToTerminal` is `True`. Even if no `DISPLAY` commands are used, the terminal window is useful during script development and debugging because it displays all received data.

If the script calls either `UPLOAD` or `DOWNLOAD` a `TApdProtocol` component is required. As with the `TApdCustomComPort`, the script will automatically create a `TApdProtocol` if it can't find one on the form. However, also like the `TApdCustomComPort`, the script provides only a few commands to customize the protocol so it's better to create the protocol component yourself before starting the script.

Debugging scripts

A script file is really an interpreted program and `TApdScript` is the interpreter. Like any program you write, script files may require a bit of debugging. Simple syntax errors will always be detected and reported by `PrepareScript`. Logic errors, however, may require some debugging effort on your part to find and correct.

`TApdScript` provides a variety of tools for debugging scripts. First, it provides three events for tracking the progress of the script:

1. `OnScriptCommandStart`—called before each command is executed.
2. `OnScriptCommandFinish`—called after each command is executed.
3. `OnScriptFinish`—called when the entire script completes.

It also provides the OnScriptDisplay event, which is called in response to each DISPLAY command in the script.

Finally, you can use the debug log available within the TApdComPort component to examine the data sent and received during a script session.

Example

This example creates a very simple script file to send the ATi4 command to a modem and wait for the OK response. Because this example includes a terminal window, the results of the ATi4 command are displayed in that window.

Create a new project, add the following components, and set the property values as indicated in Table 5.2.

Table 5.2: *Script example property values*

Component	Property	Value
TApdComPort	ComNumber	<as required>
TAdTerminal		
TApdScript	ScriptCommands	"send ati4^m"
		"wait ok 36"
TButton	Name	Start

Double-click on the Start button and modify the generated method to look like this:

```
procedure TForm1.StartClick(Sender: TObject);
begin
    ApxScript1.StartScript;
end;
```

This method starts the script. StartScript returns immediately while the script continues running in the background. Note that PrepareScript was not called. StartScript calls PrepareScript itself when either ScriptCommands contains commands or ScriptFile contains a file name. If the script contains any syntax errors, StartScript will raise an EApdScriptError exception.

Double-click on the script component's OnScriptFinished event handler in the Object Inspector and modify the generated method to look like this:

```
procedure TForm1.ApdsScript1ScriptFinish(
  CP: TObject; Condition: Integer);
begin
  ShowMessage('Script finished!');
end;
```

This method displays a message box when the script is finished.

Run the project and experiment with the generated program.

Hierarchy

TComponent (VCL)

- ❶ TApdBaseComponent (AdMisc)..... 8
 - TApdCustomScript (AdScript)
 - TApdScript (AdScript)

Properties

ComPort	Protocol	Terminal
DisplayToTerminal	ScriptCommands	❶ Version
InProgress	ScriptFile	

Methods

CancelScript	PrepareScript	StartScript
--------------	---------------	-------------

Events

OnScriptCommandStart	OnScriptDisplay	OnScriptParseVariable
OnScriptCommandFinish	OnScriptFinish	OnScriptUserFunction

Reference Section

CancelScript

method

```
procedure CancelScript;
```

↪ Cancels the background script.

Once a script is started, it executes all commands in the background without any help or interference from the foreground process. The only way to stop the script, short of letting it run to completion, is to call `CancelScript`. `CancelScript` stops the script and removes any resources (i.e., triggers, event handlers) the script may have installed. `CancelScript` does not free any components (`TApdComPort` or `TApdProtocol`) that the script may have created. Those components are re-used in subsequent calls to `StartScript` and are freed only when the script component is destroyed.

ComPort

property

```
property ComPort : TApdCustomComPort
```

↪ The comports component used by the script to send and receive data.

When the script is created it assigns the first comports component it finds on the form to `ComPort`. `ComPort` is also automatically filled in if a comports is created after the script component. If `ComPort` is unassigned when `StartScript` is called, `StartScript` dynamically creates a `TApdComPort` component and fills in `ComPort`.

DisplayToTerminal

property

```
property DisplayToTerminal : Boolean
```

↪ When `True`, the script `DISPLAY` commands display data to the terminal window.

Set this to `True` if the application has a terminal window and the terminal window should be used to display the strings from the script's `DISPLAY` commands. If the application doesn't have a terminal window this property is ignored. If this property is set to `False` or the application doesn't have a terminal window, the application must implement an `OnScriptDisplay` event handler in order to see the contents of the script's `DISPLAY` commands.


```
property InProgress : Boolean
```

↳ Returns True while a script is executing in the background.

Use this property to determine whether or not a script is executing in the background. A typical use would be to prevent a user from starting a new script or any other communications operation until the current script is finished.

OnScriptCommandFinish**event**

```
property OnScriptCommandFinish : TScriptCommandEvent
```

↳ Generated after each script command is executed.

This event complements the OnScriptCommandStart event and can also be used to implement single-stepping or for tracking the progress of the event. See OnScriptCommandStart for a description of the types and constants used by OnScriptCommandFinish.

For script commands that include a “wait” (WAIT, WAITMULTI, UPLOAD, DOWNLOAD) OnScriptCommandFinish is generated after setting up for the command and before the command actually finishes waiting. The indication that the command has finished waiting (or transferring) is the OnScriptCommandStart of the next command.

See also: OnScriptCommandStart

```

property OnScriptCommandStart : TScriptCommandEvent

TScriptCommandEvent = procedure(
    CP : TObject; Node : TApdScriptNode;
    Condition : SmallInt) of object;

TApdScriptNode = class(TObject)
    Command : TApdScriptCommand; Data : String; Option : TOption;
    Timeout : Word; Condition : Word;

TApdScriptCommand = (
    scNone, scComment, scLabel, scInitPort, scDonePort, scSend,
    scWait, scWaitMulti, scIf, scDisplay, scGoto, scSendBreak,
    scDelay, scSetOption, scUpload, scDownload, scChDir, scDelete,
    scRun, scExit);

```

↳ Generated before each script command is executed.

The primary purpose of this event is to provide a mechanism for single stepping through a script file or for tracking the progress of a script. Node contains the command to be executed. Node.Command is one of the TApdScriptCommand values shown above. Condition contains the current condition code, one of ccXxx values shown above.

See also: OnScriptCommandFinish

OnScriptDisplay

event

```

property OnScriptDisplay : TScriptDisplayEvent

TScriptDisplayEvent = procedure(
    CP : TObject; const Msg : String) of object;

```

↳ Generated in response to script DISPLAY commands.

The script processor doesn't make any assumptions about how to display the contents of DISPLAY commands. Instead, it generates an OnScriptDisplay event passing Msg, the string to be displayed. The application can then display the string in whatever manner necessary.

One exception to this process occurs when the DisplayToTerminal property is True and the script component finds a TAdTerminal window component on the form. In that case the contents of the DISPLAY command are shown in the terminal window before OnScriptDisplay is generated.

```
property OnScriptFinish : TScriptFinishEvent  
  
TScriptFinishEvent = procedure(  
    CP : TObject; Condition : SmallInt) of object;
```

↳ Generated at the end of the script.

This event is generated when the end of the script file or script list is reached and there are no more script commands to execute. It is also generated if the script encounters a fatal error during processing.

If an application dynamically creates a TApdScript component it should not free that component, or any component that the script uses, during this event. Instead, it should set a flag or post a message to itself noting that the script component can safely be destroyed at some later point in the program.

See also: InProgress

OnScriptParseVariable

```
property OnScriptParseVariable : TScriptParseVariableEvent  
  
TScriptParseVariableEvent = procedure(  
    CP : TObject; const Variable : String;  
    var NewValue : String) of object;
```

↳ Generated when a value is needed for a user variable.

This event is generated when a user variable is encountered in the script. User variables can appear any place where a parameter is expected. They are indicated by a leading '\$'. The name of the user variable will be passed exactly as it is seen in the script (including the '\$'). In this event handler, the value of the user variable needs to be specified in the NewValue. If this value is not specified, the value of the variable will be assumed to be blank.

Refer to the section “User functions and variables” on page 152 for more information.

See also: OnScriptUserFunction

```
property OnScriptUserFunction : TScriptUserFunctionEvent  
  
TScriptUserFunctionEvent = procedure(  
    CP : TObject; const Command : String;  
    const Parameter : String) of object;
```

↳ Generated when a user function has been encountered in the script.

This event is generated when a user function is encountered in the script. User functions are indicated by a leading '&'. The name of the function will be passed to this event in the Command parameter. This name will be converted to upper case and have the leading '&' removed. Whatever functionality is needed to handle the user function should be implemented in this event.

Refer to the section “User functions and variables” on page 152 for more information.

See also: OnScriptParseVariable

PrepareScript**method**

```
procedure PrepareScript;
```

↳ Prepares the script command list and checks for syntax errors.

Before a list or file of script commands can be processed, it must first be prepared. This is similar in concept to compiling a program. Each script command is read, checked for syntax errors, and written to an internal list of compiled commands. When StartScript is called, it is this internal list of compiled commands that is executed.

When PrepareScript finds a syntax error it raises an EApxScriptError exception with one of the following error messages and the offending line number:

```
Not a valid script command. Line #  
Bad format for first parameter. Line #  
Bad format for second parameter. Line #  
Label is referenced but never defined. Line #  
Bad option in SET command. Line #  
DOS error XXX while processing script. Line #
```

Protocol**property**

```
property Protocol : TApdCustomProtocol
```

↳ Determines the protocol used by UPLOAD and DOWNLOAD commands.

When the script component is created it searches the form for an existing TApdCustomProtocol and uses the first one it finds. If it doesn't find any, it uses the first TApdCustomProtocol later dropped onto the form.

If an UPLOAD or DOWNLOAD command is processed and Protocol is unassigned, the script component creates a TApdProtocol component with default values. That protocol component is destroyed when the script component is destroyed. The automatic creation of the protocol component provides very little control over the protocol. The recommended approach is for the application to create the protocol before the script is executed.

ScriptCommands**property**

```
property ScriptCommands : TStrings
```

↳ The list of script commands.

Although declared as a TStrings component, this component is treated as a TStringList and uses the TStringList property editor for adding and editing script commands at design time.

Note that ScriptCommands are used only when the ScriptFile property is empty. If the ScriptFile contains a file name, then script commands are read from that file regardless of the contents of ScriptCommands.

For convenience, at design time the script component loads the commands from ScriptFile into ScriptCommands whenever a new ScriptFile property value is set. If the contents of ScriptCommands are changed the script component automatically writes those changes to ScriptFile, but only when the script component is destroyed or a new ScriptFile set. If the contents of ScriptCommands is changed and the project is run without first closing, the project the changes to ScriptCommands will be lost.

ScriptFile**property**

```
property ScriptFile : String
```

↳ A file containing script commands.

When ScriptFile contains a file name PrepareScript and StartScript always read that file to build/execute the script, ignoring the current contents ScriptCommands.

See ScriptCommands for more information concerning the relationship of ScriptCommands and ScriptFile.

```
procedure StartScript;
```

- ↳ Begins executing the script in the background.

StartScript checks the internal table of compiled commands. If that table is empty, it calls PrepareScript to compile the commands in the file specified by ScriptFile. If ScriptFile is empty, PrepareScript tries to compile the list of commands specified by ScriptCommands. If that list is also empty, the EApdScriptError exception is raised.

If StartScript finds or creates a list of compiled commands, it begins executing those commands. It continues executing commands until it encounters a command requiring a “wait” (WAIT, WAITMULTI, UPLOAD, DOWNLOAD). It then sets up appropriate triggers and trigger handlers and exits back to the application. When the triggers occur, the script engine regains control in the background and continues executing until the next wait command, when this process is repeated.

Terminal**property**

```
property Terminal : TApdBaseWinControl
```

- ↳ Determines the terminal component used by the DISPLAY commands.

When the script component is created it searches the form for an existing terminal and uses the first one it finds. If it doesn't find any, it uses the first terminal later dropped onto the form. Unlike comport and protocol components, the script component never creates a terminal component.

If DisplayToTerminal is True, the contents of DISPLAY commands are written to this terminal window.

Chapter 6: State Machine Components

A state machine is simply a device or technique that receives input and acts upon that input based on the current condition of the device or technique. Async Professional uses state machines for a variety of tasks; such as managing protocol file transfers, sending and receiving faxes, collecting data packets, and monitoring the progress of a connection attempt.

State machines are one of the fundamental techniques used for asynchronous communications, where a command is transmitted and the reply is received later in the session. Consider the simple task of initializing a modem to detect Caller ID information and answering a call. The first step (after opening the correct serial port) is to send a generic initialization command ("ATZ"<CR>), then wait for the modem to return either a success response ("OK") or a failure response ("ERROR"). The next step is to send the AT command to enable Caller ID detection and text responses ("AT#CID=1"<CR>) and wait for either a success or failure response. Finally, the project waits for the modem's ring indicator ("RING"), answers the call ("ATA"<CR>) and waits for the connection response ("CONNECT"). Each instance of a command and response can be thought of as a separate state in a state machine, as Table 6.1 illustrates.

Table 6.1: *Simple State Machine*

State	Output	Input	Next State
Send Init	"ATZ"<CR>	"OK"	Send CID
		"ERROR"	Fail
Send CID	"AT#CID=1"<CR>	"OK"	Wait for 1st ring
		"ERROR"	Fail
Wait for 1st ring	"RING"		Wait for 2nd ring
Wait for 2nd ring	"RING"	Answer	
Answer	"ATA"<CR>	"CONNECT"	Connected
		"NO CARRIER"	Fail
Fail	Cleanup and report the error		
Connected	Continue the session		

A simple, two or three-state state machine is relatively painless to create. The state machine is usually driven by `TApdDataPacket` components monitoring for success or fail conditions. When `TApdDataPacket`'s `OnStringPacket` event is generated, the state machine progresses according to the collected data. Once a state machine grows to twenty or thirty states, with multiple conditions defining the state progression, the project code can get cumbersome, difficult to maintain, and hard to visualize.

The `TApdStateMachine` and `TApdState` components exist in order to assist in the development of orderly, well-defined, and easy to maintain and visualize state machines. The `TApdStateMachine` component is a container for `TApdState` components. `TApdState` components contain conditions that determine the accepted input to drive the state machine. To aid in the visualization of your state machine, the `TApdStateMachine` and `TApdState` components are visual components, showing connection lines with captions and customizable colors. When the state machine is executing, the currently active state is highlighted to aid in debugging and to show the progression of the state machine.

State machine philosophy

In order for the Async Professional state machine components to be useful, their design philosophy should be understood. The `TApdStateMachine` component contains and manages the `TApdState` components. The `TApdState` components own a `TCollection` descendent that determines the conditions required to progress to another `TApdState`. Each `TApdState` component can have either a single or multiple conditions. When the `TApdStateMachine` activates a state, the `TApdStateMachine` creates a list containing `TApdDataPacket` components configured according to the properties of each condition. Once one of those conditions is met, the next state is activated and the `TApdStateMachine` awaits further input to satisfy the new state's conditions.

The `TApdStateMachine` is the only state machine component that interfaces with the `TApdComPort` or `TApdWinsockPort`. The `TApdState` components merely define the conditions that the `TApdStateMachine` monitors.

States and conditions

A `TApdState` component contains a `TCollection` descendent (`TApdStateConditions`); which contains `TCollectionItem` descendents (`TApdStateCondition`). The `TApdStateCondition` class defines conditions for the `TApdState`.

A condition, in the realm of the state machine, consists of a `StartString` and an `EndString`, a `PacketSize`, and a `Timeout` to determine the condition's data requirement. The condition also contains a pointer to the next state to activate when the data conditions are met; and an `ErrorCode` to keep track of the relative success or failure of the state machine. The condition also defines visual aspects (called a "Connectoid") such as the caption, line color and line width.

A TApdState component can be either activated or deactivated, and only activated or deactivated by a TApdStateMachine. When a TApdState is activated, the TApdStateMachine assembles the collection of conditions into a list of TApdDataPackets, and assigns internal event handlers as appropriate. When the TApdDataPacket's data match conditions, or timeout, is met, the current state is deactivated and the next state is activated.

Adding, editing, and deleting conditions

The multiple-condition capability of the TApdState component does not lend itself very well to programmatic modification, or modification through the Object Inspector. The TApdState component installs a state condition editor dialog accessed by right-clicking the component and selecting the “Edit condition...” menu item. A dialog box is displayed, as shown in Figure 6.1, which provides access to add, edit and delete conditions.

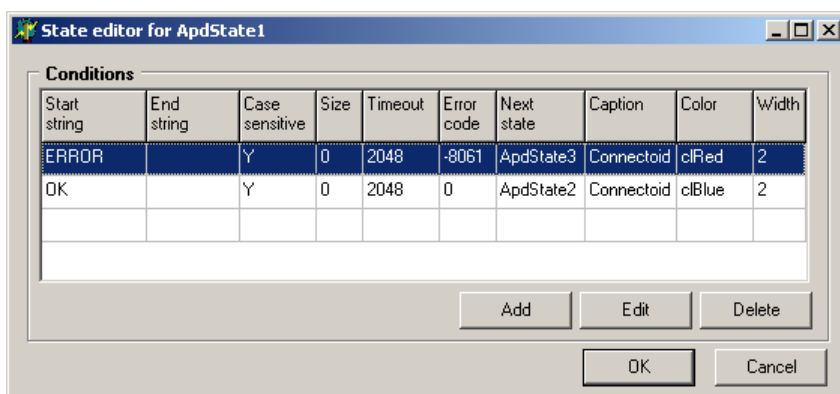


Figure 6.1: TApdState component editor.

The grid displays the current values for the installed conditions. To add a new condition to this TApdState component, click Add; to edit the selected condition click Edit; to delete the selected condition click Delete. When the modifications are complete, click OK to update the collection of conditions. To cancel any changes and revert to the original conditions, click Cancel.

When a condition is added or edited, the Condition editor, shown in Figure 6.2, is displayed. This dialog box permits editing the fields of the TApdStateCondition being added or edited.

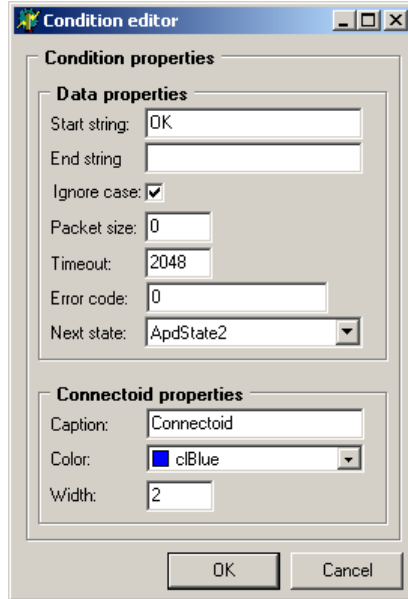


Figure 6.2: TApdStateCondition property editor.

The controls are preinitialized with the current TApdStateCondition values when the condition is being edited, and preinitialized with default values when the condition is being added. Most of the controls are self-explanatory, with the exception of the “Next state” and “Color” drop-down combo boxes. The drop-down list for “Next state” contains the names of all TApdState components owned by the TApdStateMachine that owns the TApdState being edited. The “Color” drop-down list contains the system colors and Delphi color constants. Hex or RGB representations are not supported. Click OK to accept the changes or click Cancel to cancel the changes. Note that the changes will not take effect until the TApdState component editor is terminated.

The state machine in action

For illustration, we will create a simple state machine that will initialize a modem, initialize the modem for Caller ID detection, collect the Caller ID information, and answer an incoming call. This process can be broken up into four states: initialization, Caller ID initialization, waiting for the ring signals from the modem, and answering the call. As you will see, the multiple condition capability of the TApdState, and some carefully planned recursion, will let us wait for the rings and detect the Caller ID information in a single TApdState.

Due to the number of properties that are available when defining the state conditions, only the properties that will be changed will be mentioned, other properties can be left at their default values.

Create a new project and drop a `TApdComPort` component and a `TApdStateMachine` component onto the new form. Next, drop four `TApdState` components onto the `TApdStateMachine`.

Select `ApdState1`, change the `OutputOnActivate` property to `"ATZ^M"`. Right-click the component and select the "Edit conditions..." menu. Add a new condition with a `StartString` of `"OK"` and set the `NextState` to `ApdState2`. This will tell the `TApdState` that we want to send `"ATZ"<CR>` when activated and we will wait for an `"OK"` before `ApdState2` is activated.

Select `ApdState2`, change the `OutputOnActivate` property to `"AT#CID=1^M"`. Invoke the conditions property editor and add a new condition with a `StartString` to `"OK"` and the `NextState` set to `ApdState3`. This will tell the `ApdState2` that we want to send `"AT#CID=1"<CR>` when the state is activated and wait for an `"OK."`

Select `ApdState3`, and invoke the conditions property editor. In this state, we will wait for two `"RING"` signals, and the three Caller ID tags. Invoke the conditions property editor and add a new condition with `StartString` set to `"RING."` Don't set the `NextState` yet; it will be set programmatically so the two `"RING's"` can be captured. Add another condition and set the `StartString` to `"DATE:"`, `EndString` to `"^M"` and `NextState` to `ApdState3`. Add two more conditions identical to the last except for a `StartString` of `"NAME:"` and `"NMBR:"`. These conditions will tell `ApdState3` that we want to know when `"RING"` is received, and when the Caller ID tags are received. Since we may or may not receive the Caller ID tags, and they may be in a different order, their `NextState` properties will point to their own `TApdState`.

We will stay in this state until we tell the state that we want to progress to the next state. We will also want to record the Caller ID information for later processing. To do this, create the OnStateFinish event handler for ApdState3 and enter the following:

```
procedure TForm1.ApdState3StateFinish(State: TApdCustomState;
  Condition: TApdStateCondition; var NextState: TApdCustomState);
begin
  { decide what to do when we receive "RING"s }
  if Condition.StartString = 'RING' then begin
    { it's our RING condition }
    State.Tag := State.Tag + 1;
    if State.Tag > 1 then
      {we've seen at least 2 rings, progress to the next state}
      NextState := ApdState4
    else
      { we've seen less than 2 rings, wait for more }
      NextState := ApdState3;
  end else if Condition.StartString = 'DATE:' then
    { it's our Date CID tag }
    CIDDate := ApdStateMachine1.DataString
  else if Condition.StartString = 'NMBR:' then
    { it's our Number CID tag }
    CIDNumber := ApdStateMachine1.DataString
  else if Condition.StartString = 'NAME:' then
    { it's our Name CID tag }
    CIDName := ApdStateMachine1.DataString;
end;
```

Select ApdState4 and change the OutputOnActivate property to “ATA^M”. Invoke the conditions property editor, add a new condition and change the StartString to “CONNECT.”

Next, select the ApdStateMachine1 component on the form, set the StartState property to ApdState1 and the TerminalState property to ApdState4. Add an OnStateMachineFinish event handler to provide notification when the state machine is complete and we are connected. A simple ShowMessage(“Connected”) will suffice for our purposes here.

Finally, drop the obligatory TButton on the form and create the OnClick event handler. This event is where we will start the state machine. Add the following code to the OnClick event handler:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ApdStateMachine1.Start;
end;
```

One implementation of the preceding example is illustrated in Figure 6.3.

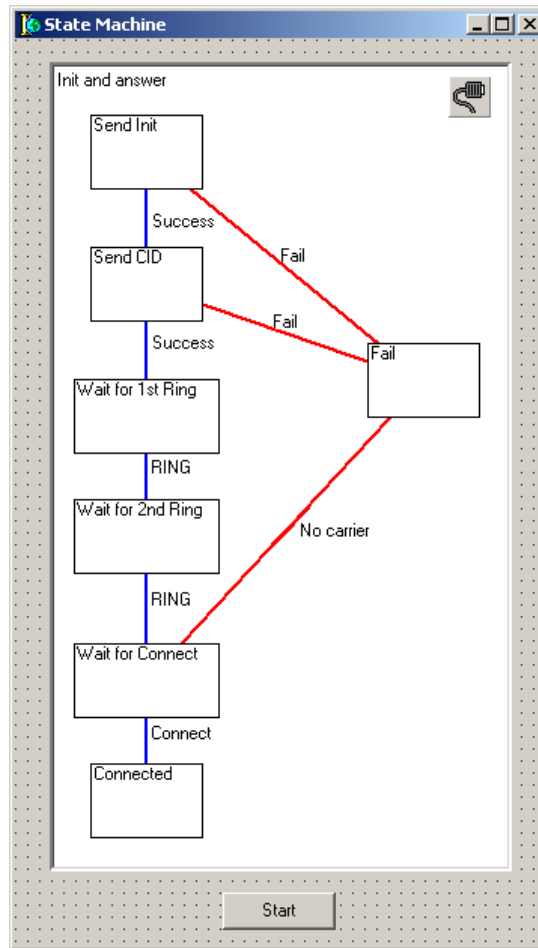


Figure 6.3: State machine example.

Compile and run the project. Click the button to start the state machine. As the states are activated they will be highlighted in yellow (change the `TApdStateActiveColor` property of the `TApdState` component to change the default highlight color). The modem will first be initialized with "ATZ", after the "OK" is received the Caller ID initialization is sent. After OK is received, `ApdState3` will be highlighted waiting for incoming calls. When a call is detected by the modem, `ApdState3` will wait for two "RING"s, collecting whatever Caller ID information is available. After two "RING"s, "ATA" is sent to the modem to answer the call and we will wait for the "CONNECT" response from the modem.

TApdStateMachine Component

The TApdStateMachine component manages TApdState components to provide an easy to use, easy to maintain, and easy to visualize state machine. The TApdStateMachine component creates internal data collection mechanisms to monitor for conditions defined by the owned TApdState components, and activates and deactivates the owned TApdState components according to the collected data.

Hierarchy

TScrollingWinControl (VCL)

- ❶ TApdBaseScrollingWinControl (OOMisc) 8
 - TApdCustomStateMachine (AdStMach)
 - TApdStateMachine (AdStMach)

Properties

ComPort	DataStream	TerminalState
CurrentState	LastErrorCode	❶ Version
Data	StartState	
DataSize	StateNames	

Methods

Cancel	Start
--------	-------

Events

OnStateChange	OnStateMachineFinish
---------------	----------------------

Reference Section

Cancel

method

```
procedure Cancel;
```

↪ Cancels the state machine.

Call the Cancel method to cancel the TApdStateMachine. The current state will be deactivated, and the OnStateMachineFinish event will be generated. All resources allocated by the TApdStateMachine to manage the current state will be deallocated.

When the Cancel method is used to terminate the TApdStateMachine, the ErrorCode parameter of the OnStateMachineFinish event will be ecCancelRequested.

See also: OnStateMachineFinish

ComPort

property

```
property ComPort : TApdCustomComPort
```

↪ Determines the serial port used by the TApdStateMachine component.

A properly initialized TApdComPort or TApdWinsockPort must be assigned to this property before starting the TApdStateMachine.

ComPort is usually set automatically at design time to the first TApdCustomComPort component that the TApdStateMachine finds on the form. Use the Object Inspector to select a different TApdCustomComPort component, if needed.

Setting the ComPort property at run time is necessary only when using a dynamically created TApdCustomComPort or TApdStateMachine, or when selecting among several TApdCustomComPort components.

To use this component through a TAPI interface, start the TApdStateMachine after the OnTapiPortOpen event has been generated (after TAPI hands the application an initialized and open serial port).

CurrentState

read-only, run-time property

```
property CurrentState : TApdCustomState
```

↳ The TApdCustomState that is currently active.

During the execution of a state machine, different states will be activated and deactivated depending on the data that is received. CurrentState is the TApdCustomState component that is currently active.

At run time, the current state will be displayed with a highlighted background.

Data

read-only, run-time property

```
property Data : Pointer
```

↳ A pointer to the collected data.

As a TApdStateMachine manages an active TApdState component, internal TApdDataPackets collect the data defined by the conditions. Data is identical to the Data parameter of TApdDataPacket's OnPacket event handler.

See also: DataSize, DataString

DataSize

read-only, run-time property

```
property DataSize : Integer
```

↳ The size of the collected data.

As the TApdStateMachine manages an active TApdState component, internal TApdDataPackets collect the data defined by the conditions. DataSize is the size of the collected data. DataSize is identical to the Size parameter of TApdDataPacket's OnPacket event handler.

See also: Data, DataString

DataString

read-only, run-time property

```
property DataString : string
```

↳ The collected data.

As the TApdStateMachine manages an active TApdState component, internal TApdDataPackets collect the data defined by the conditions. DataString is the string that was collected. DataString is identical to the Data parameter of TApdDataPacket's OnStringPacket event handler.

See also: Data, DataSize

```
property LastErrorCode : Integer
```

↳ The ErrorCode of the last TApdStateCondition.

When a TApdState is deactivated, the condition whose data requirements were met defines an ErrorCode value. ErrorCode is intended to provide a numerical result of the condition. For example, if the TApdState conditions defined a log in sequence, and the failure condition was met, ErrorCode could be set to TPS_LOGINFAIL (an Async Professional predefined constant found in AdExcept.pas). This property can be used for status reporting, or to provide a reason for a state machine failure. The TApdStateMachine ignores this value.

See also: OnStateMachineFinish

```
property OnStateChange : TApdStateMachineStateChangeEvent
```

```
TApdStateMachineStateChangeEvent = procedure(  
    StateMachine : TApdCustomStateMachine;  
    FromState : TApdCustomState;  
    var ToState : TApdCustomState) of object;
```

↳ Defines an event handler that is called when the CurrentState changes.

When the TApdStateMachine progresses through the owned TApdState components the TApdState components are deactivated and the next state is activated. The OnStateChange event is generated immediately after the deactivation. The next state is activated immediately after this event handler returns.

This event handler can be used to provide status indicators, or to modify the next state based on the collected data. To specify a different TApdState, set the ToState parameter to the new state to activate. To terminate the TApdStateMachine, set ToState to nil.

StateMachine is the TApdCustomStateMachine that generated the event. FromState is the TApdCustomState component that was deactivated. ToState is the TApdCustomState that StateMachine will activate.

See also: CurrentState, OnStateMachineFinish

```
property OnStateMachineFinish : TApdStateMachineFinishEvent

TApdStateMachineFinishEvent = procedure(
    StateMachine : TApdCustomStateMachine;
    ErrorCode : Integer) of object;
```

↳ Defines an event handler that is called when the state machine terminates.

When the TApdCustomState machine terminates, this event is generated to provide notification of the termination. The TApdCustomStateMachine can terminate due to the Cancel method, upon activation of the TerminalState TApdState, or when a condition is met and the NextState property is nil or invalid.

StateMachine is the TApdCustomStateMachine that generated the event. ErrorCode is an integer defining the reason for the termination. If the TApdCustomStateMachine was terminated due to the Cancel method, ErrorCode will be ecCancelRequested. For all other reasons, ErrorCode will be the value assigned to the ErrorCode property of the condition whose data match parameters were met.

See also: Cancel, LastErrorCode

```
procedure Start;
```

↳ Start starts the state machine.

The Start method starts the state machine. Once Start is called, the TApdState determined by the StartState is activated. The StartState property must be assigned prior to calling Start.

See also: Cancel, StartState,

```
property StartState : TApdCustomState
```

↳ Determines the state that is activated first.

When the Start method is called, the TApdState component determined by StartState is activated, which starts the state machine. If Start is called and StartState is not assigned, the EStartStateNotAssigned exception will be raised.

See also: OnStateChange, Start

```
property StateNames : TStrings
```

- ↳ Provides the names of all TApdState components associated with the state machine.

The StateNames property is primarily for internal use, to provide the names of all TApdState components that are associated with the TApdStateMachine. The Strings value is the Name of the TApdState component.

```
property TerminalState : TApdCustomState
```

- ↳ Determines the state that terminates the state machine.

The TApdCustomStateMachine can terminate due to the Cancel method, upon activation of the TerminalState TApdState, or when a condition is met and the NextState property is nil or invalid. The TerminalState property determines which state is the terminal state. When this state is activated, the OutputOnActivate property is transmitted through the serial port, and any conditions defined by the TerminalState configured. When one of the conditions is met, the TerminalState is deactivated and the OnStateMachineFinish event is generated.

See also: Cancel, OnStateMachineFinish

TApdState Component

The TApdState component defines the conditions for which the state machine is monitoring at a given time during execution of the state machine. The TApdState component does not interact with the port, it simply contains the data requirements and state progression elements of the state it is representing.

The TApdState component maintains a collection of state conditions, which means that a single TApdState component can define several different conditions at one time. For example, at the beginning of a ZModem file transfer, the transmitter can begin the transfer by sending “rz” <CR>, or a ZInit block (“**[18]B00000000000000[0D]??”). This can be represented in a single TApdState component by adding two conditions, with their NextState properties pointing to the TApdState that responds with the ZRInit packet.

The conditions are contained in the Conditions property of TApdState, which is a TCollection descendent. This property is a collection of TApdStateCondition classes, which are descendents of TCollectionItem. The TApdStateCondition defines data match strings and timeouts similar to the TApdDataPacket, as well as a TApdStateConnectoid which defines the visual connectoid between the states. The TApdStateConditions, TApdStateCondition and TApdStateConnectoid classes do not lend themselves well to programmatic manipulation, they are better suited to changing from the Object Inspector and property editors.

TApdStateCondition

The TApdState component maintains TApdStateConditions, which define the conditions to satisfy the state. TApdStateCondition is a TCollectionItem descendent, and has the properties shown in Table 6.2.

Table 6.2: *TApdStateConditions*

Name	Type	Description
StartString	String	Defines the beginning of the data match string.
EndString	String	Defines the ending of the data match string.
IgnoreCase	Boolean	Determines whether the data match string must match case or not.
PacketSize	Integer	Defines packet length criteria for the data match string.

Table 6.2: *TApdStateConditions (continued)*

Name	Type	Description
ErrorCode	Integer	Defines an error code when this condition is satisfied.
NextState	TApdCustomState	Defines the state that is activated once this condition is satisfied.
Connectoid	TApdStateConnectoid	Defines how the link from this state to NextState is rendered.

StartString, EndString, IgnoreCase and PacketSize are very similar to the TApdDataPacket properties of the same name. The interface is simplified somewhat with the omission of the StartCond and EndCond properties. If StartString is an empty string the data collection mechanism begins immediately once the state is activated. If StartString is not an empty string the data collection mechanism begins when the StartString has been received. If PacketSize is 0 the size of the collected data is ignored. If PacketSize is greater than 0 then condition will be satisfied when PacketSize characters have been received. If EndString is an empty string the condition is satisfied if either the StartString has been received or PacketSize characters have been received. If EndString is not an empty string the condition is satisfied when the EndString has been received after the data collection mechanism begins.

Hierarchy

TGraphicControl (VCL)

- ❶ TApdBaseGraphicControl (OOMisc) 8
 - TApdCustomState (AdStMach)
 - TApdState (AdStMach)

Properties

Active

ActiveColor

Conditions

Glyph

GlyphCells

InactiveColor

OutputOnActivate

❗ Version

Methods

Terminate

Events

OnStateActivate

OnStateFinish

Reference Section

Active

read-only, run-time property

property Active : Boolean

Default: False

↪ Indicates whether this state is active or not.

As the TApdStateMachine component progresses through the TApdState components that define the state machine, the TApdStateMachine component activates and deactivates the TApdStates. This property indicates whether this TApdState is active or not.

When the TApdStateMachine component activates a TApdState, the conditions that define the state are read from the collection maintained by the Conditions property of TApdState, and corresponding TApdDataPackets are created. When the first condition is satisfied, the TApdStateMachine deactivates the current state and activates the next TApdState. Active is set after the data collection mechanism is configured, and after the data collection mechanism is destroyed.

See also: OutputOnActivate

ActiveColor

property

property ActiveColor : TColor

Default: clYellow

↪ Determines the color of the component when the state is active.

When the TApdState is active, it will be rendered with a background color determined by the ActiveColor property. When the state is inactive it will be rendered with a background color determined by the InactiveColor property. If the Glyph property is assigned, that image will be rendered instead with a background determined in part by GlyphCells.

See also: Active, Glyph, GlyphCells, InactiveColor


```
property Conditions : TApdStateConditions
TApdStateConditions = class(TCollection)
public
    function Add : TApdStateCondition;
    property Items[Index: Integer] : TApdStateCondition;
end;
```

↳ Defines the collection of conditions for the state.

This property is a collection of `TApdStateCondition` classes, which defines the data match conditions for this `TApdState` component. The `TApdStateConditions` property defines a property editor, which is the same as the component editor for this component. This design time editor is discussed earlier in this chapter.

```
property Glyph : TBitmap
```

↳ Defines a bitmap used to illustrate the state.

If the `Glyph` property is not assigned, the `TApdState` component is rendered as an empty rectangle with a caption. If the `Glyph` property is assigned a valid bitmap, the component is rendered with the bitmap.

The `Glyph` can contain multiple cells to display a different image for each stage of the `TApdState` component. The `GlyphCells` property determines the number of cells contained in `Glyph`. See `GlyphCells` for a more detailed description.

See also: `GlyphCells`

property GlyphCells : Integer

Default: 1

↪ Determines the number of cells contained in the Glyph property.

The Glyph property can be unassigned, it can contain a single image that is displayed for the duration of the state machine, or it can display a different image depending on the state of the TApdState. GlyphCells determines how many images are contained in Glyph. GlyphCells supports values from 0 through 3, and renders the cell according to the following:

Glyph Cell	Condition When Rendered
0,1	When the TApdState is inactive.
2	When the TApdState is active.
3	After the TApdState has been deactivated.

For example, if Glyph contained a single cell that should be displayed for the duration of the state machine, GlyphCells can be either 0 or 1. To display one image when the Active property is False and another image when Active is True, GlyphCells should be 2 and the Glyph should contain two images with the same dimensions placed side by side.

When the Glyph cell is rendered, the Caption property will be displayed above the image. The Active, ActiveColor and InactiveColor properties determine the background color behind the Caption.

See also: Active, ActiveColor, Glyph, InactiveColor

InactiveColor

property InactiveColor : TColor

Default: clWhite

↪ Determines the color of the component when the state is not active.

When the TApdState is active, it will be rendered with a background color determined by the ActiveColor property. When the state is inactive it will be rendered with a background color determined by the InactiveColor property. If the Glyph property is assigned, that image will be rendered instead with a background determined in part by GlyphCells.

See also: Active, ActiveColor, Glyph, GlyphCells

```
property OnStateActivate : TApdStateNotifyEvent
```

```
TApdStateNotifyEvent = procedure(  
    State : TApdCustomState) of object;
```

↳ Defines an event that is generated when the state is activated.

OnStateActivate provides notification when the TApdState component becomes active. This event is generated when the TApdStateMachine component activates this TApdState component. When this event is generated, the TApdStateMachine has already created and configured the data collection mechanism defined by Conditions; and the string defined by OutputOnActivate has already been transmitted. Use this event for notification and status purposes only.

State is the TApdState component that generated the event.

See also: Activate, OutputOnActivate

OnStateFinish**event**

```
property OnStateFinish : TApdStateFinishEvent
```

```
TApdStateFinishEvent = procedure(  
    State : TApdCustomState; Condition : TApdStateCondition;  
    var NextState : TApdCustomState) of object;
```

↳ Defines an event that is generated when the state's conditions are satisfied.

OnStateFinish provides notification when a TApdState's conditions have been met. This event is generated immediately before this state is deactivated and the Condition's NextState is activated.

State is the TApdState component that generated the event. Condition is the TApdStateCondition whose data match conditions were satisfied. NextState is the TApdState component that will be activated next. NextState can be changed during this event to provide a mechanism to select the next state based on collected data or other factors. If NextState is nil the TApdStateMachine will terminate the state machine.

See also: Active, OnStateActivate

```
property OutputOnActivate : string
```

↳ Defines a string to transmit when the state is activated.

As the TAPdStateMachine progresses through the state machine, successive TAPdState components are activated and deactivated. The string defined by OutputOnActivate is transmitted by the TAPdStateMachine through the TAPdCustomComPort determined by TAPdStateMachine's ComPort property when the state is activated. OutputOnActivate provides a convenient way of initiating the state.

For example, during a modem initialization state machine you may want to transmit "ATZ"<CR> and wait for an "OK" or "ERROR" response from the modem. The collection of conditions would contain a TAPdStateCondition defining a StartString of "OK" and another TAPdStateCondition defining a StartString of "ERROR." The OutputOnActivate property could be set to "ATZ^M" at design time. When the state is activated, "ATZ"<CR> will be transmitted and the response received by the same state's conditions.

OutputOnActivate supports printable and non-printable characters in the string. At design time, enter non-printable characters in caret or pound notation. For example, to add a <CR> character, the "^M" or "#13" sequence (without quotes) can be used. At run time, the literal character can be used.

See also: Active, OnStateActivate

Terminate**method**

```
procedure Terminate(ErrorCode : Integer)
```

↳ Terminates the TAPdState component.

Under normal circumstances, the data match conditions defined by the Conditions property of TAPdState would cause the TAPdStateMachine to deactivate the state. The Terminate method can be used to stop the data collection mechanism. When the Terminate method is called, the TAPdStateMachine will deactivate the state (causing the OnStateFinish event to be generated), and the OnStateMachineFinish event of TAPdStateMachine will be generated. The ErrorCode parameter of this method will be passed along to the ErrorCode parameter of the OnStateMachineFinish event handler.

See also: OnStateFinish

Chapter 7: Status Light Components

The components included in this chapter: `TApdStatusLight` and `TApdSLController`, allow you to add a status light display, similar to the LEDs found on external modems, to your communications programs. The display can give a visual indication of modem status lights such as RTS and RI.

`TApdStatusLight` is a simple component that displays two bitmaps, or two different colors, depending on whether the light is “lit” or “unlit.” The component’s `Lit` property determines which of the two states is displayed, and its `Glyph` property determines whether bitmaps or solid colors are used. This component works hand-in-hand with the `TApdSLController` component.

The `TApdSLController` component monitors the status of a `TApdComPort` component and changes the state of one or more `TApdStatusLight` components to reflect that status.

TApdStatusLight Component

TApdStatusLight is a simple component that displays two bitmaps, or two different colors, depending on whether the light is “lit” or “unlit.” The component’s Lit property determines which of the two states is displayed, and its Glyph property determines whether bitmaps or solid colors are used.

This component works hand-in-hand with the TApdSLController component (see page 191). TApdSLController reacts to changes in serial port status and changes the Lit property of various TApdStatusLight components to reflect the status of the port.

Hierarchy

TGraphicControl (VCL)

 TApdCustomStatusLight (AdStatLt)

 TApdStatusLight (AdStatLt)

Properties

Glyph	LitColor
Lit	NotLitColor

Reference Section

Glyph

property

```
property Glyph : TBitmap
```

↪ Determines two custom bitmaps used to display the status light.

If the Glyph property is not assigned, a “lit” status light is drawn as a solid red square and an “unlit” one as a green square. Each square has shadowed edges to give it a three dimensional appearance.

Glyph can be used to display custom bitmaps instead. The Glyph bitmap is actually two bitmaps in one. It should be twice as wide as the component’s Width property. The bitmap displayed when the status light is lit is Width pixels wide starting at the left edge of the bitmap. The bitmap displayed when the status light is not lit is Width pixels wide starting Width pixels from the left edge of the bitmap.

Lit

property

```
property Lit : Boolean
```

Default: False

↪ Determines the state in which the status light is drawn.

This property is normally assigned by the trigger handlers of the TApdSLController component.

LitColor

property

```
property LitColor : TColor
```

Default: clRed

↪ Determines the color of a no-glyph status light in its lit state.

If the component’s Glyph property is not set to a valid bitmap, the status light is drawn as a slightly raised colored square. LitColor is the color in which that square is drawn when the light is lit.

property NotLitColor : TColor

Default: clGreen

↪ Determines the color of a no-glyph status light in its unlit state.

If the component's Glyph property is not set to a valid bitmap, the status light is drawn as a slightly raised colored square. NotLitColor is the color in which that square is drawn when the light is not lit.

TApdSLController Component

The TApdSLController component monitors the status of a TApdComPort component and changes the state of one or more TApdStatusLight components to reflect that status. The goal of the component is to give communications programs a status light display similar to the LEDs found on external modems.

TApdSLController is capable of monitoring the port's line signals (DCD, DTR, CTS, and RI), line breaks and errors, and whether data is currently being received or transmitted.

The Lights property

The controller has a property called Lights which holds pointers to the status light components that the controller will be monitoring. Lights is of type TLightSet which is simply a class that contains a property of type TApdCustomStatusLight for each line condition that TApdSLController can monitor. In the Object Inspector, the Lights property has eight sub-properties that are used to assign status lights to the line conditions you want to monitor.

Table 7.1 provides a list of all Lights sub-properties and the port condition they monitor.

Table 7.1: *Lights property sub-properties and port conditions*

Sub-property	Line condition
BREAKLight	Lit for BreakOffTimeout ticks when a line break occurs.
CTSLight	Lit when CTS signal high.
DCDLight	Lit when DCD signal high.
DSRLight	Lit when DSR signal high.
ERRORLight	Lit for ErrorOffTimeout ticks when a line error occurs.
RINGLight	Lit for RingOffTimeout ticks after RI signal goes high.
RXDLight	Lit when data is being received.
TXDLight	Lit when data is being transmitted.

Using a TApdSLController

To use a TApdSLController, first create a TApdStatusLight component for each line condition you want the controller to monitor. Next, drop a TApdSLController component on the form and link it to the TApdComPort component you wish to monitor. Next, link the controller's light properties to the status light components. Set the controller's Monitoring property to True at run time when you want to start monitoring.

Figure 7.1 shows the Object Inspector for a properly created TApdSLController component that can be used for monitoring the CTS, DCD, and DSR line signals.



Figure 7.1: Viewing the Object Inspector of a properly created TApdSLController component.

When the CTS signal changes, the component named CTSMonitor is changed accordingly. Similarly, the component named DSRMonitor changes when the DSR signal changes and the DCDMonitor component changes when the DCD signal changes.

Hierarchy

TComponent (VCL)

 TApdCustomSLController (AdStatLt)

 TApdSLController (AdStatLt)

Properties

BreakOffTimeout

ComPort

ErrorOffTimeout

Lights

Monitoring

RingOffTimeout

RXDOffTimeout

TXDOffTimeout

Reference Section

BreakOffTimeout **property**

`property BreakOffTimeout : LongInt`

Default: 36

- ↪ Determines the number of ticks the BREAKLight remains lit after a line break is detected.

ComPort **property**

`property ComPort : TApdCustomComPort`

- ↪ Determines the serial port monitored by the status lights.

The status light controller monitors the status of lines on a single serial port. This property must be linked to the TApdComPort component that will be monitored.

ErrorOffTimeout **property**

`property ErrorOffTimeout : LongInt`

Default: 36

- ↪ Determines the number of ticks the ERRORLight remains lit after a line error is detected.

Lights **property**

`property Lights : TLightSet`

- ↪ Determines all of the lights displayed in the status bar.

Lights is simply a class that contains properties for each port condition that TApdSLController can monitor. In the Object Inspector, these individual light properties show up as sub-properties. That is, the inspector shows a property called Lights which will expand into a list of individual properties when double-clicked.

For more information, see “The Lights property” on page 191.

Monitoring

run-time property

```
property Monitoring : Boolean
```

- ↳ Determines whether the status lights are being updated.

Setting Monitoring to True causes the status light controller to install various triggers that are used to update the status lights. Setting it to False removes those triggers.

ComPort must be assigned before setting Monitoring to True at run time.

RingOffTimeout

property

```
property RingOffTimeout : LongInt
```

Default: 8

- ↳ Determines the number of ticks the RINGLight remains lit after a ring is detected.

7

RXDOffTimeout

property

```
property RXDOffTimeout : LongInt
```

Default: 1

- ↳ Determines the number of ticks the RXDLight remains lit after a character is received.

TXDOffTimeout

property

```
property TXDOffTimeout : LongInt
```

Default: 1

- ↳ Determines the number of ticks the TXDLight remains lit after a character is transmitted.

Chapter 8: The Terminal Components

Traditionally, a terminal is a piece of hardware with a screen and keyboard that provides a method to display information from a host computer and to enter information into the same computer. Today, more often than not, terminals are personal computers that run a program that emulates the original terminal; it interprets the same data in the same fashion and displays it in the same way. Similarly, it emulates the original keyboard and sends the same kind of information back to the host computer in the same format.

The data presented by the host computer takes one of two forms. The first form is intended for display only. The terminal will just send the stream of data directly to the screen without trying to interpret it in any fashion. What you see is what is sent. This is known as *teletype mode* (abbreviated as TTY).

The other form of data sent to a terminal consists of two types, intermixed: displayable data and embedded terminal control sequences. The Terminal control sequences cause the terminal to move the cursor around the screen, to block off certain parts of the display from being altered, to delete text from areas of the screen, to scroll the display, to switch character sets, and so on. The terminal has to monitor the data being sent by the host computer, be able to identify the terminal control sequences in the stream, and then extract and act on them. All other data would be sent to the screen as usual. We usually refer to this mode as the terminal emulation mode.

All emulation means is that the PC is pretending to be the terminal in such a fashion that the host computer is not aware that there is no real terminal present, just a clever program. Ideally, the PC program emulates the terminal so well that the user cannot tell the difference between the program's window and the original display. What you see is what was intended. Unfortunately, the same cannot be said of the keyboard; terminal keyboards generally have extra keys not available on a PC keyboard, and so the emulation program has to map available PC keystrokes to the original terminal keyboard keys, and obviously the user has to be aware of these mappings.

There are a variety of standards for terminal control sequences from such companies as IBM, Digital Corporation (DEC), Wyse, and so on. One of the most widely known is the DEC VT100 standard, which helped form the basis for the ANSI standard. Async Professional implements the full VT100 standard, including support for different character sets, scrolling regions, and keyboard application modes. By definition, this is also a subset of the ANSI standard. Also, since the vast majority of PCs running Windows have color monitors, the Async Professional VT100 emulation also supports the terminal control sequences needed to display text in color.

Terminal Design Considerations

Async Professional provides terminal emulation capabilities with its terminal component, TAdTerminal and its emulators, all descended from TAdTerminalEmulator. These components have been designed with two goals in mind: first, to be usable with the minimum of effort on your part; and, second, to be flexible so that you can extend them to work with other terminal types.

The former goal is important to those programmers who know that having a terminal that is compliant with the VT100 standard (or a subset of the ANSI standard) is all that is required for their application. They do not want to worry about linking this kind of display emulation with that kind of keyboard emulation to the terminal component, they would rather just point the emulator at the terminal component and let the components do the rest.

Catering for the latter goal, extensibility, is more difficult. There are many facets to emulating a terminal. You have to worry about how to store the data from the host; in other words, the actual displayable characters, the character attributes, the colors (both foreground and background), the character sets. You have to worry about interpreting the terminal control sequences being sent by the host and acting upon them. These terminal control sequences may erase parts of the display, insert extra lines, or scroll the text in different ways. You have to track these changes in your data.

Once you have those kind of problems nailed down, you then have to consider the keyboard. Ideally, you would like to emulate the original terminal keyboard completely, but this isn't generally possible. Apart from the alphabetic part of the keyboard, terminal keyboards have different keys. You must map some PC keys (maybe normal, Control or Alt shifted) onto their terminal keyboard equivalents. You have to consider what control sequence the terminal keyboard would send to the host when a given key was pressed.

After all that, you have to actually draw the terminal display in a normal window on the screen. With the terminal classes in Async Professional, the attempt has been made to simplify this entire process (see Figure 8.1).

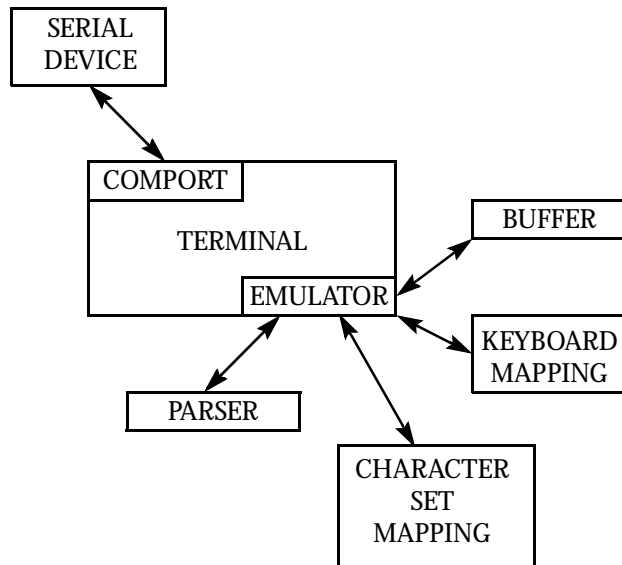


Figure 8.1: Terminal design example diagram.

Terminal buffer

First, there is the TAdTerminalBuffer class. This class provides a non-visual buffer containing a representation of the data being displayed by the terminal window. The buffer is organized in the same fashion as the terminal display: a matrix of data organized into rows and columns. In fact, the buffer maintains a set of such matrices: one for the displayable characters, one for the attributes (bold, underline, invisible, and so on), one for the color of the text, one for the color of the background, and finally one for the character set identifiers. There is a set of methods for performing the standard terminal operations: erasing or deleting part of the display, inserting rows and characters, scrolling areas of the display, changing the default colors and attributes, writing new characters to the screen, and so on. The class also maintains a list of character positions and rectangles on the screen that need repainting. This class does not of itself understand any terminal control sequences; its functionality is driven by some controlling terminal emulation component. In writing a new terminal emulation component, you would generally use this buffer class as is; the only time you would create your own descendant would be if the terminal you are emulating requires some transformation of the display not provided by the standard class.

Terminal parser

Next, there is the `TAdTerminalParser` class. This class is designed to interpret terminal control sequences. The `TAdTerminalParser` class is an ancestor class, it merely defines the interface through which a parser should work. Async Professional provides a single descendant, the `TAdVT100Parser` class, which provides parsing capabilities for the VT100 series of terminals. Studying the code for this class would provide hints on how to write a specialized parser class descendant for another terminal.

Keyboard-mapping table

Next, there is the keyboard-mapping table. This class stores the information required to map a PC keystroke (as reported by Windows) into the control sequence that has to be sent to the terminal's host computer. This process takes place in three distinct steps, the information for each being stored in the same structure (actually, a hash table). First, the virtual key code reported by Windows is looked up in the table. If found, the name of the virtual key would be returned. This name, together with the state of the various shift keys (including Ctrl and Alt), is then looked up in the same table. If found, it will return the name of the terminal key to which the PC key maps. For example, with the standard VT100 mappings provided with Async Professional, the up arrow key on the PC keyboard, `VK_UP`, is mapped to `DEC_UP`, the name for the up arrow on the VT100 keyboard. Finally, the terminal key name is looked up in the same table in order to obtain the control sequence that needs to be sent to the host computer.

This seemingly repetitive triple lookup process exists to enhance comprehension and extensibility. Consider: the first and last lookups are fully determined by the Windows operating system and the terminal definition. They should not be altered. Virtual key code \$26 is `VK_UP`, by definition, and the VT100 up arrow key will send either `<Esc>[A` or `<Esc>OA` depending on the terminal mode, again by definition. Thus, you could specify that virtual key code \$26 will send either `<Esc>[A` or `<Esc>OA`, but it wouldn't help with understanding the mapping later on; whereas a mapping of `VK_UP` to `DEC_UP` is immediately understandable and is easily changed. For example, if you wanted to make F6 the same as the up arrow key, you could just add the mapping of `VK_F6` to `DEC_UP`. You don't have to manually look up the virtual key code for F6, nor the escape sequence generated by the VT100 up arrow key to write the mapping.

For convenience, the keyboard-mapping table can be loaded in two ways. First, the mappings can be read from a simply formatted text file. Second, the mappings can be stored in a resource linked into your application, from which the class can easily load them. The latter method has the advantage that no external files are required to define keyboard mappings. To aid in the generation of the resource file, the class has a method by which a binary file containing the mappings can be written. This binary file can then be compiled into a resource and linked into your application.

Character set mapping table

Next there is the character set mapping table. This class stores the information required to actually display a given character on the terminal display. This class simplifies the task of creating one terminal that uses several different character sets for displaying text. For each character set, there may be a different glyph for each character. The classic example is that of the VT100 terminal. This terminal has several character sets, of which two are the most commonly used: the USASCII character set and the special graphics character set. To take as an example, the character 'm' is displayed as a lower-case m in the USASCII character set, but is displayed as a line draw lower left corner (the glyph that looks like an L) in the special graphics character set.

The character set mapping table is a list of character ranges in character sets and the fonts and glyphs that should be used to display them. To aid in the display of text on the terminal keyboard—and to make the process more efficient—the character set mapping table will analyze a string to be displayed in a particular character set, and generate a script of font changes and the strings to be displayed in those fonts.

For convenience, the character set mapping table can be loaded in two ways. First, the mappings can be read from a simply formatted text file. Second, the mappings can be stored in a resource linked into your application, from which the class can easily load them. The latter method has the advantage that no external files are required to define keyboard mappings. To aid in the generation of the resource file, the class has a method by which a binary file containing the mappings can be written. This binary file can then be compiled into a resource and linked into your application.

Terminal emulator

The final step in writing your own terminal is possibly the most intricate: writing a terminal emulator component. This component will have its own buffer instance, it will have a parser instance and it will use a keyboard-mapping table and a character set mapping table. The emulator would write to a TAdTerminal window and would be passed incoming data and keyboard data from that component. The sequence of events for displaying something on the screen would go something like this:

1. Accept a character from the input data stream.
2. Pass the character to the parser. The parser would decide whether the character was displayable or part of a control sequence (a command) and pass the result back.
3. If the parser indicated that the character was displayable, the component would pass the character to the buffer (equivalent to writing it to the display).

4. If the parser indicated that the character was a command, the terminal component would act on the command. This may be as simple as sending something back to the server or as complex as scrolling the display. The latter operation would be passed to the buffer to do.
5. Every now and then, the terminal component would get a paint message, at which point it would have to interrogate the buffer to find the parts of the display that need repainting, and draw them in its client area.

As far as the keyboard goes, the terminal component would convert the virtual key codes into control sequences in the manner already described and send these sequences to the server. For the majority of the keys on the keyboard, the letter keys, no translation is required and the relevant character is sent as is.

TAdTerminalBuffer Component

The TAdTerminalBuffer class defines a data structure for maintaining the data required for a communications terminal display. Essentially, this consists of:

- The characters that should be shown.
- The character sets from which those character glyphs are drawn.
- The color in which the character glyphs will be displayed.
- The color for the background behind the characters.
- A set of display attributes for the characters.

It is important to realize that the terminal buffer class is only a data structure; it has no responsibilities for the display of the data it stores. That work is delegated to the terminal and emulator components. In essence, an emulator component creates an instance of a terminal buffer to store the data that it will display on the screen. The emulator component will call methods of the terminal buffer to perform such tasks as writing characters, inserting and deleting lines, scrolling the data, changing colors and attributes, and the like. These tasks will, in theory, be performed in response to control sequences being received by the terminal from some remote program or host and being decoded by an emulator. Hence the methods of the terminal buffer class mimic the standard operations performed by all character-based terminals, but in particular by the DEC VT100 terminals.

Conceptually, the terminal buffer class manages five different screens worth of information as independent matrices. These matrices are representations of the terminal screen and hence are divided into rows and columns. One of the matrices holds the characters that should be displayed on the screen. Another, holds the foreground color (i.e., the color of the text), and yet another, holds the background color. A fourth holds the character set from which the characters are drawn. The final matrix holds the special display characteristics for the text, whether it is bold, underlined, blinking and so on. Using this scheme it is therefore possible for every addressable character cell on the terminal display to be a different character, from different character sets, using different colors and attributes. The character matrix stores either a single byte per character cell (ANSI mode) or two bytes per character (UNICODE mode), the choice being made when the terminal buffer is created. The identifier for the character set is assumed to be limited to one of 256 possible values, and hence just a single byte is stored per character cell to hold this information. The colors are assumed to be stored as RGB values, and hence each character cell has a 4-byte long integer color value for both the foreground color and the background color. (Notice that the terminal buffer does not enforce the rule that the colors have to be RGB values, you could store standard values from the 4-bit color set instead if you wished—the terminal buffer just

assumes that a color is a 4-byte quantity.) The attributes are stored as a set of different possibilities: bold, underlined, strikethrough, blinking, reverse or invisible.

A word is required here regarding character sets. The VT100 terminal was a 7 bit device. In theory, characters could have ASCII values from 0 to 127; however, the first 32 characters in this set were control characters rather than displayable characters. Hence, there were only 96 different characters that could be displayed. Since this was not sufficient, the VT100 had other sets of 96 characters that could be switched in and out, the most distinctive being the line-draw characters in the special graphics character set. To display a character at a particular position on the screen the terminal had to know which character set to take the displayed glyph from, and also the ASCII value of the character. (A *glyph* is the visual representation of an ASCII value in a particular character set.) Thus, on a VT100, the ASCII value 123 would be displayed as the left curly brace, {, if the standard character set was in force, or the symbol for pi, π , if the special graphics character set was in force.

Another point should to be made about the terminal buffer: it does not parse any data stream for control sequences. This is the job of the emulator portion of a terminal component. The emulator should scan the incoming data stream for control sequences (for example, those beginning with “<Esc>[”), decode them, and then get the terminal buffer to update itself according to the operation requested.

Apart from the data to be displayed, a terminal component requires two basic pieces of information: Has the cursor moved? What needs to be redisplayed? The terminal buffer maintains a cursor position variable: the row and column number where new characters are to be written should they arrive. Various methods update the cursor position, the most obvious ones being the cursor movement methods to move the cursor up, down, left or right. Again, it should be stressed that there is no visual representation of a cursor in a terminal buffer object: the cursor is just a pair of row and column values. It is up to the terminal component displaying the data from the buffer to maintain and show a cursor, for example, perhaps a blinking underscore or block. (By the way, this can get confusing: with terminals the blinking point where editing takes place is called a *cursor*, whereas with Windows programs, it is known as a *caret*, the cursor being the mouse pointer.) The terminal buffer not only maintains a cursor position, it also interfaces a routine that returns whether the cursor position has changed since the last time the routine was called.

If the terminal component is to have a chance of keeping up with fast data streams, it cannot continually redisplay the entire terminal screen every now and then. It must have a way of knowing what has changed on the terminal screen, so that it only needs to update that particular section. For example, if a character was written to the terminal, the terminal component needs to know how much of its window it needs to repaint. In the majority of cases, that's a single character cell; the new character, in other words. In some cases, the terminal data needs to be scrolled before the character can be written. In this case the terminal display needs to repaint a lot more of the window to show the effect of writing the character. The terminal buffer maintains an internal list of invalidated character cells (in

fact, as cell rectangles) and the terminal component can read these and use the information to redisplay parts of its window.

The terminal buffer has two views of the data: the *scrollback view* and the *display view*. The scrollback view shows a history of data that have scrolled off the top of the display view. The display view is essentially the representation of the terminal screen itself. There are typically more rows in the scrollback view than in the display view (otherwise there wouldn't be anything to scroll back through). The number of columns in both the scrollback and display views is however the same.

The user of the terminal buffer will refer to positions on the terminal screen as one-based values. For example, on a 24 rows x 80 columns terminal, the rows are numbered from 1 to 24 and the columns from 1 to 80. This is different from the usual Windows way of looking at things, where values are counted from 0 instead. The rows in the scrollback view are negative numbers or zero. For example, the last row to be scrolled off the display view (i.e., the terminal itself) will be numbered row 0, the next to last, row -1, and so on. If a 24 row terminal screen is cleared or erased, the entire scrollback view is scrolled up with the previous display, and the rows of this previous display would then be numbered -23 to 0.

The terminal buffer also supports the concept of a scrolling region. This is a region of the terminal screen to which the cursor is restricted. Once a scrolling region is defined and activated, all cursor movement is restricted to this area and characters can only be written to this area. When a scrolling region is activated, row and column numbers are no longer absolute values, counted from the top left character cell of the screen, but are now relative values, counted from the top left character cell of the scrolling region. For example, suppose a scrolling region is defined to be within rows 5 and 10 and it is activated. Moving the cursor to row 1, column 1, results in it being moved to the home position of the scrolling region, not the home position of the screen. In absolute terms, the cursor ends up at row 5, column 1 instead. The terminal buffer maintains a definition of a scrolling region and also a flag that states whether the scrolling region is in force or not. If a new scrolling region is defined, the activation flag is automatically set, in other words, the scrolling region becomes effective immediately.

Generally, when a data stream is sent to a terminal, the sender does not send coloring or attribute information with every single character that needs to be displayed. Instead, the terminal is instructed to use a particular color or attribute from that point forward, until another color or attribute is requested. Characters written to the terminal in between these two instructions will automatically be in the requested color or have the requested attribute. The terminal also has a reset state, with default colors and attributes. The terminal buffer mimics this behavior by having both current values and default values for the colors, attribute and character set. If the terminal is reset, the terminal buffer automatically sets the current values equal to the default values.

Hierarchy

TObject (VCL)

 TAdTerminalBuffer (ADTrmBuf)

Properties

BackColor	DefForeColor	SVRowCount
Charset	ForeColor	UseAutoWrap
Col	OriginCol	UseNewLineMode
ColCount	OriginRow	UseScrollRegion
DefBackColor	Row	UseWideChars
DefCharset	RowCount	

Methods

ClearAllHorzTabStops	EraseFromBOL	InsertLines
ClearAllVertTabStops	EraseLine	MoveCursorDown
ClearHorzTabStop	EraseScreen	MoveCursorLeft
ClearVertTabStop	EraseToEOL	MoveCursorRight
Create	EraseToEOW	MoveCursorUp
DeleteChars	GetCharAttrs	Reset
DeleteLines	GetDefCharAttrs	SetCharAttrs
DoBackHorzTab	GetInvalidRect	SetCursorPosition
DoBackspace	GetLineAttrPtr	SetDefCharAttrs
DoBackVertTab	GetLineBackColorPtr	SetHorzTabStop
DoCarriageReturn	GetLineCharPtr	SetScrollRegion
DoHorzTab	GetLineCharSetPtr	SetVertTabStop
DoLineFeed	GetLineForeColorPtr	WriteChar
DoVertTab	HasCursorMoved	WriteString
EraseChars	HasDisplayChanged	
EraseFromBOW	InsertChars	

Reference Section

BackColor

property

```
property BackColor : TColor
```

Default: clBlack

↳ Defines the background color for succeeding text.

The BackColor property determines the background color that will appear behind text displayed in the terminal. Once the background color has been set, any new text written to the terminal will appear with this color until the background color is set to another value.

See also: DefBackColor, DefForeColor, ForeColor

CharSet

property

```
property CharSet : Byte
```

Default: 0

↳ Defines the character set from which characters are drawn.

The CharSet property determines the character set from which a character will be drawn. Once a new character set is set, all characters written to the display should be displayed with glyphs from this character set.

The terminal buffer class imposes no structure or valid values to a character set. It is up to the terminal emulator component to impose meaning to the different character set values (e.g., by assuming the value 0 means “default character set”).

See also: DefCharset

ClearAllHorzTabStops

method

```
procedure ClearAllHorzTabStops;
```

↳ Removes all horizontal tab stops.

The terminal buffer does not have any default tab stops (for example, a tab stop every 8 characters).

See also: ClearHorzTabStop, SetHorzTabStop

ClearAllVertTabStops

method

```
procedure ClearAllVertTabStops;
```

✚ Removes all vertical tab stops.

The terminal buffer does not have any default vertical tab stops.

See also: ClearVertTabStop, SetVertTabStop

ClearHorzTabStop

method

```
procedure ClearHorzTabStop;
```

✚ Clears a horizontal tab stop at the current cursor column.

If no horizontal tab stop is set for this column, ClearHorzTabStop does nothing.

See also: ClearAllHorzTabStop, SetHorzTabStops

ClearVertTabStop

method

```
procedure ClearVertTabStop;
```

✚ Clears a vertical tab stop at the current cursor row.

If no vertical tab stop is set for this row, ClearVertTabStop does nothing.

See also: ClearAllVertTabStops, SetVertTabStop

Col

property

```
property Col : Integer
```

✚ Defines the column of the cursor.

The Col property refers to the cursor. Reading the Col property returns the column number of the cursor, and setting it moves the cursor to that column in the current row.

The value used for the Col property is one-based; in other words, columns are counted from 1. If there are 80 columns across the terminal screen, the columns will be known as columns 1 to 80. If the column number used is out of the range of the screen or the current scrolling region, it is forced silently into bounds.

If a scrolling region is activated, the values for Col will be relative to the home position of the scrolling region, not the home position of the screen itself.

See also: Col, ColCount, SetCursorPosition, UseScrollingRegion

ColCount

property

```
property ColCount : Integer
```

Default: 80

↳ Defines the number of columns across the terminal screen.

The ColCount property is the number of columns displayed by the terminal screen. For the VT100 terminal, for example, there are two possible values: 80 and 132.

If the ColCount property is changed, it is checked to be at least 2; otherwise, an exception is raised.

Changing the ColCount property for an existing terminal screen does not clear the data being displayed by the screen; you will need to do this as a separate step. The positions of any horizontal tab stops are maintained, except for those that lie outside the new value for ColCount. However, any scrolling region is discarded, and the cursor is reset to the home position of the screen.

See also: Col, RowCount

Create

method

```
constructor Create(aUseWideChars : Boolean);
```

↳ Creates an instance of the TAdTerminalBuffer class.

The aUseWideChars parameter, if True, determines whether the Terminal is configured to work with UNICODE characters, and allocates the underlying character matrix accordingly.

DefBackColor

property

```
property DefBackColor : TColor
```

Default: clBlack

↳ Defines the default background color.

If Reset is called, the current background color is set to the value of DefBackColor.

See also: BackColor, DefForeColor, Reset

DefCharset

property

```
property DefCharset : Byte
```

Default: 0

↳ Defines the default character set value.

If Reset is called, the current character set is set to the value of DefCharset.

See also: CharSet, Reset

DefForeColor

property

```
property DefForeColor : TColor
```

Default: clSilver

↳ Defines the default color to be used for displaying text.

If Reset is called, the current foreground color is set to the value of DefForeColor.

See also: DefBackColor, ForeColor, Reset

8

DeleteChars

method

```
procedure DeleteChars(aCount : Integer);
```

↳ Deletes characters from the current cursor position.

The characters to the right of the deleted ones are moved over to take their place. The area on the extreme right uncovered by this action is filled with space characters using the current colors and attributes.

The DeleteChars method is limited to the current scrolling region.

The cursor is left in the same position.

DeleteLines

method

```
procedure DeleteLines(aCount : Integer);
```

↳ Deletes lines from the current cursor position.

The lines underneath the lines being deleted are moved up to take their place. The area at the bottom uncovered by this action is filled with space characters using the current colors and attributes. Deleting aCount lines is equivalent to scrolling up aCount lines.

The DeleteLines method is limited to the current scrolling region.

The cursor is left in the same position.

DoBackHorzTab

method

```
procedure DoBackHorzTab;
```

↳ Moves the cursor left to the previous tab stop.

If there is no previous tab stop, the cursor is moved to the first column of the line. If it already at this position, it is not moved.

This method is limited to the current scrolling region.

See also: DoHorzTab, SetHorzTabStop

DoBackspace

method

```
procedure DoBackspace;
```

↳ Backspaces the cursor.

The character underneath the new position of the cursor is not erased by this operation. A backspace is equivalent to moving the cursor one position to the left, except that, if the cursor is at the beginning of the row, it is not moved at all.

This method is limited to the current scrolling region.

See also: MoveCursorLeft

DoBackVertTab

method

```
procedure DoBackVertTab;
```

↳ Moves the cursor up to the previous tab stop.

If there is no previous tab stop, the cursor is moved to the first row. If it already at this position, it is not moved.

This method is limited to the current scrolling region.

See also: DoVertTab, SetVertTabStop

DoCarriageReturn

method

```
procedure DoCarriageReturn;
```

↳ Moves the cursor to the beginning of the current row.

This method is limited to the current scrolling region.

See also: DoLineFeed

DoHorzTab

method

```
procedure DoHorzTab;
```

↩ Moves the cursor right to the next tab stop.

If there is no next tab stop, the cursor is moved to the last column of the line. If it already at this position, it is not moved.

This method is limited to the current scrolling region.

See also: DoBackHorzTab, SetHorzTabStop

DoLineFeed

method

```
procedure DoLineFeed;
```

↩ Moves the cursor down one row.

This method has two modes of operation, distinguished by the value of UseNewLineMode. If UseNewLineMode is False, the cursor is moved down a row, staying in the same column. If the cursor is in the bottom row when this happens, the screen or scrolling region is scrolled up. The new row so formed is initialized to space characters using the current colors and attributes.

If UseNewLineMode is True, the cursor is moved down a row in the manner already described, except this time it is moved to the first column.

This method is limited to the current scrolling region.

See also: DoCarriageReturn

DoVertTab

method

```
procedure DoVertTab;
```

↩ Moves the cursor down to the next tab stop.

If there is no next tab stop, the cursor is moved to the last row. If it already at this position, it is not moved.

This method is limited to the current scrolling region.

See also: DoBackVertTab, SetVertTabStop

EraseChars**method**

```
procedure EraseChars(aCount : Integer);
```

↳ Erases characters from the current cursor position.

The erasing operation is done by replacing aCount characters with space characters, using the current colors and attributes. The EraseChars method will automatically wrap and continue at the end of rows, but it will not cause a scroll if the count requested exceeds the bottom row. The cursor position is included.

This method is limited to the current scrolling region.

EraseFromBOL**method**

```
procedure EraseFromBOL;
```

↳ Erases characters from the beginning of the row to the current cursor position.

The erasing operation is done by replacing the characters with space characters, using the current colors and attributes. The cursor position is included.

This method is limited to the current scrolling region.

EraseFromBOW**method**

```
procedure EraseFromBOW;
```

↳ Erases characters from the beginning of the screen to the current cursor position.

The erasing operation is done by replacing the characters with space characters, using the current colors and attributes. The cursor position is included.

This method is not limited to the current scrolling region; it applies to the whole screen.

EraseLine**method**

```
procedure EraseLine;
```

↳ Erases the current row.

The erasing operation is done by replacing the characters with space characters, using the current colors and attributes.

This method is limited to the current scrolling region.

EraseScreen

method

```
procedure EraseScreen;
```

↳ Erases the entire screen.

The erasing operation is done by replacing the characters with space characters, using the current colors and attributes.

This method is not limited to the current scrolling region; it applies to the whole screen. The screen is erased by scrolling the entire scrollbar view by RowCount rows. This has the effect of erasing the screen, but also saves the previous version of the screen in the scrollbar area.

EraseToEOL

method

```
procedure EraseToEOL;
```

↳ Erases characters from the current cursor position to the end of the row.

The erasing operation is done by replacing the characters with space characters, using the current colors and attributes. The cursor position is included.

This method is limited to the current scrolling region.

EraseToEOW

method

```
procedure EraseToEOW;
```

↳ Erases characters from the current cursor position to the end of the screen.

The erasing operation is done by replacing the characters with space characters, using the current colors and attributes. The cursor position is included.

This method is not limited to the current scrolling region; it applies to the whole screen.

ForeColor

property

```
property ForeColor : TColor
```

Default: clSilver

↳ Defines the foreground color for new text.

The ForeColor property determines the color in which new text will be displayed in the terminal. Any new text sent to the Terminal will appear in this color.

See also: DefForeColor

```

procedure GetCharAttrs(var aValue : TAdTerminalCharAttrs);

TAdTerminalCharAttr = (tcaBold, tcaUnderline,
    tcaStrikethrough, tcaBlink, tcaReverse, tcaInvisible);

TAdTerminalCharAttrs = set of TAdTerminalCharAttr;

```

↳ Returns the current set of display attributes.

The display attributes define the style of new text. The attributes are bold, underlined, strikethrough, blink, reverse image (i.e., the text displayed in BackColor, the background in ForeColor), and invisible.

If you wish to add a new attribute to the current set, write code like this:

```

var
    Attrs : TAdTerminalCharAttrs;
begin
    ...
    MyBuffer.GetCharAttrs(Attrs);
    Attrs := Attrs + [tcaUnderline];
    MyBuffer.SetCharAttrs(Attrs);

```

See also: GetDefCharAttrs, SetCharAttrs

```

procedure GetDefCharAttrs(var aValue : TAdTerminalCharAttrs);

TAdTerminalCharAttr = (tcaBold, tcaUnderline,
    tcaStrikethrough, tcaBlink, tcaReverse, tcaInvisible);

TAdTerminalCharAttrs = set of TAdTerminalCharAttr;

```

↳ Returns the default set of display attributes.

The display attributes define the style of new text. The attributes are bold, underlined, strikethrough, blink, reverse image (i.e., the text displayed in BackColor, the background in ForeColor), and invisible. If Reset is called, the current display attributes are set to the value returned by GetDefCharAttrs.

See also: GetCharAttrs, Reset, SetDefCharAttrs


```
function GetInvalidRect(var aRect : TRect) : Boolean;
```

↳ Returns the next invalid rectangle.

An invalid rectangle is a TRect structure that defines an area of the terminal screen that needs repainting. The fields of the record, Left, Top, Right, and Bottom, define the area, not in pixels, but in row and column numbers. Thus the rectangle (1, 2, 3, 4) describes the area from the top left hand corner at row 2, column 1, to the bottom right hand corner at row 4, column 3. The row and column numbers are absolute values and do not depend on the current scrolling area, even if it activated. They are one-based values.

The return value is False if there was no invalid rectangle; otherwise, it is True and aRect has the first invalid rectangle. The intent of this method is for the terminal display component to continually call this method, repainting the areas of the screen affected, until the method returns False.

Every change to the terminal is recorded by the terminal buffer as a list of invalid rectangles. Conceivably, there could be very many by the time the terminal emulator component comes to repainting. Rather than have the display component continually call this method to get the full list of invalid rectangles, if there are a large number, the buffer uses an optimization whereby it will merge all of the invalid rectangles into the one smallest rectangle that overlaps all of the others. This is done automatically.

See also: HasCursorMoved, HasDisplayChanged

GetLineAttrPtr

method

```
function GetLineAttrPtr(aRow : Integer) : pointer;
```

↳ Returns a pointer to the first element of the display attributes array.

Each element in the display attributes array is a TAdTerminalCharAttrs set.

This method is not limited to the current scrolling region; it applies to the whole screen. Thus, the pointer that is returned points to the data for column 1 of the required row.

GetLineBackColorPtr

method

```
function GetLineBackColorPtr(aRow : Integer): pointer;
```



Returns a pointer to the first element of the background colors array.

Each element in the background colors array is of type TColor.

This method is not limited to the current scrolling region; it applies to the whole screen. Thus, the pointer that is returned points to the data for column 1 of the required row.

See also: BackColor, DefBackColor, GetLineForeColorPtr

GetLineCharPtr

method

```
function GetLineCharPtr(aRow : Integer): pointer;
```



Returns a pointer to the first element of the characters array.

Each element in the characters array is either a single byte character, or a two-byte UNICODE character. The original call to the Create constructor defines which it is.

This method is not limited to the current scrolling region; it applies to the whole screen. Thus, the pointer that is returned points to the data for column 1 of the required row.

GetLineCharSetPtr

method

```
function GetLineCharSetPtr(aRow : Integer): pointer;
```



Returns a pointer to the first element of the character set array.

Each element in the character set array is a byte. It is the responsibility of the owning terminal display component to interpret the values.

This method is not limited to the current scrolling region; it applies to the whole screen. Thus, the pointer that is returned points to the data for column 1 of the required row.

GetLineForeColorPtr

method

```
function GetLineForeColorPtr(aRow : Integer): pointer;
```



Returns a pointer to the first element of the foreground colors array.

Each element in the foreground colors array is of type TColor.

This method is not limited to the current scrolling region; it applies to the whole screen. Thus, the pointer that is returned points to the data for column 1 of the required row.

See also: DefForeColor, ForeColor, GetLineBackColorPtr

HasCursorMoved

method

```
function HasCursorMoved : Boolean;
```

↳ Returns whether the cursor has moved.

The terminal buffer maintains an internal flag that notes whether the cursor has moved at any time. This method returns the value of this flag and then resets it to False. Hence, if the return value is False, the cursor has not moved since the last time the method was called, and if it returns True, the cursor has moved.

HasDisplayChanged

method

```
function HasDisplayChanged : Boolean;
```

↳ Returns whether the terminal has changed in appearance.

This method is a handy shortcut to be used instead of calling `GetInvalidRect` and getting False.

See also: `GetInvalidRect`

8

InsertChars

method

```
procedure InsertChars(aCount : Integer);
```

↳ Inserts new chars at the cursor.

This method is equivalent to hit the space bar. The new chars inserted are initialized to space characters, using the current colors and attributes. The chars that are advanced out of view are deleted.

The number of characters to insert is `aCount` and is constrained by the current display region.

InsertLines

method

```
procedure InsertLines(aCount : Integer);
```

↳ Inserts new lines at the cursor.

This method is equivalent to scrolling down the screen. The new rows inserted are initialized to space characters, using the current colors and attributes. The rows that are scrolled out of view are deleted.

The cursor is left in the same location.

This method is limited to the current scrolling region.

```
procedure MoveCursorDown(aScroll : Boolean);
```

↳ Moves the cursor down one row.

The cursor is moved onto the next row at the same column position. If the aScroll parameter is False and the cursor is currently on the bottom row, it is not moved. If the aScroll parameter is True and the cursor is on the bottom row, the screen or scrolling region is scrolled up one line and the cursor remains in the same position. The new line inserted at the bottom is initialized to space characters, using the current colors and attributes.

This method is limited to the current scrolling region.

```
procedure MoveCursorLeft(aWrap : Boolean; aScroll : Boolean);
```

↳ Moves the cursor left one position.

The column number of the cursor is decremented, unless the cursor is at the first position of the row.

If, in fact, the cursor is at the row's home position, the values for aWrap and aScroll come into play. If aWrap is False, the cursor does not move. If aWrap is True, the cursor moves up one row, and is positioned at the last column of that previous row. If, furthermore, the cursor was originally at the first column of the top row and aScroll was False, the cursor does not move. If, on the other hand, aScroll was True, the screen or scrolling region is scrolled down one row and the cursor then moved to the last column of the top row. The new row is initialized to space characters, using the current colors and attributes.

This method is limited to the current scrolling region.

See also: Col, MoveCursorRight, SetCursorPos

```
procedure MoveCursorRight(aWrap : Boolean; aScroll : Boolean);
```

↳ Moves the cursor right one position.

The column number of the cursor is incremented, unless the cursor is at the last position of the row.

If, in fact, the cursor is at the row's last position, the values for aWrap and aScroll come into play. If aWrap is False, the cursor does not move. If aWrap is True, the cursor moves down one row, and is positioned at the first column of that next row. If, furthermore, the cursor was originally at the last column of the bottom row and aScroll was False, the cursor does

not move. If, on the other hand, `aScroll` was `True`, the screen or scrolling region is scrolled up one row and the cursor then moved to the first column of the bottom row. The new row is initialized to space characters, using the current colors and attributes.

This method is limited to the current scrolling region.

See also: `MoveCursorLeft`, `Col`, `SetCursorPos`

MoveCursorUp

method

```
procedure MoveCursorUp(aScroll : Boolean);
```

↳ Moves the cursor up one row.

The cursor is moved onto the previous row at the same column position. If the `aScroll` parameter is `False` and the cursor is currently on the top row, it is not moved. If the `aScroll` parameter is `True` and the cursor is on the top row, the screen or scrolling region is scrolled down one line and the cursor remains in the same position. The new line inserted at the top is initialized to space characters, using the current colors and attributes.

This method is limited to the current scrolling region.

8

OriginCol

read-only property

```
property OriginCol : Integer
```

↳ Defines the column origin of the current scrolling region.

If the scrolling region is not in effect, `OriginCol` is 1, being the left-most column of the screen. If the scrolling region is activated, `OriginCol` is the column number of the left-most column of the scrolling region.

`OriginCol` is read-only. To set the scrolling region, call `SetScrollRegion`.

See also: `OriginRow`, `SetScrollRegion`, `UseScrollRegion`

OriginRow

read-only property

```
property OriginRow : Integer
```

↳ Defines the row origin of the current scrolling region.

If the scrolling region is not in effect, `OriginRow` is 1, being the top row of the screen. If the scrolling region is activated, `OriginRow` is the row number of the first row of the scrolling region.

`OriginRow` is read-only. To set the scrolling region, call `SetScrollRegion`.

See also: `OriginCol`, `SetScrollRegion`, `UseScrollRegion`

Reset

method

```
procedure Reset;
```

↪ Resets the current colors and attributes to their defaults.

The screen is not changed by this method.

Row

property

```
property Row : Integer
```

↪ Defines the row of the cursor.

The Row property refers to the cursor. Reading the Row property returns the row number of the cursor, and setting it moves the cursor to that row in the same column.

The value used for the Row property is one-based; in other words, rows are counted from 1. If there are 24 rows down the terminal screen, the rows will be known as rows 1 to 24. If the row number used is out of the range of the screen or the current scrolling region, it is forced silently into bounds.

If a scrolling region is activated, the values for Row will be relative to the home position of the scrolling region, not the home position of the screen itself.

See also: Col, RowCount, SetCursorPosition

RowCount

property

```
property RowCount : Integer
```

Default: 24

↪ Defines the number of rows in the display view.

The RowCount property gets or sets the number of rows displayed by the terminal. Attempts to set RowCount to a value greater than SVRowCount are ignored.

See also: ColCount, Row

```
procedure SetCharAttrs(const aValue : TAdTerminalCharAttrs);  
  
TAdTerminalCharAttr = (tcaBold, tcaUnderline,  
    tcaStrikethrough, tcaBlink, tcaReverse, tcaInvisible);  
  
TAdTerminalCharAttrs = set of TAdTerminalCharAttr;
```

↳ Sets the current set of display attributes.

The display attributes define the style of new text. The attributes are bold, underlined, strikethrough, blink, reverse image (i.e., the text displayed in BackColor, the background in ForeColor), and invisible.

If you wish to add a new attribute to the current set, write code like this:

```
var  
    Attrs : TAdTerminalCharAttrs;  
begin  
    ...  
    MyBuffer.GetCharAttrs(Attrs);  
    Attrs := Attrs + [tcaUnderline];  
    MyBuffer.SetCharAttrs(Attrs);
```

See also: GetCharAttrs, SetDefCharAttrs

```
procedure SetCursorPosition(aRow, aCol : Integer);
```

↳ Moves the cursor to a given position.

This method is equivalent to setting the Row and Col properties individually.

See also: Col, Row

SetDefCharAttrs

method

```
procedure SetDefCharAttrs(const aValue : TAdTerminalCharAttrs);  
  
TAdTerminalCharAttr = (tcaBold, tcaUnderline,  
    tcaStrikethrough, tcaBlink, tcaReverse, tcaInvisible);  
  
TAdTerminalCharAttrs = set of TAdTerminalCharAttr;
```

↪ Sets the default set of display attributes.

The display attributes define the style of new text. The attributes are bold, underlined, strikethrough, blink, reverse image (i.e., the text displayed in BackColor, the background in ForeColor), and invisible. If Reset is called, the current display attributes are set to the value set by SetDefCharAttrs.

See also: GetDefCharAttrs, Reset, SetCharAttrs

SetHorzTabStop

method

```
procedure SetHorzTabStop;
```

↪ Sets a horizontal tab stop at the current cursor column.

If a horizontal tab stop is already set for that column, this method has no effect.

See also: ClearAllHorzTabStops, ClearHorzTabStop

SetScrollRegion

method

```
procedure SetScrollRegion(aTopRow, aBottomRow : Integer);
```

↪ Sets the scrolling region.

A scrolling region is a range of lines (from aTopRow to aBottomRow) within which writes to the screen and scrolling are restricted. This is often used for adding status lines that remain visible on screen while other text in the Terminal is scrolled.

Calling SetScrollRegion will automatically activate the new scrolling region.

See also: UseScrollingRegion

SetVertTabStop

method

```
procedure SetVertTabStop;
```

↪ Sets a vertical tab stop at the current cursor row.

If a vertical tab stop is already set for that row, this method has no effect.

See also: ClearVertTabStop, ClearAllVertTabStops

SVRowCount

property

```
property SVRowCount : Integer
```

Default: 200

↳ Defines the number of rows in the scrollbar view.

The `SVRowCount` property gets or sets the number of rows available in the scrollbar buffer. `SVRowCount` is being set, and the new value is less than `RowCount`, `RowCount` is also set to the new value of `SVRowCount`.

See also: `RowCount`

UseAutoWrap

property

```
property UseAutoWrap : Boolean
```

↳ Defines what happens when a character is written at the last column of a row

`UseAutoWrap` only has an effect if the cursor is at the last column of a row. If `UseAutoWrap` is `False`, the character is written at the last column of the current row, and the cursor does not move. If `UseAutoWrap` is `True`, the character is written at the last column of the current row, and then the cursor is moved to the first column of the next row, scrolling if necessary. (In fact, this last operation is coded as a call to `MoveCursorRight` with `aWrap` and `aScroll` both `True`.)

See also: `MoveCursorRight`

UseNewLineMode

property

```
property UseNewLineMode : Boolean
```

↳ Defines what the `DoLineFeed` method does.

If `UseNewLineMode` is `False`, `DoLineFeed` moves the cursor down one row, scrolling if necessary. The cursor stays in the same column.

If `UseNewLineMode` is `True`, `DoLineFeed` moves the cursor down one row, scrolling if necessary. The cursor is moved to the first column of the new row.

See also: `DoLineFeed`

UseScrollRegion

property

```
property UseScrollRegion : Boolean
```

↳ Defines whether the scrolling region is active.

If `UseScrollRegion` is `False`, the scrolling region is inactive and writes of text and scrolling apply to the whole screen. If `UseScrollRegion` is `True`, the scrolling region is in force and all screen changes are limited to the scrolling region.

Calling `SetScrollRegion` automatically forces this property to `True`, the new scrolling is brought into effect immediately.

See also: `SetScrollRegion`

UseWideChars

read-only, run-time property

```
property UseWideChars : Boolean
```

↳ Defines whether UNICODE characters are being stored by the terminal buffer.

The value of `UseWideChars` is set by a parameter to the `Create` constructor. In Delphi 1, this value is always `False`; it is only an option for 32-bit programs.

See also: `Create`

WriteChar

method

```
procedure WriteChar(aCh : AnsiChar);
```

↳ Writes a single character at the cursor.

The cursor is advanced after the character is written. Please see the `UseAutoWrap` property for a discussion of what happens if the cursor was on the last column of a row. The character is written using the current colors and attributes.

See also: `UseAutoWrap`, `WriteString`

WriteString

method

```
procedure WriteString(const aSt : string);
```

↳ Writes a string at the cursor.

This method is coded as a simple loop calling `WriteChar` for each character in the string.

See also: `WriteChar`

The Terminal Parsers

The purpose of a terminal parser is to identify terminal control sequences in the stream of data coming into the terminal (these control sequences are also more commonly known as *escape sequences*) and return the command specified to the caller. The terminal parser is the class that embodies the knowledge of the terminal's escape sequences; if another terminal is to be emulated then some of the first code to write is a new parser descendant to encapsulate the knowledge about the new terminal to be supported.

The parser ancestor class

To facilitate this process, there is an ancestor parser class, the `TAdTerminalParser` class. The main operations supported by this class are:

- Process a single character (virtual method).
- Clear the parser (virtual method).
- Get the command (property).
- Get the arguments (property).
- Get the sequence (property).

To gain a better understanding of these operations, we'll look at them individually from a high level.

Processing a single character is a virtual method that must be overridden in descendants. It will return one of four states:

1. The parser did not understand the character in the current context, and so it should be ignored.
2. The character is a displayable character and should be shown on the terminal screen.
3. The character started or continued an escape sequence, however that sequence is as yet incomplete.
4. The character completed an escape sequence; the parser converted it into a command and now this command must be processed.

It is up to the overridden method in a descendant class to determine how sequences are built up, and converted into commands, and so on. For an example of this process, please refer to the source code for the `TAdVT100Parser` class, the class that converts VT100 escape sequences into commands.

The operation of clearing the parser should reset the parser into a state such that no sequence is being built up, and hence no knowledge of previous characters is maintained. (The Clear operation is a virtual method of the class, which should be overridden in descendants.)

If the character processing method returns a value that signifies that a command has been identified, the Command property will return the current unprocessed command. This property is reset to a null command if a sequence is being built up.

The Arguments property is an array property returning the arguments for the current command. If there is no current command, the values are all zero. Essentially, certain control sequences will define arguments or parameters for the command being defined, for example, the command to move the cursor left might have an argument that defines how many positions to move, whether just the implied 1, or several.

The Sequence property returns the actual escape sequence that has just been parsed. If the current command is null, this property returns the empty string. This enables you to look at the sequence, maybe to parse it outside the parser's control, or to log it to a trace file.

The VT100 terminal parser

Async Professional provides one descendant of the ancestor parser class, TAdVT100Parser, the parser for VT100 terminals. The VT100 parser class encapsulates the knowledge of the standard VT100 escape sequences and to which command they refer. In addition, Async Professional's VT100 parser provides support for two extensions that, strictly speaking, are not part of the standard VT100 command set. The first of these is the understanding of escape sequences that set color: the original VT100 was strictly monochromatic. The second extension is to support escape sequences that insert, delete and erase characters and lines.

The VT100 parser has two modes to reflect the behavior of the standard VT100 terminal. The two modes are known as ANSI mode and VT52 mode. When in VT52 mode, the parser only accepts standard VT52 escape sequences, together with the <Esc>< sequence (which is used to switch back to ANSI mode). In ANSI mode, the VT52 sequences are ignored. The command to switch from one to the other is processed immediately within the method that processes characters as well being returned by it.

ANSI escape sequences for the VT100 terminal (and terminals that follow the ANSI specification) are always of the following form:

```
<Esc>[P...PI...IF
```

<Esc> is the escape character (ASCII \$27), '[' is the left bracket, the Ps are ASCII characters in the range of \$30 to \$3F, the Is are ASCII characters in the range of \$20 to \$2F, and the final F is in the range of \$40 to \$7E. Because of this definition, if the parser does not recognize a particular command, it can easily discard it—the end of the escape sequence is well defined.

With the VT100 terminal in ANSI mode, escape sequences either start with `<Esc>[,<Esc>#, <Esc>(, or <Esc>)`, or form a two character escape sequence `<Esc>x` where `x` is the command identifier. The `<Esc>[` sequences all follow the standard ANSI format previously mentioned. The other three sequence types are three character sequences with the final character identifying the command. Hence, the VT100 parser can know when unknown escape sequences terminate.

With the VT100 parser in VT52 mode, all escape sequences are two character sequences of the form `<Esc>x` with the `x` being the command identifier. The only exception is `<Esc>Y` where the two characters following the `Y` also form part of the sequence (`<Esc>Y` is “cursor position” and the two following characters encode the row and column numbers respectively). Hence, the VT100 parser working in VT52 mode can know when unknown sequences terminate.

The VT100 terminal also supports one-byte control characters, characters like tab, carriage return, line feed and so on. The VT100 parser decodes these as well as escape sequences, even when the control characters appear in the middle of escape sequences.

Table 8.1 defines the control characters and escape sequences understood by the VT100 parser. The individual characters in the escape sequence have been separated by spaces to make the parameters stand out more. The values `Pn`, `Ps`, `Pr`, or `Pc` denote parameters, meaning a numeric, switch, row or column value. Any one-byte control character not listed here is ignored: it will not be acted on or displayed.

Table 8.1: *VT100 parser control characters and escape sequences*

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

Table 8.1: *VT100 parser control characters and escape sequences (continued)*

Control Sequence	TerminalMode	Description
Esc # 4	VT100	Line is double height, bottom half.
Esc # 5	VT100	Line is single width, single height.
Esc # 6	VT100	Line is double width single height.
Esc # 8	VT100	Fill screen with E's.
Esc (A	VT100	Set G0 to UK charset.
Esc (B	VT100	Set G0 to US charset.
Esc (0	VT100	Set G0 to special linedraw charset.
Esc (1	VT100	Set G0 to alternate ROM charset.
Esc (2	VT100	Set G0 to alternate ROM LD charset.
Esc) A	VT100	Set G1 to UK charset.
Esc) B	VT100	Set G1 to US charset.
Esc) 0	VT100	Set G1 to special linedraw charset.
Esc) 1	VT100	Set G1 to alternate ROM charset.
Esc) 2	VT100	Set G1 to alternate ROM LD charset.
Esc 7	VT100	Save cursor and attributes.
Esc 8	VT100	Restore cursor and attributes.
Esc <	VT100	Enter ANSI mode (ignored).
Esc =	VT100	Enter application keypad mode.
Esc >	VT100	Enter numeric keypad mode.
Esc D	VT100	Index.
Esc E	VT100	Next line.
Esc H	VT100	Set tab stop at cursor.
Esc M	VT100	Reverse Index.
Esc [Pn @	VT100 enh	Insert characters at cursor.
Esc [Pn A	VT100	Cursor up.
Esc [Pn B	VT100	Cursor down.
Esc [Pn C	VT100	Cursor right.
Esc [Pn D	VT100	Cursor left.
Esc [Pr; Pc H	VT100	Cursor position.
Esc [Ps J	VT100	Erase part of display to cursor.
Esc [Ps K	VT100	Erase part of display from cursor.
Esc [Pn L	VT100 enh	Insert lines.
Esc [Pn M	VT100 enh	Delete lines.

Table 8.1: *VT100 parser control characters and escape sequences (continued)*

Control Sequence	TerminalMode	Description
Esc [Pn P	VT100 enh	Delete characters at cursor.
Esc [Pn X	VT100 enh	Erase characters at cursor.
Esc [Ps c	VT100	What are you?
Esc [Pr; Pc f	VT100	Cursor position.
Esc [Ps g	VT100	Clear tab stops.
Esc [Ps h	VT100	Set mode.
Esc [Ps l	VT100	Reset mode.
Esc [Ps; ...;Ps m	VT100	Set attributes, including color.
Esc [Ps n	VT100	Request terminal report.
Esc [Ps; ...;Ps q	VT100	Set LEDs.
Esc [Pt; Pb r	VT100	Set scrolling region (top, bottom row).
Esc [2; Ps y	VT100	Invoke confidence test.
Esc c	VT100	Reset.
Esc A	VT52	Cursor up.
Esc B	VT52	Cursor down.
Esc C	VT52	Cursor right.
Esc D	VT52	Cursor left.
Esc F	VT52	Set special character set.
Esc G	VT52	Set ASCII character set.
Esc H	VT52	Cursor to home.
Esc I	VT52	Reverse line feed.
Esc J	VT52	Erase to end of screen.
Esc K	VT52	Erase to end of line.
Esc Y	VT52	Direct cursor address.
Esc Z	VT52	Identify.
Esc <	VT52	Enter ANSI mode.
Esc =	VT52	Enter alternate keypad mode.
Esc >	VT52	Exit alternate keypad mode.

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

TAdTerminalParser Class

The TAdTerminalParser Class is the ancestor class that defines the functionality of a terminal parser. A terminal parser is a class that knows how to identify and extract terminal control sequences (also known as escape sequences) and convert them into commands and arguments. The parser is not responsible for executing the commands; that task is left to the terminal emulation component that uses the parser.

The methods defined by the TAdTerminalParser class are all virtual in order that they can be overridden in descendant classes, but they are not defined as abstract. Instead, in this ancestor class, they are all “do nothing” methods.

Hierarchy

TObject (VCL)

 TAdTerminalParser (AdTrmPsr)

Properties

Argument	Command
ArgumentCount	Sequence

Methods

Clear	ProcessChar
Create	ProcessWideChar

Reference Section

Argument

read-only, array property

```
property Argument[aInx : Integer] : Integer
```

↳ Returns the arguments for the current command

When a terminal control sequence has been completely received, the `ProcessChar` method will return `pctComplete`. Just prior to returning this success value, the parser will identify the command described by the sequence, and extract out the arguments (or parameters) from the sequence. The `ArgumentCount` property gives the number of arguments the parser finds. You can retrieve the individual arguments by using the `Argument` property at this time.

If you try and retrieve any arguments before a terminal control sequence has been completely received, no error or exception is generated. The return value will always be zero in this case.

In this ancestor class, the `Argument` property always returns zero.

See also: `ArgumentCount`, `ProcessChar`

ArgumentCount

read-only property

```
property ArgumentCount : Integer
```

↳ Returns the number of arguments for the current command

When a terminal control sequence has been completely received, the `ArgumentCount` property will return the number of arguments or parameters the parser found in the sequence. You can use this value to know how many arguments you can read from the `Argument` array property.

If a sequence is still being built up, the `ArgumentCount` property will be zero. It will only have a non-zero value after the `ProcessChar` method returns `pctComplete` (and before `ProcessChar` is called again) and if the sequence had at least one argument.

In this ancestor class, the `Argument` property always returns zero.

See also: `Argument`, `ProcessChar`

```
procedure Clear;
```

↳ Clears the internal state of the parser

By calling `Clear`, you will reset the parser to a state such that no sequence is being built up and no command is pending.

```
property Command : Byte
```

↳ Returns the current command

`Command` returns the command identified by the `ProcessChar` method. Once `ProcessChar` identifies a complete terminal control sequence, the parser will convert the sequence into the relevant command and extract its arguments. It is at this point that the `Command` property will be set to the command defined by the sequence. At any other time `Command` is set to zero (the `eNone` constant from `OOMISC.PAS`). In other words, if `ProcessChar` returns `pctComplete`, `Command` will be set to the relevant command defined by the just-received sequence, and it will be reset to `eNone` the next time `ProcessChar` is called.

`Command` can take on any of the `eXxx` values defined in `OOMISC.PAS` (such as `eCUB`, `eVTS` and so on). Please refer to the `OOMISC.PAS` source file for definitions of the supported commands.

In this ancestor class, the `Command` property always returns `eNone`.

See also: `ProcessChar`

```
constructor Create(aUseWideChar : Boolean);
```

↳ Creates the parser instance

`Create` allocates and initializes an instance of the parser class. The `aUseWideChar` parameter defines whether the parser will only accept wide characters (`True`) or single byte characters (`False`). If `True`, the `ProcessWideChar` method should be used to process the input data stream, character by character; if `False`, the `ProcessChar` method should be used.

See also: `ProcessChar`, `ProcessWideChar`

```
function ProcessChar(aCh : AnsiChar) : TAdParserCmdType; virtual;  
TAdParserCmdType = (pctNone, pctChar, pctPending, pctComplete);
```

↳ Processes a single character.

ProcessChar is the main workhorse of the parser class. It is the method that takes a character and decides whether that character forms part of a terminal control sequence or is just one that must be displayed. ProcessChar must maintain the current state of the parser (in other words, whether the parser is building up a terminal control sequence, has just completed a sequence, etc.).

ProcessChar can return one of several values. `pctNone` means that the character passed in was not understood by the parser. Either the character was a single-byte control character (in other words, a non-displayable character less than the space character) that would be ignored by the original terminal, or the character terminated an escape sequence, but that escape sequence was not known by the parser. The caller of ProcessChar can safely ignore the problem since there is no recovery action to take because the parser has been left in a correct state. `pctChar` means that the input character should be displayed on the terminal. `pctPending` means that the input character formed part of an escape sequence that is being built up, character by character. There is nothing further to do at this point since the escape sequence has not yet been completed.

The last possible result value is `pctComplete`. This value indicates that the parser has captured a complete terminal control sequence. Furthermore it indicates that the parser has identified the command defined by the escape sequence and has set the `Command` property. It will have extracted all of the command arguments or parameters and set both the `Argument` and `ArgumentCount` properties. Finally, it will make sure that the `Sequence` property returns the entire escape sequence.

In this ancestor class, the ProcessChar method always returns `pctNone`.

If ProcessChar is called for a parser that was created to expect wide characters, a parser exception will be raised.

See also: `Argument`, `ArgumentCount`, `Command`, `Create`, `ProcessWideChar`, `Sequence`

```
function ProcessChar(aCh : WideChar) : TAdParserCmdType; virtual;  
TAdParserCmdType = (pctNone, pctChar, pctPending, pctComplete);
```

↳ Processes a single character.

ProcessWideChar is the wide character (or UNICODE) version of ProcessChar. It works in the same fashion with the exception of accepting UNICODE characters only. Please see the ProcessChar method reference for details.

In this ancestor class, the ProcessWideChar method always returns pctNone.

If ProcessWideChar is called for a parser that was created to expect single-byte ASCII characters, a parser exception will be raised.

See also: Create, ProcessChar

```
property Sequence : string
```

↳ Returns the current terminal control sequence.

Once ProcessChar identifies a complete terminal control sequence, and is about to return pctComplete, the parser will set this property to the full sequence string. The property will maintain its value until the next time ProcessChar is called.

If a sequence is being built up or there is no current command, the value of the Sequence property is the empty string. Reading the Sequence property at the wrong time generates no error or exception.

In this ancestor class, the Sequence property always returns the empty string.

See also: ProcessChar

TAdVT100Parser Class

The TAdVT100Parser class is the descendant of the TAdTerminalParser. It defines a parser that understands VT100 terminal data streams.

Hierarchy

TObject (VCL)

- ➊ TAdTerminalParser (AdTrmPsr) 229
 - TAdVT100Parser (AdTrmPsr)

Properties

Argument	➊ Command	➊ Sequence
➊ ArgumentCount	InVT52Mode	

Methods

➊ Clear	ProcessChar
➊ Create	➊ ProcessWideChar

Reference Section

Argument

read-only, array property

```
property Argument[aInx : Integer] : Integer
```

↪ Returns the arguments for the current command.

Please see `TAdTerminalParser` for a description of how the `Argument` property works.

Some VT100 escape sequences use default values for certain arguments. For example, the Cursor Right sequence is `<Esc>[C`, but it is also possible to specify the number of character positions to move. In our case the sequence would be `<Esc>[2C` for two positions. The VT100 parser in `TAdVT100Parser` would create a single argument of 1 (the default) for the first case, and a single argument of 2 (an explicit argument) for the second case. There is no way to find out if the parser has created an implicit default argument or used an explicit argument, unless you wish to parse the `Sequence` property yourself. In general, this would not matter: the value of an argument is all you need to process the command properly.

In certain cases the parser will be unable to determine the value of the default argument since it would depend on information it does not have. An example for this is the `Set Scrolling Region` command. This command has two arguments: the top row and the bottom row of the scrolling region. The defaults are the top row and the bottom row of the screen itself, neither of which are known by the parser. If the parser is unable to work out what the default value of an argument is, it will set the relevant element of the `Argument` array property to `-1`, meaning “default.”

There is a set of VT100 escape sequences that use the ‘?’ character in the parameter part of the sequence. An example is `<Esc>[?3h` to set the VT100 terminal into 132-column mode. The parser in the `TAdVT100Parser` class parses this escape sequence as having two arguments, the ‘?’ and the ‘3’. Since the `Argument` property only returns integers, the ‘?’ character must be replaced by a special integer value, in this case, `-2`. Hence, in our example, the command would have two arguments: `-2` and `3`. (The reason for this slightly eccentric behavior, versus just ignoring the ‘?’ character altogether, is that ‘?’ signifies a DEC extension rather than just a straightforward ANSI escape sequence. The parser needs to return this information as well.)

```
property InVT52Mode : Boolean
```

↳ Returns whether the terminal has been switched to VT52 mode.

The VT100 terminal can be switched between two modes: ANSI and VT52. The terminal will understand different escape sequences in each of the two modes. Once the terminal is in a given mode, the escape sequences for the other mode will not be understood or acted on.

The VT100 parser class tracks the escape sequences that switch the terminal from mode to mode and will set an internal flag to denote whether the parser should identify ANSI escape sequences or VT52 escape sequences. The mode switch sequences, `<Esc>[?2l` to switch to VT52 mode and `<Esc><` to switch back to ANSI mode, are the only sequences that are tracked inside the parser, since they directly affect the operation of the parser.

The `ProcessChar` method will return the `eDECANM` result value if the parser identifies one of the escape sequences that switch terminal modes. The `InVT52Mode` property will have been set by the time that `ProcessChar` terminates and returns this value. Hence, if `ProcessChar` returns `eDECANM`, you can check the value of the `InVT52Mode` to find out which mode the terminal should now be in.

See also: `ProcessChar`

ProcessChar

method

```
function ProcessChar(aCh : AnsiChar) : TAdParserCmdType; override;
TAdParserCmdType = (pctNone, pctChar, pctPending, pctComplete);
```

↳ Processes a single character.

Please see `TAdTerminalParser` for a description of how the `ProcessChar` method works.

The VT100 terminal (and the ANSI specification) has one peculiarity that is generally not well implemented by terminal emulators. The VT100 terminal supports several one-byte control characters: line feed (\$10), carriage return (\$13), horizontal tab (\$09), and so on. Like the VT100 terminal, the VT100 parser class recognizes these one-byte control characters, and acts upon them, even in the middle of an escape sequence. For example, `<Esc><CR>7` (where `<CR>` is the carriage return character) will be interpreted as `<CR><Esc>7`; the `<CR>` being acted upon immediately, rather than being ignored. For this example, the result values returned by `ProcessChar` for the three characters would be: `pctPending` for the `<Esc>`; `pctComplete` for the `<CR>` (and the `Argument`, `ArgumentCount`, `Command` and `Sequence` properties would be set accordingly); and, finally, `pctComplete` for the `<Esc>7` sequence (again setting the `Argument`,

ArgumentCount, Command and Sequence properties accordingly). Notice that this behavior necessitates that the parser save the not-quite-complete escape sequence somewhere: this is done automatically.

The ANSI specification details the behavior if an <Esc> character appears in the middle of an escape sequence: the second <Esc> cancels the original escape sequence and starts a new one. So, for example, with <Esc>[12<Esc>7, the second <Esc> character would cancel the partial escape sequence that begins <Esc>[12 and start a new one.

Two one-byte control characters that have special significance are CAN (\$18) and SUB (\$1A). These cancel the current escape sequence (if one is being built up). If ProcessChar is called with either of these characters, at any time, it will clear the VT100 parser and return pctNone.

Although the ProcessChar method will successfully parse and decode the standard VT100 escape sequences, in general it will not carry out any of the decoded commands internally. That job is left to the terminal component itself. However, there is an exception to the rule. There are two escape sequences that affect the operation of the VT100 parser. The first is the <Esc>[?2l sequence, which switches the terminal into VT52 mode. The second is the <Esc>< sequence, which switches it back to ANSI mode. The importance of these two modes is that the terminal recognizes different escape sequences in each of these modes and, hence, the parser has to be able to know when to parse the ANSI sequences and when to parse the VT52 sequences. For example, in ANSI mode the <Esc>D sequence is the same as move the cursor one line down (the “Index” operation) whereas in VT52 mode it means move the cursor one position left. If the parser did not recognize which state the terminal was in, it may parse this sequence badly. Hence, the VT100 parser class maintains the InVT52Mode property and this is set internally by the ProcessChar method to reflect the current ANSI/VT52 mode on receipt of one of these two sequences.

See also: InVT52Mode

The TAdKeyboardMapping Class

The TAdKeyboardMapping class provides a simple, convenient method to specify the PC keystrokes that map onto the emulated terminal keystrokes, and also what control sequence those terminal keystrokes are going to send to the host computer.

There are three parts to the keyboard mapping. The first part is merely for convenience: it is a mapping of the keyboard's virtual key codes to the virtual key names. This mapping is standard and is provided with Microsoft's documentation. An example of a single mapping is to specify that the virtual key code \$70 is the F1 key, which is usually known by the name "VK_F1" in Microsoft's documentation.

The second part of the keyboard mapping is the definition of the character or control sequence that is sent by the original keyboard to the host computer when a key is pressed. For a VT100 terminal, for example, pressing the PF1 key will either send an <Esc>P or an <Esc>OP sequence to the host, depending on the mode the terminal is in at the time. Again, this mapping is standard and is provided as part of the terminal manufacturer's documentation.

8

Whereas the two mappings just described are fixed by standards, the last mapping is where the creativity and individuality comes in. This set of mappings details which PC keystroke gets mapped to which terminal keystroke. There are no standards for this (though some mappings should be fairly obvious: for example, the up arrow key should be mapped to the terminal's up arrow key), and so you can customize to a large extent which keys perform which action using this third mapping.

The TAdKeyboardMapping class is designed to be used by a terminal emulator. The terminal component will trap the user's keystroke and pass it onto the emulator. The emulator will lookup the virtual key code in the keyboard mapping table and will retrieve the virtual key name. It will then prefix this name with the names of the shift keys that are active, and lookup this combination in the keyboard mapping class. If the lookup succeeds, the keyboard mapping class returns the name of a terminal key and this, in turn, can be looked up to find the character or control sequence that should be sent to the host computer. If at any time a lookup fails, there is no special mapping for the keystroke and so the default action takes over (for example, for an alphabetic key, the relevant character is sent to the host computer).

Although it may seem excessive to have three lookups per keystroke to get to the final sequence to send to the host, in reality the design it provides a balance between fast conversion of keystrokes and ease of specification of the various mappings. The lookups are performed using a hash table, which is considered the data structure of choice for this kind of operation.

The `TAdKeyboardMapping` class is convenient to use when you have to specify a large number of mappings. There are two methods for loading a set of mappings: first from a specially, yet simply, formatted text file, and, second, from a resource within the application. Thus, it is possible to have a default mapping linked into the application, but also to be able to provide a way of altering the mappings at run time. To help create the resource, the class also has a method to write its current mapping set to a binary file, which can then be compiled into a resource file. The following code shows this process:

```
var
  KeyMap : TAdKeyboardMapping;
begin
  KeyMap := TAdKeyboardMapping.Create;
  try
    KeyMap.LoadFromFile('ADKEYVT1.TXT');
    KeyMap.StoreToBinFile('VT100.BIN');
  finally
    KeyMap.Free;
  end;
end;
```

The code creates a `TAdKeyboardMapping` instance called `KeyMap`. A set of mappings is then read from a file called `ADKEYVT1.TXT` (this file is a default set of keyboard mappings for a VT100 emulator that is provided with Async Professional). The mappings are then written out to a file called `VT100.BIN`. This latter file can be compiled into a resource using the standard resource compiler that comes with Delphi (either `BRCC.EXE` or `BRCC32.EXE`). The resource script (RC file) required for this is as follows:

```
MyVT100KeyMap RCDATA VT100.BIN
```

If you name the resource file `VT100.RC`, the resource compiler will create a file called `VT100.RES`. Adding the resource to your application at that point is merely a case of adding the following line to your project file and recompiling:

```
{$R VT100.RES}
```

Now you can call the `LoadFromRes` method of your `TAdKeyboardMapping` instance to load this set of keyboard mappings.

Hierarchy

TObject (VCL)

 TAdKeyboardMapping (ADTrmKey)

Properties

 Count

Methods

 Add

 Get

 LoadFromRes

 Clear

 LoadFromFile

 StoreToBinFile

Reference Section

Add

method

```
function Add(const aKey : TAdKeyString;  
    const aValue : TAdKeyString) : Boolean;  
  
TAdKeyString = string[63];
```

↪ Adds a new keyboard mapping to the instance.

The `aKey` string is the lookup value (the key string) and `aValue` is the string associated with it. `Add` does not validate these two strings to be in any particular format, so it is possible to fill the instance with nonsensical strings that would never be looked up.

If the key string and its value were successfully added, `Add` returns `True`. If the key string is already present, `Add` will return `False` and leave the existing mapping as is.

Please see `LoadFromFile` for a discussion on how to define the key strings and their values.

See also: `LoadFromFile`

Clear

method

```
procedure Clear;
```

↪ Clears all keyboard mappings.

`LoadFromFile` and `LoadFromRes` automatically call `Clear` prior to loading a set of mappings. If you wanted to load a set of mappings from another source, you would have to call `Clear` first, and then `Add` for every mapping in your set.

See also: `Add`, `LoadFromFile`, `LoadFromRes`

Count

read-only, run-time property

```
property Count : Integer
```

↪ Determines the number of mappings in the class.

The `Count` property will return the number of different mappings contained in the class. It does not differentiate between the different types of mappings.

```
function Get(const aKey : TAdKeyString) : TAdKeyString;
TAdKeyString = string[63];
```

↳ Returns the value of a looked up key string.

aKey is the key string to lookup. If it was found, the return value will be the string with which it is associated. If it was not found, the return value is the empty string. No exception is raised if the key string was not found.

See also: Add

LoadFromFile

method

```
procedure LoadFromFile(const aFileName : string);
```

↳ Loads a set of keyboard mappings from a text file.

The name of the file is given by aFileName.

The file is a text file in a particular format. LoadFromFile will follow these rules when reading the file.

- Any completely blank line is ignored.
- Any line starting with a * is a comment and is skipped.
- Any line starting with at least one space is a detail line. A detail line consists of two “words”, where a word is a set of up to 63 characters without an embedded space and is case sensitive. Words are separated by spaces (not tab characters). Any other characters appearing after the two words is taken to be a comment and is skipped.
- Any detail line that cannot be parsed is simply ignored.
- Any line that doesn’t match the above is skipped.

The words in a detail line have some formatting associated with them. The emulator imposes this formatting so that it can generate the correct key strings when the user presses a key. The LoadFromFile method does not validate these formats in any way.

“\e” in a word means the Escape character

\xnn, where nn is a hex number, represents that ASCII character

If you want to specify shift keys with virtual key names, use the mnemonics “shift”, “ctrl”, and “alt.” Combine them with the virtual key name using the + sign. If you want to specify more than one shifted key, make sure that they are in the order “shift”, “ctrl”, “alt.”

An example of a virtual key code to virtual key name mapping would be:

```
* This is the definition of F1
  \x70  VK_F1
```

An example of a defining the sequence to be sent by a terminal key is:

```
* This is the definition of PF1 on a VT100
* (it sends <Esc>OP)
  DEC_PF1  \eOP
```

An example of mapping Alt+F1 so that it acts like the PF1 key on a VT100 is:

```
* Map Alt+F1 to PF1
  alt+VK_F1  DEC_PF1
```

Let's follow how the emulator would use these mappings. The user presses Alt+F1. The emulator would convert the virtual key code returned by Windows (an integer equal to 70 hex) into a string ("x70"). It would then lookup this string in the keyboard mapping object and get the value "VK_F1" back. It then prefixes the "alt" keyword to this value to give "alt+VK_F1" and looks this up. This returns the value "DEC_PF1". This is in turn looked up to return the value "\eOP". The emulator then interprets this string as <Esc>OP in order to send the correct sequence to the host. If a lookup fails at any stage, the keystroke is assumed to be unmapped. If the emulator cannot interpret the final control sequence string, it will just send it as is.

Please see ADKEYVT1.TXT for a complete set of mappings that define one way of mapping the VT100 keys onto the PC keyboard.

The only errors than can occur with LoadFromFile are file I/O errors. Internally LoadFromFile uses a string list (TStringList) to read the entire file into memory and so any exceptions raised will be those raised by this class.

See also: Add

LoadFromRes

method

```
procedure LoadFromRes(
  aInstance : THandle; const aResName : string);
```

↳ Loads a set of keyboard mappings from a resource.

The name of the resource is given by aResName, and the resource is to be found in the module instance given by hInstance. The resource must be in the binary format created by the StoreToBinFile method. If any error occurs during the load process, for example the resource cannot be found or the resource is not in the correct format, the mappings are cleared. No exception is raised.

```
procedure StoreToBinFile(const aFileName : string);
```

↳ Stores the current set of mappings to file.

The name of the file is given by `aFileName`. The binary file so created can be compiled into a resource that can be read by `LoadFromRes`.

The only errors than can occur are file I/O errors. Internally `StoreToBinFile` uses a file stream (`TFileStream`) and so any exceptions raised will be those raised by this class.

The TAdCharSetMapping Class

The TAdCharSetMapping class provides a method to emulate the different character sets used by terminals by using glyphs from different fonts.

One of the problems of emulating a terminal with a Windows program is that terminals, especially the older ones, use specialized glyphs that are not available in the normal Windows fonts. The character set mapping class attempts to work around this problem by enabling the emulator to obtain the glyphs for different character sets from different fonts.

Before proceeding with the character set mapping class, we should define a few terms. A *character* is a binary value, usually byte-sized. The way we usually think of a character is both as its binary value and as its visual form. Hence, when we think of the character 'a', for example, we think of its value (\$61) and of its visual form (a drawing of a lowercase letter 'a', like the one you just read just now). However, the link between the character 'a' and the visual form of the character is not fixed. Different fonts may display the character 'a' in different ways, and some visual representations (what are known as *glyphs*) may look nothing like a lowercase 'a'. An example is the Symbol font where the character 'a' is drawn as a Greek lowercase α . With terminals, there may be several character sets that are available at once, and these character sets function in the same way as fonts. On the VT100, for example, there are two main character sets: the standard one and the special graphics one, which displays the line draw characters. In the first, the character 'a' is drawn as a lowercase 'a'. In the second it is drawn as a checkerboard glyph.

The character set mapping class is designed to be used by a terminal emulator. When the terminal display needs to be painted, the emulator will use a character set mapping class to identify which glyphs need to be painted on the terminal window. For this it will pass a string of characters to the mapping class, together with the character set to be used. The TAdCharSetMapping object will return a script to the emulator. This script will consist of a series of text drawing commands all of the form: "using font X, draw string Y." Usually the script will consist of just one command, since the characters will generally all come from the standard ASCII set and hence can be drawn using just one font.

The `TAdCharSetMapping` class gets its mapping data from one of two sources: a specially formatted text file, or a resource linked into the application. Thus it is possible to have a default mapping linked to the application, but, also, to be able to provide a way of altering the mappings at run time (maybe to suit the fonts on the user's machine). To help create the resource the character set mapping class has a method to write its current mapping set to a binary file, which can then be compiled into a resource file. The following code shows this process:

```
var
    CharSetMap : TAdCharSetMapping;
begin
    CharSetMap := TAdCharSetMapping.Create;
    try
        CharSetMap.LoadFromFile('ADCHSVT1.TXT');
        CharSetMap.StoreToBinFile('VT100.BIN');
    finally
        CharSetMap.Free;
    end;
end;
```

8

The code creates a `TAdCharSetMapping` instance called `CharSetMap`. A set of mappings is then read from a file called `ADCHSVT1.TXT` (this file is a default set of character set mappings for a VT100 emulator that is provided with Async Professional). The mappings are then written out to a file called `VT100.BIN`. This latter file can be compiled into a resource using the standard resource compiler that comes with Delphi (either `BRCC.EXE` or `BRCC32.EXE`). The resource script (RC file) required for this is as follows:

```
MyVT100CharSetMap RCDATA VT100.BIN
```

If you name the resource file `VT100.RC`, the resource compiler will create a file called `VT100.RES`. Adding the resource to your application at that point is merely a case of adding the following line to your project file and recompiling:

```
{ $R VT100.RES }
```

Now you can call the `LoadFromRes` method of your `TAdCharSetMapping` instance to load this set of character set mappings.

Hierarchy

TObject (VCL)

 TAdCharSetMapping (ADTrmMap)

Properties

 Count

Methods

Add	GetNextDrawCommand	StoreToBinFile
Clear	LoadFromFile	
GenerateDrawScript	LoadFromRes	

Reference Section

Add

method

```
function Add(const aCharSet : TAdKeyString;
  aFromCh : AnsiChar; aToCh : AnsiChar;
  aFont : TAdKeyString; aGlyph : AnsiChar) : Boolean;

TAdKeyString = string[63];
```

✚ Adds a new character set mapping to the instance.

The `aCharSet` parameter is the name of the character set. This name is defined by the emulator if there is no standard name for it. The character set name must be unique. In other words, different character sets will have different names. `aFromCh` and `aToCh` define an inclusive range of characters for the mapping (if `aToCh` equals `aFromCh`, the mapping is for a single character). `aFont` is the name of the font from which the glyph or glyphs are taken. `aGlyph` is the glyph from which the mapping starts. `aFromCh` is mapped to `aGlyph`, the character after `aFromCh` is mapped to the glyph after `aGlyph`, and so on until `aToCh`.

The result value is `True` if the character set mapping was added, `False` otherwise. The latter result would mean that the combination of `aCharSet` and the supplied range clashed with a mapping already present.

To help in designing portable character set mappings, there is one special value that can be used for the font name. If the name is “<Default>”, then the emulator will use the currently defined font for the terminal component to display text.

For example, suppose you wish to add a character set mapping for the standard ASCII characters, using Courier New as the font. This is the statement you would use:

```
var
  MyMap : TAdCharSetMapping;
begin
  ...
  if not MyMap.Add(
    'MyCharSet', ' ', '~', 'Courier New', ' ') then
    ..mapping not added..
```

This tries to add a mapping for all of the characters between space and ‘~’ in the `MyCharSet` character set to that same characters (i.e., glyphs) from the Courier New font.

Clear

method

```
procedure Clear;
```

↪ Clears all character set mappings.

LoadFromFile and LoadFromRes automatically call Clear prior to loading a set of mappings. If you wanted to load a set of mappings from another source, you would have to call Clear first, and then Add for every mapping in your set.

See also: Add, LoadFromFile, LoadFromRes

Count

read-only, run-time property

```
property Count : Integer
```

↪ Determines the number of mappings in the class.

The Count property will return the number of different mappings contained in the class. It does not differentiate between the different character sets or ranges. It is of use primarily for writing out the mappings to a file or other object.

GenerateDrawScript

method

```
procedure GenerateDrawScript(  
    const aCharSet : TAdKeyString; const aText : string);
```

↪ Generates a draw script from a string.

The emulator, when it needs to display text on the terminal window, will separate out the text to be drawn into strings from different character sets. For each individual string it will call the GenerateDrawScript method of its internal character set mapping object to generate a series of drawing commands that can be used to draw the text on the screen. Each command will be of the form “switch to font X, write string Y.” The emulator then reads the commands one at a time using the GetNextDrawCommand method, and draws the text in the required font.

If GenerateDrawScript is called before all of the commands from the previous script have been read, the previous commands are destroyed and will no longer be available.

The aCharSet parameter is the name of the character set. aText is the string of characters that have to be drawn using the given character set. The GenerateDrawScript method will work out which fonts and glyphs are required to draw the characters and generate a list of drawing commands that, when executed one after the other, will produce the required effect.

See also: GenNextDrawCommand

```
function GetNextDrawCommand(
    var aFont : TAdKeyString; var aText : string) : Boolean;
```

↳ Retrieves the next draw command from the current script.

The emulator, when it needs to display text on the terminal window, will separate out the text to be drawn into strings from different character sets. For each individual string it will call the `GenerateDrawScript` method of its internal character set mapping object to generate a series of drawing commands that can be used to draw the text on the screen. Each command will be of the form “switch to font X, write string Y.” The emulator then reads the commands one at a time using the `GetNextDrawCommand` method, and draws the text in the required font.

The `GetNextDrawCommand` method returns `True` if there is another command, and sets `aFont` to the font name required and `aText` to the string that needs to be drawn in that font. It returns `False` if there are no more commands in the current script. The emulator will continue to call `GetNextDrawCommand` and draw the specified text in the given font until the method returns `False` and the script is exhausted.

See also: `GenerateDrawScript`

LoadFromFile

method

```
procedure LoadFromFile(const aFileName : string);
```

↳ Loads a set of character set mappings from a text file.

The name of the file is given by `aFileName`.

The file is a text file in a particular format. `LoadFromFile` will follow these rules when reading the file.

- Any completely blank line is ignored.
- Any line starting with a `*` is a comment and is skipped.
- Any line starting with at least one space is a detail line. A detail line consists of five “words.” A word is defined as a set of up to 63 characters without an embedded space, or as a set of up to 63 characters enclosed by quote marks (single or double). A word is case sensitive. Words are separated by spaces (NOT tab characters). Any other characters appearing after the five words is taken to be a comment and is skipped.
- Any detail line that cannot be parsed is simply ignored.
- Any line that doesn’t match the above is skipped.

The five words, in order, are the same as the five parameters to the Add method. They denote the following:

- The character set name.
- The from character for the range.
- The to character for the range.
- The font name.
- The from glyph for the range.

The words that define a character can either be the character itself or be the hex representation of the character in the form \xnn with “nn” being the hex value. Hence, the space character would be represented by \x20, and the character ‘a’ would be shown by the single letter a.

To help in designing portable character set mappings, there is one special value that can be used for the font name. If the name is “<Default>”, then the emulator will use the currently defined font for the terminal component to display text.

An example of defining a character set mapping for the standard ASCII characters would be

```
* This defines the standard ASCII characters
MyCharSet \x20 ~ 'Courier New' \x20
```

This would be read as: the character set name is “MyCharSet”; the range of characters is from the space character to the ‘~’ character, inclusive; the font name is “Courier New”; and the starting glyph is that for the space character.

Please see ADCHSVT1.TXT for a complete set of mappings that define one way of mapping the VT100 character sets keys onto standard Windows fonts.

The only errors than can occur with LoadFromFile are file I/O errors. Internally LoadFromFile uses a string list (TStringList) to read the entire file into memory and so any exceptions raised will be those raised by this class.

See also: Add

```
procedure LoadFromRes(  
    aInstance : THandle; const aResName : string);
```

↳ Loads a set of keyboard mappings from a resource.

The name of the resource is given by `aResName`, and the resource is to be found in the module instance given by `hInstance`. The resource must be in the binary format created by the `StoreToBinFile` method. If any error occurs during the load process, for example the resource cannot be found or the resource is not in the correct format, the mappings are cleared. No exception is raised.

StoreToBinFile**method**

```
procedure StoreToBinFile(const aFileName : string);
```

↳ Stores the current set of mappings to file.

The name of the file is given by `aFileName`. The binary file so created can be compiled into a resource that can be read by `LoadFromRes`.

The only errors than can occur are file I/O errors. Internally `StoreToBinFile` uses a file stream (`TFileStream`) and so any exceptions raised will be those raised by this class.

The TAdTerminalEmulator Class

The TAdTerminalEmulator class is the base class for all terminal emulators. An emulator is designed to work hand-in-hand with a terminal component to provide the look and feel of a particular terminal. The emulator does all the hard work: it interprets the input data stream, looking for terminal control sequences and ordinary text characters and processing them; it accepts all keystrokes from the terminal component, identifies them and converts them to output the correct terminal sequence. It also maintains the buffer that describes what the terminal display looks like, the text characters, colors, attributes and character sets. It has a character set mapping object that enables it to identify which fonts are used for which glyphs for display. Finally, and possibly the most important, it uses the terminal component's canvas object to draw a representation of the current view of the terminal.

Async Professional provides two descendants of TAdTerminalEmulator. The first is the simplest emulator of all, the teletype or TTY emulator, TAdTTYEmulator. This emulator performs no keystroke conversion, no character set mapping, and no parsing of the input stream. Every character that is received is drawn directly onto the terminal display. All standard keystrokes (alphabetic characters, numeric characters, and control characters) are sent directly to the host computer. All other keystrokes (function keys, cursor movement keys, and the like) are ignored. When you drop a terminal component onto the form and do not use an emulator component, the terminal component will create an internal instance of the TTY emulator and use that instead.

The second emulator provided by Async Professional is the VT100 terminal emulator, TAdVT100Emulator. This emulator provides a complete implementation of a standard VT100 terminal. Features provided by this emulation include:

- Double height and double width characters.
- Support for the keyboard LEDs.
- The ability to switch character sets to use the line draw characters.
- Applying the scrolling region.
- Support for the various VT100 modes, including keyboard modes.

The emulator also provides support for the following features that are not part of the standard VT100 specification, but are nevertheless generally expected to be present in an emulation. These features are extracted from the ANSI specification and the VT220 specification:

- Support for erasing, deleting and inserting characters.
- Support for the different ANSI color attributes.

Finally, please note that the emulator does not support the following, sometimes optional, hardware characteristics of some VT100 terminals. In fact, in response to a “What Are You?” request (`<Esc>[c`) the VT100 terminal emulator will respond as a “basic VT100 with no options” (`<Esc>[?0c`).

- Interlace mode (switching between 240 & 480 scan lines per frame).
- Smooth scrolling (the emulator performs jump scrolling all the time).
- STP processor option.
- AVO (advanced video option).
- GPO (graphics processor option).

To use an emulator component you would drop one on the form and then drop a terminal component onto the same form. The terminal component would find the emulator component and link up with it. At that point the terminal and emulator components will function as one composite component and will perform all the necessary work to make the combination act as an original terminal. There is no extra work to be done on your part.

Obviously, if you wish to alter the behavior of an emulator, you will need to know about its properties, events and methods, and understand the use of the internal objects the emulator utilizes. If you wish to write a different terminal emulator then you would also need to know this extra information, but for general use of the Async Professional terminal component family, it will generally be “drop and go.”

Hierarchy

TComponent (VCL)

- TApdBaseComponent (OOMisc) 8
- TAdTerminalEmulator (ADTrmEmu)

Properties

Buffer	NeedsUpdate	● Version
CharSetMapping	Parser	
KeyboardMapping	Terminal	

Methods

BlinkPaint	KeyDown	LazyPaint
GetCursorPos	KeyPress	Paint

Reference Section

BlinkPaint

virtual method

```
procedure BlinkPaint(aVisible : Boolean); virtual;
```

↳ Paints the blinking text.

Most terminals support blinking text. For a Windows emulation of blinking text, the text must first be drawn normally, then a short time later that text is erased so that it is essentially invisible, and then a further short time later the text is redrawn normally. The effect for the user is that the text blinks on and off regularly.

The terminal component performs no painting of its own. Instead it defers that job to the emulator component, where the knowledge of such processing for a given terminal is embodied. Instead the terminal component maintains the blink timer, a timer object that fires at regular intervals. The terminal component will call the emulator's `BlinkPaint` method to either display all of the blinking text (`aVisible` is `True`) or to erase it and just show the background color (`aVisible` is `False`). It is up to the emulator to maintain a list of regions of the terminal display that must be drawn and redrawn in this fashion.

See also: `LazyPaint`, `Paint`

Buffer

read-only, run-time property

```
property Buffer : TAdTerminalBuffer
```

↳ Returns the terminal buffer.

The `Buffer` property is the data structure that holds the elements that go to make up the visual representation of the terminal. These elements are the text characters, the foreground and background colors, the attributes of the text (underlined, blinking, and so on), and the character set ids for the characters.

The emulator creates a buffer object in its `Create` constructor and frees it in the `Destroy` destructor. It is not possible to replace the buffer object in between those two times.

CharSetMapping

read-only, run-time property

```
property CharSetMapping : TAdCharSetMapping
```

↳ Returns the character set mapping object for the emulator.

The `CharSetMapping` property is the data structure that holds the various character-set-to-font-glyph mappings for the emulator. The emulator will attempt to map a character in a particular character set to a glyph in a font, before displaying that character in the terminal component's window.

The emulator creates a character set mapping object in its Create constructor and frees it in the Destroy destructor. It is not possible to replace the character set mapping object in between those two times. However, it is possible to replace the set of mappings for the character set mapping object: just call its LoadFromFile or LoadFromRes methods. This will have the effect of changing the functionality of the emulator.

Note that some emulators will not have a character set mapping object; reading CharSetMapping will return nil. This is because the emulation concerned—for example, the TTY emulator—does not require any character set mapping services.

GetCursorPos

virtual method

```
procedure GetCursorPos(var aRow, aCol : Integer); virtual;
```

↪ Returns the cursor position.

The emulator maintains the position of the terminal's cursor. The terminal component can call the GetCursorPos method to get the row and column values for the cursor's position, so that it can draw a blinking caret at this position on its client window.

Using terminal terminology, the “cursor” means the keyboard edit point. It is the place where new text data will be written to the screen. In Windows terminology however, this is known as the caret, the cursor being the mouse pointer.

KeyboardMapping

read-only, run-time property

```
property KeyboardMapping : TAdKeyboardMapping
```

↪ Returns the keyboard mapping object for the emulator.

The KeyboardMapping property is the data structure that holds the various keystroke-to-terminal control sequence mappings for the emulator. The emulator will attempt to map a keystroke provided by a call to the KeyDown method into a terminal control sequence to be sent to the host computer by calling the Get method of the keyboard mapping object.

The emulator creates a keyboard mapping object in its Create constructor and frees it in the Destroy destructor. It is not possible to replace the keyboard mapping object in between those two times. However, it is possible to replace the set of mappings for the keyboard mapping object: just call its LoadFromFile or LoadFromRes methods. This will have the effect of changing the functionality of the emulator.

Note that some emulators will not have a keyboard mapping object; reading KeyboardMapping will return nil. This is because the emulation concerned—for example, the TTY emulator—does not require any keyboard mapping services.

```
procedure KeyDown(var Key : Word; Shift: TShiftState); virtual;
```

↳ Processes a key down message from the terminal component.

The terminal component performs no keystroke processing of its own. Instead it defers that job to the emulator component, where the knowledge of such processing for a given terminal is embodied.

The terminal component's `KeyDown` method—the standard VCL `KeyDown` method for `TWinControl` descendants—merely calls the `KeyDown` method of its attached emulator and performs no other processing of its own. As a convenience, the terminal will also call the emulator's `KeyDown` method for system keys (those reported by a `WM_SYSKEYDOWN` message), as well as those keystrokes that are reported through the terminal's keyboard hook interface.

The emulator will use its keyboard mapping object to determine what to do with the keystroke. If there is a conversion defined, the emulator will use the terminal's `ComPort` property to send the character sequence associated with the keystroke.

If there is no conversion defined in the mapping object the emulator ignores the keystroke. If the keystroke was an ordinary alphabetic key, the emulator's `KeyPress` method will be eventually called by the terminal component with the character. At this point the emulator can send the character to the host computer. Hence, the set of mappings for the keyboard mapping object does not have to specify lower and upper case alphabetic, or numeric mappings.

```
procedure KeyPress(var Key : AnsiChar); virtual;
```

↳ Processes a key press message from the terminal component.

The terminal component performs no keystroke processing of its own. Instead it defers that job to the emulator component, where the knowledge of such processing for a given terminal is embodied.

The terminal component's `KeyPress` method—the standard VCL `KeyPress` virtual method for `TWinControl` descendants—merely calls the `KeyPress` method of its attached emulator. The `Key` parameter defines the key pressed.

The usual job of the emulator at this point is to send the character to the host computer by means of the terminal component's `ComPort` property.

```
procedure LazyPaint; virtual;
```

↳ Processes a lazy display paint request from the terminal component.

The terminal component performs no painting of its own. Instead it defers that job to the emulator component, where the knowledge of such processing for a given terminal is embodied.

There are two types of painting to be done. The first is painting because the new data that has come from the serial device needs to be shown (this could be triggered by the lazy display processing of the terminal, for example). The second is painting due to all or part of the terminal component's client window being invalidated and Windows has issued a WM_PAINT message.

The LazyPaint method is called in the former case. The emulator has been parsing the incoming data stream and altering the buffer to reflect the new text and other changes embedded in the stream. The terminal component has determined that the new changes need to be shown on the client window. There are three ways the terminal can make that determination. The first is when the emulator sets its NeedsUpdate property to True to denote that the terminal display has changed and the new data needs to be shown. The second is when the terminal component is in lazy display mode (its UseLazyDisplay property is True), and a certain number of bytes (given by the TAdTerminal.LazyByteDelay property) has been processed. The third is again when the terminal component is in lazy display mode and the required amount of time, given by the LazyTimeDelay property, has elapsed.

The emulator needs to interrogate its terminal buffer object to find out the changed character cells in the terminal display and to display them on the terminal component's canvas.

See also: BlinkPaint, Paint

NeedsUpdate**read-only, run-time property**

```
property NeedsUpdate : Boolean
```

↳ Defines whether the terminal display has changed.

The emulator sets its NeedsUpdate property to True when it makes a change to its terminal buffer object, for example, due to a new character being written to the terminal, or due to some other change, like scrolling, that has happened. The terminal component defined by the Terminal property will read this property every now and then, and if True will call the emulator's LazyPaint method so that the emulator can update the view of the terminal.

```
procedure Paint; virtual;
```

↳ Processes a paint request from the terminal component.

The terminal component performs no painting of its own. Instead it defers that job to the emulator component, where the knowledge of such processing for a given terminal is embodied.

There are two types of painting to be done. The first is painting because the new data that has come from the serial device needs to be shown (this could be triggered by the lazy display processing of the terminal, for example). The second is painting due to all or part of the terminal component's client window being invalidated and Windows has issued a WM_PAINT message. The Paint method is called in the latter case. The emulator should find out the clipping region of the invalidated window and redraw the text contained there, using the internal buffer object to define the colors, attributes, character sets and so on.

The Paint method of the emulator is called by the Paint method of the terminal component, the standard VCL virtual method for TWinControl descendants.

See also: BlinkPaint, LazyPaint

```
property Parser : TAdTerminalParser
```

↳ Accesses the emulator's terminal control sequence parser.

The Parser property is the engine that interprets the incoming data stream for the emulator. It is the parser that decides which sets of characters are terminal control sequences and which are merely characters to be displayed.

The emulator creates a parser object in its Create constructor and frees it in the Destroy destructor. Indeed, some emulator descendant classes do not require a parser object (for example, the TTY emulator) and hence do not ever create one. Hence, be aware that reading the Parser property may return nil.

The emulator will call the ProcessChar method of the parser—if there is one—for every character it receives from the terminal component. It will act on the return value of this call, making use of the parser's Command, Sequence, and Argument properties to alter the buffer object to reflect the new view of the terminal.

It is possible to replace the parser at run time. This will have the effect of altering the behavior of the emulator component. The class will dispose of the old parser and start using the new one. Be aware that it is possible to switch parsers while the old parser is in the middle of processing a terminal control sequence. In that case, the partially received control sequence is lost. It is best to replace the parser just after calling the old parser's `ProcessChar` method, if it does not return `pctPending`.

Terminal

property

```
property Terminal : TAdTerminal
```

↪ Defines the visual terminal component.

The Terminal property defines the terminal component whose canvas is used for drawing the terminal display, and which provides access to the keyboard for the emulator. The terminal component also hold the reference to the serial device through its `ComPort` property.

The terminal component and emulator component use each other to provide the correct functionality of the terminal emulation. The terminal component holds a link to the COM port that provides the incoming data stream, and passes all of this data directly to the emulator. The emulator, by use of its terminal parser instance, will interpret the data and modify the buffer (a `TAdTerminalBuffer` instance) to hold the new representation of the terminal display. The terminal component will also pass to the emulator all keystrokes entered by the user, so that the emulator, with the use of its keyboard mapping component, can identify the keystroke and convert it into the correct response to the host computer. Every now and then, especially if the terminal component is using lazy writing, the terminal will tell the emulator to draw a depiction of the current state of the terminal display on the terminal component's canvas.

If you set the Terminal property to nil, the emulator will detach itself from the previous terminal component. This component will then use its internal TTY emulator.

See also: `TAdTerminal.ComPort`, `TAdTerminal.Emulator`

The TAdTTYEmulator Class

The TAdTTYEmulator class emulates a teletype terminal, one that doesn't support any terminal control sequences and one that merely displays every character received. Similarly there is no conversion of keystrokes either: if a key for a displayable characters is pressed, that character is sent to the host without interpretation.

The following properties are nil for a TAdTTYEmulator instance: Parser, KeyboardMapping, and CharSetMapping. These objects are not required by the TTY emulator and so are never created. The TTY emulator does maintain a terminal buffer object.

Hierarchy

TComponent (VCL)

- ❶ TApdBaseComponent (OOMisc) 8
- ❷ TAdTerminalEmulator (ADTrmEmu) 253

TAdTTYEmulator (ADTrmEmu)

Properties

❷ Buffer	❷ NeedsUpdate	❶ Version
❷ CharSetMapping	❷ Parser	
❷ KeyboardMapping	❷ Terminal	

Methods

❷ BlinkPaint	❷ KeyDown	❷ LazyPaint
❷ GetCursorPos	❷ KeyPress	❷ Paint

The TAdVT100Emulator Class

The TAdVT100Emulator class emulates a Digital Equipment Corporation (DEC) VT100 terminal. It also supports some common extensions to the normal VT100 escape sequence set, namely support for multiple colors and support for inserting, erasing and deleting characters.

The TAdVT100Emulator class provides support for the standard VT100 terminal modes. These are as follows:

- Line feed/newline: whether the line feed character inserts a new line or merely advances the cursor to the next line with a possible scroll.
- Cursor key mode: whether the cursor movement keys send application mode sequences or cursor mode sequences.
- ANSI/VT52 mode: whether the terminal interprets ANSI escape sequences or the restricted VT52 sequences.
- Column mode: whether the terminal displays 80 or 132 characters across.
- Scrolling mode: whether the terminal jump scrolls or smooth scrolls. Although the VT100 emulator maintains this setting, all scrolling is performed with jump scrolls.
- Screen mode: whether the display is normal or reverse-imaged.
- Origin mode: whether the home position for the cursor obeys the current scrolling region or not.
- Wraparound mode: whether the terminal wraps the cursor to column 1 of the next line when a character is displayed in the final column.
- Auto repeat: whether keys auto-repeat or not when held down.
- Interface mode: whether the terminal displays with 240 or 480 scanlines. Although the VT100 emulator maintains this setting, it does not perform any action when it changes.
- Graphic processor option: whether the terminal uses its GPO. Although the VT100 emulator maintains this setting, it does not perform any action when it changes.
- Keypad mode: whether keys on the numeric keypad send numeric characters or escape sequences.

There are several escape sequences sent by the host computer to which the VT100 terminal must respond. The VT100 emulator sends the responses shown in Table 8.2.

Table 8.2: *VT100 emulator escape sequence responses*

Request	Response
Cursor position report	The position of the cursor (<Esc>[row ; col R).
Status report	Terminal OK (<Esc>[0n).
What Are You?	Base VT100, no options (<Esc>[?;0c).
VT52 Identity request	VT100 acting as VT52 (<Esc>/Z).

Hierarchy

TComponent (VCL)

- ① TApdBaseComponent (OOMisc) 8
- ② TAdTerminalEmulator (ADTrmEmu) 253
- TAdVT100Emulator (ADTrmEmu)

8

Properties

ANSIMode	② KeyboardMapping	① Version
AppKeyMode	LEDs	WrapAround
AppKeyPadMode	② NeedsUpdate	② BlinkPaint
AutoRepeat	NewLineMode	② GetCursorPos
② Buffer	② Parser	② KeyDown
② CharSetMapping	RelOriginMode	② KeyPress
Col132Mode	RevScreenMode	② LazyPaint
GPOMode	SmoothScrollMode	② Paint
Interlace	② Terminal	

Methods

② BlinkPaint	② KeyDown	② LazyPaint
② GetCursorPos	② KeyPress	② Paint

Reference Section

ANSIMode

run-time property

```
property ANSIMode : Boolean
```

↪ Determines whether the terminal acts on ANSI or VT52 sequences.

The VT100 terminal can act on two different and separate sets of terminal control sequences. If ANSIMode is True, the parser will only interpret ANSI escape sequences, and VT52 sequences are ignored. If False, the parser only understands the restricted VT52 command sequence set and all ANSI escape sequences are ignored.

It is the host computer that determines this mode. Although you can change this property it is inadvisable to do so: bizarre displays may result.

AppKeyMode

run-time property

```
property AppKeyMode : Boolean
```

↪ Determines which sequences are sent for the cursor movement keys.

The VT100 terminal can send two different escape sequences for the cursor movement keys (the arrow keys). DEC documentation calls them application mode and cursor key mode.

It is the host computer that determines this mode. Although you can change this property it is inadvisable to do so: it may result in the host computer not understanding keystrokes.

AppKeypadMode

run-time property

```
property AppKeypadMode : Boolean
```

↪ Determines which sequences are sent for the numeric keypad.

The VT100 terminal can send two different escape sequences for the numeric keypad (including the PF keys). DEC documentation calls them application keypad mode and numeric keypad mode.

It is the host computer that determines this mode. Although you can change this property it is inadvisable to do so; it may result in the host computer not understanding keystrokes.

AutoRepeat

run-time property

```
property AutoRepeat : Boolean
```

- ↳ Determines whether held down keys auto-repeat.

The VT100 terminal supports turning off auto-repeat for keys held down on the keyboard. In general, auto-repeat is on.

On the VT100 the user most often sets this mode, not the host computer.

Col132Mode

run-time property

```
property Col132Mode : Boolean
```

- ↳ Determines whether to display 80 or 132 characters across the display.

The VT100 terminal can support two display widths: 80 characters or 132 characters across. If Col132Mode is True, the terminal is displaying at 132-column resolution. If False, the norm, the terminal is displaying at 80-column resolution.

It is the host computer that determines this mode. Although you can change this property it is inadvisable to do so: it may result in bizarre displays since the host is assuming something that is no longer true.

GPOMode

run-time property

```
property GPOMode : Boolean
```

- ↳ Determines whether the graphics processor option is active.

Although maintained, the VT100 emulator does nothing with this property. Note that the VT100 emulator will respond to a “What are you?” request with a “basic VT100 with no options” reply, which disables any possibility of using the graphics processor option.

Interlace

run-time property

```
property Interlace : Boolean
```

- ↳ Determines whether to use 240 or 480 scanlines.

The VT100 terminal can display using 240 scanlines (True) or 480 (False). Although maintained, the VT100 emulator does nothing with this property.

property LEDs : Integer

↳ Returns the state of the LEDs on the VT100 keyboard.

The VT100 keyboard has a set of four LEDs embedded in the face of the keyboard. The host computer can set them on or off if required. The LEDs property returns an integer value as a binary representation of the current state of the LEDs: if the first LED is on, bit 0 is set in the property value; if the second LED is on, bit 1 is set and so on.

The LEDs are only visual, they do not affect the use of the keyboard or functionality of the terminal at all. Indeed, the host computer cannot even read their state.

NewLineMode**run-time property**

property NewLineMode : Boolean

↳ Determines the action of a line feed character.

A line feed character (hex 10) received by a VT100 terminal can do one of two things. If this property is False, the cursor is advanced one row down, keeping in the same column, scrolling the display up if necessary. The Enter key sends a single <CR> character.

If this property is True, the cursor is advanced one row down, moving to the first column, scrolling the display up if necessary. The Enter key sends a pair of characters: <CR><LF>.

It is the host computer that determines this mode. Although you can change this property it is inadvisable to do so: it may result in bizarre displays since the host is assuming something that is no longer true.

RelOriginMode**run-time property**

property RelOriginMode : Boolean

↳ Determines where the home position is to be found.

If this property is False, the cursor home position is at the top left corner of the display, even if a scrolling region has been defined. Rows and columns are counted from this origin (this can be viewed as *absolute position mode*). Rows and columns are always counted from 1. Cursor position commands can act outside the active scrolling region.

If this property is True, the cursor home position is at the top left corner of the scrolling region. Rows and columns are counted from this origin (this can be viewed as *relative position mode*—positions are relative to the scrolling region). Rows and columns are always counted from 1. Cursor position commands only act inside the active scrolling region.

It is the host computer that determines this mode. You can change this property to write text outside of the scrolling region, but it is advisable to change it back immediately; otherwise, it may result in bizarre displays since the host is assuming something that is no longer true.

RevScreenMode

run-time property

```
property RevScreenMode : Boolean
```

↳ Determines whether the display is in reverse image or not.

If this property is False, normal text is shown as white on black. If True, normal text is shown in reversed: as black on white.

On the VT100 the user most often sets this mode, not the host computer, although the host can change the mode if it wishes.

8

SmoothScrollMode

run-time property

```
property SmoothScrollMode : Boolean
```

↳ Determines whether scrolling is smooth or jumps.

The VT100 terminal can scroll the display in two modes: a smooth scroll, or a jump scroll. Although maintained, the VT100 emulator does nothing with this property.

WrapAround

run-time property

```
property WrapAround : Boolean
```

↳ Determines whether displayed text wraps at the right margin.

If this property is False, text received when the cursor is at the right hand side does not cause the cursor to be moved to the first column of the next row.

If this property is True, text received when the cursor is at the right hand side will cause the cursor to be moved to the first column of the next row to display all the text.

It is the host computer that determines this mode. It is inadvisable to change this mode, otherwise it may result in bizarre displays since the host is assuming something that is no longer true.

The TAdTerminal Component

The TAdTerminal component represents the visual part of a terminal. It is the only visual component in the Async Professional family of terminal classes. It is responsible for maintaining the window handle and for performing the low-level processing to get all the possible keystrokes available on a PC keyboard.

Apart from this, it performs next to no work itself, handing off most of the display and other capabilities to an emulator component (a descendant of TAdTerminalEmulator). Essentially, the terminal component acts as a conduit between the PC screen and keyboard and the other classes that perform all of the work.

The main behaviors of the terminal component and its associated objects are as follows:

- Receives characters from the serial device and passes them onto the emulator for interpretation and processing.
- Traps all keystrokes and passes them onto the emulator for conversion into their terminal equivalents.
- The emulator maintains a buffer of several matrices, one for each of text, background color, text color, attributes, and character sets. This buffer stores the information needed to show the visual representation of the terminal.
- The terminal has the capability of maintaining a scrollbar buffer to enable the user to scroll back through old data that has scrolled off the display. The terminal will automatically show scrollbars in that case.
- The terminal will automatically display scrollbars if the client window is smaller than the terminal display.
- If the terminal component is dropped onto a form and there is no existing emulator component on that form, the terminal will create an internal instance of a TTY (teletype) emulator. This means that the terminal component can be used as is for simple tasks.

In order for the terminal's window to show data, there are two objects that must be connected to it: the COM port (a TAPdComPort component), and an emulator (a TAdTerminalEmulator descendant component). To use a terminal component, you would drop a terminal, a COM port, and a terminal emulator onto the form and connect them up by setting the ComPort and Emulator properties of the terminal. Once they are connected up in this fashion, you can set the terminal's Active property to True and start using the terminal.

The COM port provides the terminal with incoming bytes from a serial communications device, such as a serial port or Winsock layer. The terminal, in turn, provides the data that must be output to the serial device. In and of itself, the terminal component doesn't know what to do with the incoming data; it has no built in knowledge of terminal control sequences and how to separate the text from them. Hence, it passes the incoming data stream directly to the terminal emulator component. This component in turn would use a parser object to identify terminal control commands, a buffer object to store the displayable data, and so on. The terminal would also pass all keyboard input to the emulator as well, so that the emulator can process the keystrokes and determine what to pass back to the host (it would use a keyboard mapping object to do this). The emulator will use the COM port of the terminal component to pass back data through the serial device to the host computer.

As for display, the emulator would assume control over the window handle of the terminal component and draw directly onto its canvas.

The reason for this more complex design—rather than rolling the emulator and terminal components into one—is flexibility. With this design it is possible to switch emulators on the fly without having to destroy the window handle used by the terminal and thus avoiding flicker. Hence, you can easily start off with a TTY emulator for the terminal, and switch to a VT100 emulator later on. The terminal component thus stores generic information about a terminal (e.g., the number of characters across and down), whereas the emulator component holds specific information about a specific terminal (e.g., whether to accept VT52 or ANSI sequences for a VT100 terminal).

Figure 8.2 shows the interrelationship between the client window, the terminal display and the scrollbar buffer.

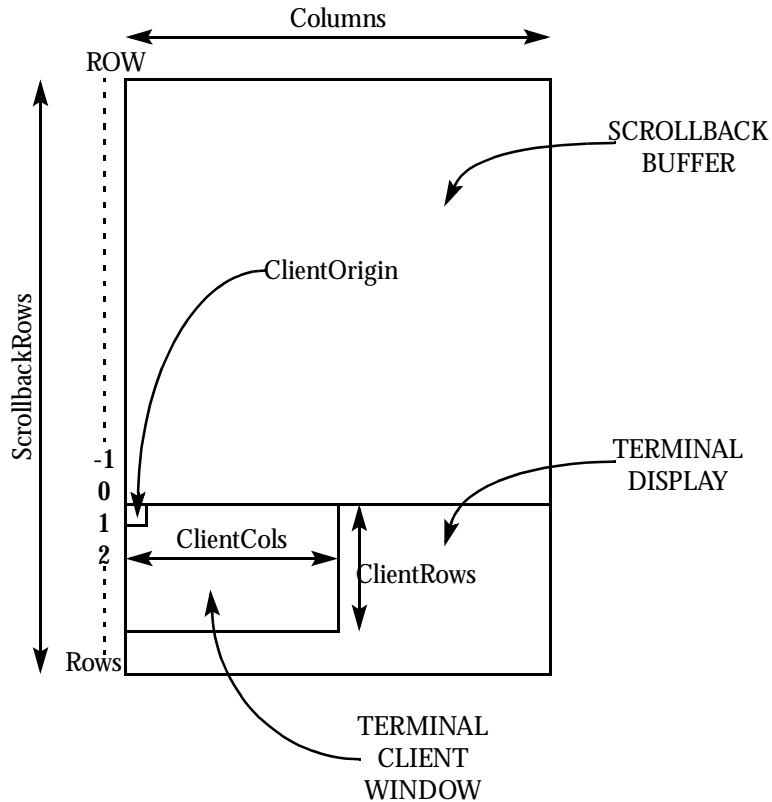


Figure 8.2: Interrelationship between the client window, the terminal display, and the scrollbar buffer.

The client window acts as a viewport on the terminal display. In general, you would make sure that the client window was large enough to display all of the data in the terminal display. If that is so, no scroll bars are shown. Should the client window be smaller vertically, or horizontally, or both, than the terminal display, the relevant scrollbars are automatically shown to enable the user to scroll though and view the entire terminal display.

If scrollbar mode is activated, the client window acts as a viewport on the entire scrollbar buffer, not just on the terminal display. Scrollbars would automatically be shown, if required, to enable the user to scroll through the entire buffer. In scrollbar mode, the terminal component no longer processes any data from the serial device nor does it trap keyboard messages and pass them on to the emulator.

The terminal component also supports a lazy write mode. In this mode, incoming data is processed but not displayed until one of two conditions occurs: either a certain number of bytes is received, or a certain amount of time has elapsed. Once the required time has gone by or the number of bytes has been received, the terminal component will force its display to be updated, and the process starts all over again. Fine tuning of these two properties will result in a terminal that is more responsive, and has more efficient and smoother behavior.

Font handling

The issue of fonts becomes complex when dealing with a terminal that should support displaying non-standard glyphs for characters. For example, when emulating the VT100 terminal, the standard Windows fixed pitch fonts do not have every single glyph. The CharSet mapping class attempts to ameliorate the situation by providing a mapping from a character in a character set to a glyph in a Windows font. Apart from the simple TTY emulation, there will unfortunately be several fonts in action for the usual terminal emulations. The ability to switch fonts easily as with simple components cannot apply in the case of the terminal. The terminal's situation is entirely more complex: text displayed may contain glyphs from several unrelated fonts.

What the terminal component and its associated classes attempt to do is to make the displayed glyphs the same point size. The point size of the fonts used by the terminal is taken to be the same as the Size property of its Font property. Hence, to change the size of the text displayed by the terminal, you would need to change the Font.Size property. For certain emulations, the terminal's Font property will in fact determine the font with which the text is displayed—in other words, there is no CharSet mapping required.

The terminal classes will allow the use of proportional fonts. However, it should be noted that text displayed using these proportional fonts will be shown in fixed size character cells, and proportional fonts displayed in fixed pitch can look awkward and clunky. Our advice is to try and stick to fixed pitch fonts wherever possible.

Capturing data

The terminal component has a built-in facility to capture data, that is, to write all data received by the serial port to a capture file. The capture file can be used for later review of an on-line session.

The data is written to the capture file before the emulator has an opportunity to see the data stream and to parse out the terminal control sequences. Hence, the capture file is not a simple text file, it could consist of numerous control characters and escape sequences as well as normal text.

Characters typed at the keyboard are not saved in the capture file. However, such characters usually end up in the capture file anyway, since the host computer echoes them back to the terminal.

Using the clipboard

The terminal window provides support for copying displayed text to the Windows clipboard. Because the text in a terminal window is read-only, cutting and pasting are not supported. Block marking using the cursor keys is also not supported by the terminal window, since it implies that you can move the caret and in general it is the remote computer that controls the location of the caret.

The terminal window does support the following:

- Normal mode block marking using the mouse.
- Scrollback mode block marking using the mouse.
- Copying the marked block to the clipboard.

When in normal mode, only the visible contents of the terminal window can be marked. When in scrollback mode, the visible contents can be marked and, by moving the cursor above or below the window, the window can be scrolled to allow any part of the scrollback buffer to be marked.

You can copy the marked block to the clipboard by calling the terminal window routine `CopyToClipboard`.

- ☛ **Caution:** Although the `CopyToClipboard` method will copy the marked text to the clipboard, you should be aware that what you see may not be what you get. The terminal classes in Async Professional support the notion of different character sets. In other words, with certain emulations, the glyphs that are displayed on the terminal display may seem to have no connection with the characters that are actually there. For example, with a VT100 terminal, when using the USASCII character set the character 'm' will be displayed as a lowercase 'm'. However, when using the special graphics character set, the character 'm' is rendered as the lower left corner of the line draw set (the glyph that looks like an 'L'). The problem is that, when you copy the marked text to the clipboard, you will lose the character set definition for each character. Hence, you will just get the character 'm' in the clipboard and not know whether it really was shown as an 'm' or some other glyph.

Hierarchy

TWinControl (VCL)

❶ TApdBaseWinControl (OOMisc)	8
TAdTerminal (ADTrmEmu)	

Properties

Active	CharWidth	Line
Attributes	Columns	Rows
BackColor	ComPort	ScrollbackRows
BlinkTime	Emulator	Scrollback
Capture	ForeColor	UseLazyDisplay
CaptureFile	HalfDuplex	❶ Version
CharHeight	LazyByteDelay	WantAllKeys
CharSet	LazyTimeDelay	

Methods

Clear	CreateWnd	DestroyWnd
ClearAll	Create	WriteChar
CopyToClipboard	Destroy	WriteString

Events

OnCursorMoved

Reference Section

Active

property

property Active : Boolean

Default: True

↳ Determines whether the terminal is accepting serial events.

Setting Active to True causes the terminal to start processing serial and keyboard data and to display this information in the terminal window. At run time, if an attempt is made to set Active True without a serial port component being assigned to the ComPort property, or without an emulator component being assigned to the Emulator property, the attempt is ignored. It is acceptable to set Active to True at design time; this causes the terminal to start processing events automatically when the form is created at run time.

You must set Active to False if the terminal is being used in combination with another component, such as a file transfer component, that needs exclusive access to some or all of the data stream. During a file transfer, set Active to False for the duration of the transfer and True when the transfer is over and data should appear in the terminal window again.

The following example sets Active to False as it starts receiving a file to prevent the terminal from displaying the received data. An OnProtocolFinish event re-enables the terminal window once the protocol informs the application that the transfer is complete.

```
AdTerminal1.Active := false;
ApdProtocol1.StartReceive;
...
procedure TMyForm.ProtocolFinish(
  CP : TObject; ErrorCode : Integer);
begin
  AdTerminal1.SetFocus;
  AdTerminal1.Active := true;
end;
```

See also: ComPort, Emulator

```
property Attributes[aRow, aCol : Integer] : TAdTerminalCharAttrs  
TAdTerminalCharAttr = (tcaBold, tcaUnderline, tcaStrikethrough,  
    tcaBlink, tcaReverse, tcaInvisible);  
TAdTerminalCharAttrs = set of TAdTerminalCharAttr;
```

↪ Accesses the attributes of text in the display.

Attributes enables the direct manipulation of the attributes for characters displayed by the terminal. Attributes is an array property indexed by a combination the row number (aRow) and the column number (aCol). Both aRow and aCol are one-based: the home position of the terminal display is at row 1 column 1. Note, however, that if you have a scrollbar buffer that aRow can take on negative values, as well, to identify non-visible rows in the scrollbar buffer.

The result value is a set of possible attributes. They are tcaBold for bold text; tcaUnderline for underlined text; tcaStrikethrough for text that has a line through it, as if it had been deleted (~~this text is struck through~~); tcaBlink for blinking text; tcaReverse for reversed text; and tcaInvisible for text that is not visible.

The following example sets all text on row 5 to blinking:

```
for I := 1 to AdTerminal1.Columns do begin  
    AdTerminal1.Attributes[5, I] :=  
        AdTerminal1.Attributes[5, I] + [tcaBlink];  
end;
```

However, do notice that this direct manipulation is fairly inefficient.

See also: BackColor, CharSet, ForeColor

BackColor

run-time, array property

```
property BackColor[aRow, aCol : Integer] : TColor
```

↪ Accesses the background color of text in the display.

BackColor enables the direct manipulation of the background color for characters displayed by the terminal. BackColor is an array property indexed by a combination the row number (aRow) and the column number (aCol). Both aRow and aCol are one-based: the home position of the terminal display is at row 1 column 1. Note, however, that if you have a scrollbar buffer that aRow can take on negative values, as well, to identify non-visible rows in the scrollbar buffer.

However, do notice that this direct manipulation is fairly inefficient.

See also: Attributes, CharSet, ForeColor

BlinkTime

property

```
property BlinkTime : Integer
```

Default: 500

↪ Defines the time in milliseconds between cycles for blinking text.

Some terminals enable text displayed by the terminal to be blinking. The BlinkTime property defines the elapsed time for a full cycle for the text being displayed, being invisible, and being displayed again.

Note that, to provide this functionality, the terminal sets up a timer to tick at this rate. A Windows timer is low-priority; if the PC is performing other work, it will seem as if the blinking text has either stopped blinking or has disappeared completely. Also, if you set the BlinkTime property too low, the terminal and emulator will spend most of their time updating the window, especially if there's a lot of blinking text.

Capture

property

```
property Capture : TAdCaptureMode
```

```
TAdCaptureMode = (cmOff, cmOn, cmAppend);
```

Default: cmOff

↪ Defines whether the data received by the terminal is captured to file.

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

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

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

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

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

If the `CaptureFile` property has not been set to the name of a file (i.e., it is the empty string), setting `Capture` to `cmOn` or `cmAppend` will have no effect. The attempt will be ignored and no exception will be raised. If the `CaptureFile` property has been set, an attempt is made to create or open the file so named. This operation can of course fail for any of a number of different reasons. Internally, the terminal uses a file stream—`TFileStream`—to access the file, so any I/O exceptions raised will be those used by the standard Delphi class.

See also: `CaptureFile`

CaptureFile

property

```
property CaptureFile : string
```

Default: "APRO.CAP"

↪ Defines the name of the file where the terminal writes captured data.

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

See also: `Capture`

```
property CharHeight : Integer
```

↳ Defines the height in pixels of a character cell for the terminal.

The terminal display can be viewed as a matrix of fixed-sized character cells. There are Rows cells vertically and Columns cells horizontally. The size of these character cells is calculated as the maximum required to display all of the possible characters in all of the possible character sets. Since different characters in different character sets could be displayed using different Windows fonts, the character cell size can be viewed as a function of the different possible Windows fonts used by the emulator. The value of CharHeight is the maximum needed to display any glyph.

The overall size of the characters displayed by the terminal is given by the Size property of the Font property of the terminal component. The Font property is used as the default font for those terminal emulations that don't support character sets. For terminals that do support character sets, the size of the glyphs is given by the Font.Size value.

See also: CharWidth

```
property CharSet[aRow, aCol : Integer] : Byte
```

↳ Accesses the character set of text in the display.

The Async Professional terminal supports the notion of different character sets for different text on the display. Normally the text is shown in one character set only, but in certain cases (for example, the line draw glyphs with the VT100 terminal) the glyphs shown on the terminal are drawn from other character sets. Rather than define an enumerated type or class for each character set used by the terminal, character sets are identified by an anonymous byte value. It is up to the emulator to define which byte value represents which character set.

CharSet enables the direct manipulation of the character set for characters displayed by the terminal. CharSet is an array property indexed by a combination the row number (aRow) and the column number (aCol). Both aRow and aCol are one-based; the home position of the terminal display is at row 1 column 1. Note, however, that if you have a scrollbar buffer that aRow can take on negative values, as well, to identify non-visible rows in the scrollbar buffer.

However, do notice that this direct manipulation is fairly inefficient.

See also: Attributes, BackColor, ForeColor

```
property CharWidth : Integer
```

↳ Defines the width in pixels of a character cell for the terminal.

The terminal display can be viewed as a matrix of fixed-sized character cells. There are Rows cells vertically and Columns cells horizontally. The size of these character cells is calculated as the maximum required to display all of the possible characters in all of the possible character sets. Since different characters in different character sets could be displayed using different Windows fonts, the character cell size can be viewed as a function of the different possible Windows fonts used by the emulator. The value of CharWidth is the maximum needed to display any glyph.

The overall size of the characters displayed by the terminal is given by the Size property of the Font property of the terminal component. The Font property is used as the default font for those terminal emulations that don't support character sets. For terminals that do support character sets, the size of the glyphs is given by the Font.Size value.

See also: CharHeight

Clear**method**

```
procedure Clear;
```

↳ Clears the terminal display.

To clear the display, the terminal component will internally scroll the window up by Rows lines. This means that the current display will scroll into the non-visible part of the scrollback buffer and can be viewed in scrollback mode. The top lines of the scrollback buffer will of course disappear, being completely scrolled off the top of the buffer.

(The Clear method is the equivalent of the ClearWindow method of the deprecated TApdTerminal.)

See also: ClearAll

```
procedure ClearAll;
```

✚ Clears the entire scrollbar buffer including the terminal display.

To clear the scrollbar buffer, the terminal sets all characters in the buffer to the space character; all attribute values to the empty set (equivalent to “normal” text); all background color values to the Color property; all foreground color values to Font.Color; all charset values to 0.

(The ClearAll method is the equivalent of the ClearBuffer method of the deprecated TApdTerminal.)

See also: Clear

Columns

property

```
property Columns : Integer
```

Default: 80

8

✚ Defines the number of columns across the terminal display.

The value of the Columns property is the number of standard-sized characters that can be written across the terminal display. In general, it is a value like 80, but in certain circumstances it could be 132. If the original terminal supports double-width characters, the value of Columns reflects that for standard-sized characters, not the double-width ones.

Altering the value of Columns will cause the underlying buffer to be resized. The terminal will attempt to save as much of the original data as possible during the resize operation.

Setting the value of Columns to less than that supported by the original terminal itself is liable to produce funny looking displays, since the host computer will assume that the terminal is the correct size and position text accordingly.

If the host computer switches the terminal into a mode with a different number of columns (say, from 80 to 132 characters across), the value of Columns will change to reflect that switch.

☛ **Caution:** Because of the structure of the buffer class, in Delphi 1, the product of Columns and ScrollbackRows cannot exceed 16,383. For an 80 column display, that means the number of scrollbar rows cannot exceed 204. There is no such limitation with 32-bit compilers.

```
property ComPort : TApdCustomComPort
```

↳ Defines the serial device to which the terminal is connected.

ComPort is usually set automatically at design time to the first TApdCustomComPort descendant component the terminal component finds on the form. If necessary, use the Object Inspector to select a different TApdCustomComPort descendant component.

Setting the ComPort property at run time is necessary only when using a dynamically created TApdCustomComPort object or when selecting among several TApdCustomComPort descendant components.

Note that setting the value of ComPort may not be enough to get the terminal to display data. You must also activate the terminal by ensuring that Active is set to True, and you must also supply a terminal emulator component by setting the Emulator property to process the incoming data stream.

See also: Active, Emulator

CopyToClipboard**method**

```
procedure CopyToClipboard;
```

↳ Copies the marked block to the clipboard.

This routine copies the currently marked block to the Windows clipboard. The block is copied in CF_TEXT format, where a carriage return/line feed follows each line.

Please read the caution within the section entitled “Using the clipboard” on page 273 in the introduction to the TAdTerminal component. This discusses why, after calling this method, the clipboard may not contain what you see on the display.

Create**constructor**

```
constructor Create(aOwner : TComponent); override;
```

↳ Creates an instance of the terminal component.

The constructor will also create an instance of a teletype emulator (TTY emulator) for use when the Emulator property is not set. This ensures that the terminal at least functions as a TTY terminal without any extra work.

See also: Destroy

CreateWnd

method

```
procedure CreateWnd; override;
```

- ↳ Creates the window handle for the terminal component.

Once the window handle has been created, this method installs a keyboard hook in order to trap all keystrokes made by the user.

See also: DestroyWnd

Destroy

destructor

```
destructor Destroy; override;
```

- ↳ Destroys an instance of the terminal component.

The destructor will free all memory and resources allocated by the instance. Of particular note is that if data is being captured when Destroy is called, the capture file is closed properly first.

See also: Create

8

DestroyWnd

method

```
procedure DestroyWnd; override;
```

- ↳ Destroys the window handle of the terminal component.

If a keyboard hook was properly installed by the CreateWnd procedure, DestroyWnd will remove it before destroying the window handle associated with the terminal component.

See also: CreateWnd

Emulator

property

```
property Emulator : TAdTerminalEmulator
```

- ↳ Defines the terminal emulator that processes incoming data.

The terminal component is merely a conduit between the serial device defined by the ComPort property and the terminal emulator defined by Emulator. The terminal component also provides the window handle, and hence, the canvas, that the emulator will use to draw the terminal display.

The emulator processes the input data stream looking for terminal control sequences. It will then use the terminal component's window handle to paint a representation of the terminal screen. The terminal component also traps keystrokes and passes them to the emulator. The emulator will then convert these keystrokes into valid messages or sequences for the host computer and use the terminal's ComPort to pass them back.

The Emulator property is usually set automatically at design time to the first TAdTerminalEmulator descendant component the terminal component finds on the form. If necessary, use the Object Inspector to select a different TAdTerminalEmulator descendant component.

Usually, setting the Emulator property at run time is necessary only when switching between different terminal emulations. The terminal does not clear the display at such times, it is up to the new terminal emulator to reset its buffer and to display the cleared screen.

Note that setting the value of Emulator may not be enough to get the terminal to display data. You must also activate the terminal by ensuring that Active is set to True, and you must also supply a COM port component by setting the ComPort property to an appropriate TApdCustomComPort component.

See also: Active, ComPort

ForeColor

run-time, array property

```
property ForeColor[aRow, aCol : Integer] : TColor
```

↪ Accesses the foreground color of text in the display.

ForeColor enables the direct manipulation of the foreground color (the color of the glyphs themselves) for characters displayed by the terminal. ForeColor is an array property indexed by a combination the row number (aRow) and the column number (aCol). Both aRow and aCol are one-based: the home position of the terminal display is at row 1 column 1. Note, however, that if you have a scrollbar buffer that aRow can take on negative values, as well, to identify non-visible rows in the scrollbar buffer.

See also: Attributes, BackColor, CharSet

HalfDuplex

property

property HalfDuplex : Boolean

Default: False

↳ Determines whether local data is echoed to the terminal display.

If HalfDuplex is False (the default), data entered at the local keyboard is displayed only if the remote host computer echoes it back.

If HalfDuplex is True, data entered at the local keyboard is automatically displayed in the terminal window. If the host computer is echoing the input data back as well, each character is displayed twice.

LazyByteDelay

property

property LazyByteDelay : Integer

Default: 128

↳ Determines the number of bytes received before the display is forcibly repainted.

The TAdTerminal supports a lazy writing mode. When this mode is active, rather than update the display every time a new character appears from the serial device, the terminal and its associated classes will only display new data after a certain amount of time, or after a certain number of bytes have been received, or both. This gives the terminal a more efficient and smoother feel. The value of LazyByteDelay defines how many bytes must be received before the terminal window is updated. The default value is a compromise between efficiently handling the display and providing timely visual feedback to the user.

See also: LazyTimeDelay, UseLazyWrite

LazyTimeDelay

property

property LazyTimeDelay : Integer

Default: 250

↳ Determines the number of elapsed milliseconds before the display is forcibly repainted.

The TAdTerminal supports a lazy writing mode. When this mode is active, rather than update the display every time a new character appears from the serial device, the terminal and its associated classes will only display new data after a certain amount of time, or after a certain number of bytes have been received, or both. This gives the terminal a more efficient and smoother feel. The value of LazyTimeDelay defines how many milliseconds must pass before the terminal window is updated. The default value is a compromise between efficiently handling the display and providing timely visual feedback to the user.

See also: LazyByteDelay, UseLazyWrite

```
property Line[aRow : Integer] : string
```

↳ Accesses the text data for a row in the display.

Line returns and sets the characters that make up a row in the terminal. It enables the direct manipulation of the text displayed by the terminal. Line is an array property indexed by the row number (aRow). aRow is one-based: the home position of the terminal display is at row 1 column 1. Note, however, that if you have a scrollbar buffer that aRow can take on negative values, as well, to identify non-visible rows in the scrollbar buffer.

However, do notice that this direct manipulation is fairly inefficient.

💡 **Caution:** Line returns the character values that make up a row. For some terminals, the glyph you see on the terminal for a particular character value is not only based on the character value itself, but also on the character set that is being used to display that character. For example, on a VT100 terminal, if the character 'm' is displayed using the USASCII character set, you will see the usual lower case 'm' glyph on the display. However, if the same character 'm' is displayed using the Special Characters character set, you will see the lower left corner linedraw glyph (the one that looks like an 'L'). All the Line property will return is the 'm' character value at that particular column position.

See also: Attributes, BackColor, CharSet, ForeColor

```
property Rows : Integer
```

Default: 24

↳ Defines the number of rows down the terminal display.

The value of the Rows property is the number of standard-sized characters that can be written vertically on the terminal display. In general, it is a value like 24 or 25. If the original terminal supports double-height characters then the value of Rows still reflects that for standard-sized characters, not the double-height ones.

Notice that Rows is the number of rows on the original terminal, not the number of rows in the scrollbar buffer. The terminal display will consist of Rows lines, with Columns characters in each.

Altering the value of Rows may cause the underlying buffer to be resized. The terminal will attempt to save as much of the original data as possible during the resize operation. The data will be preserved from the bottom of the terminal upwards. In other words, if you reduce the number of rows from 20 to 15, say, you will see the bottom 15 rows of the original display after the operation completes, not the top 15.

Setting the value of Rows to less than that supported by the original terminal itself is liable to produce funny looking displays, since the host computer will assume that the terminal is the correct size and position text accordingly.

The rows in the terminal display are counted from 1, with the top row of the terminal being row 1.

Scrollbar

property

```
property Scrollback : Boolean
```

Default: False

↳ Defines whether the terminal is in scrollbar mode.

If Scrollback is set True, the terminal is placed into scrollbar mode. In this mode, keystrokes are no longer translated into their terminal equivalents and instead serve to navigate through the scrollbar buffer. Hence, in scrollbar mode, the Page Up/Page Down keys will move the user through the scrollbar buffer, as will the standard arrow keys.

If scrollbar mode is activated, the terminal will also no longer receive data from the serial device. It is recommended that you impose flow control on the COM port when you switch into scrollbar mode (either send an XOFF character or drop a hardware flow control signal), to help avoid the dispatcher's input buffer overflowing. The terminal does not do this itself. Imposing flow control helps keep the terminal data stable to allow the user to navigate through the scrollbar buffer in a profitable manner. Obviously, when you leave scrollbar mode, you would end the flow control condition to allow more data to come through and be processed.

See also: ScrollbackRows

```
property ScrollbackRows : Integer
```

Default: 200

✚ Defines the number of rows in the scrollback buffer.

The scrollback buffer consists of the visible part of the terminal display, together with the previous data that has scrolled off the top of the terminal display. In general you would set ScrollbackRows such that you could hold four or five screens' worth of previous data.

The value of the ScrollbackRows property must be greater than or equal to the value of Rows. If you attempt to set ScrollbackRows to a value less than Rows, the new value is adjusted to be equal to Rows. No exception is generated in this situation. If the original terminal supports double-height characters then the value of ScrollbackRows still reflects that for standard-sized characters, not the double-height ones.

Altering the value of ScrollbackRows may cause the underlying buffer to be resized. The terminal will attempt to save as much of the original data as possible during the resize operation. If the value of ScrollbackRows is reduced the data is removed from the top of the buffer rather than the bottom.

The rows in the terminal display are counted from 1, with the top row of the terminal being row 1. The rows above the actual terminal display in the scrollback area are counted backwards from 1. Hence, the row above the top row of the actual terminal display is row 0, the one above that row -1, and so on.

UseLazyDisplay

property

```
property UseLazyDisplay : Boolean
```

Default: True

✚ Defines whether the terminal immediately displays new incoming data or not.

The TAdTerminal supports a lazy writing mode. When this mode is active, rather than update the display every time a new character appears from the serial device, the terminal and its associated classes will only display new data after a certain amount of time, or after a certain number of bytes have been received, or both. This gives the terminal a more efficient and smoother feel.

If UseLazyDisplay is False, the terminal will display every incoming character as and when it arrives. If UseLazyDisplay is True, the terminal will display the new data on the screen after LazyByteDelay bytes have been received since it last updated the window, or after LazyTimeDelay milliseconds have elapsed.

The lazy writing mode only applied to data written to the terminal, either from the serial device, or from the keyboard in half duplex mode, or from data explicitly written from calling `WriteChar` or `WriteString`. If the terminal component's window is invalidated due to another window covering it and then being moved, or from the application being minimized and then restored, the terminal display is immediately repainted.

See also: `LazyByteDelay`, `LazyTimeDelay`

WantAllKeys

property

property `WantAllKeys` : Boolean

Default: True

↳ Defines whether the terminal component hooks and retrieves all keystrokes.

Part of the job of a terminal component is the ability to map the PC keyboard onto a terminal keyboard. This latter keyboard might be a completely different layout than the PC keyboard and, apart from the alphabetic key section, have different keys for different host functions. The keyboard mapping should attempt to match PC keys (whether they are alt-shifted, ctrl-shifted, or whatever) onto appropriate terminal keys.

8

A problem that will occur is that keys like F1, F10, Enter, Tab, and so on, have a well-defined meaning in the Windows world. Normally, controls on a form would ignore these keys since they have dialog-specific or application-wide meanings. However, for a terminal component it often makes sense to have these keys perform a terminal related function and to suppress the standard Windows meaning. If `WantAllKeys` is True, the terminal component will attempt to hook and trap all keystrokes generated while it has focus. Hence, for example, F1 will not bring up the help system (it will not cause a `WM_HELP` message to be sent to the control), F10 will not activate the main menu of the application, and so on.

If `WantAllKeys` is False, the terminal component will not perform anything special with regard to the keyboard. It will just trap `WM_KEYDOWN` and `WM_SYSKEYDOWN` messages and pass them on to the emulator for processing. Standard Windows keys will perform their usual functions.

```
procedure WriteChar(aCh : AnsiChar);
```

↳ Writes a single character to the terminal.

The character written to the terminal will go through the same steps that a character that had arrived from the serial device would go through. In other words, the character is first passed to the emulator, which decides what to do with it. If the emulator decides that the character is part of a terminal control sequence, it would appear as if the character had not been accepted—it would not appear on the display—when in reality it had.

The terminal will accept a character written with `WriteChar` at any time, even when it is actively receiving data from the serial device. Be aware that under these circumstances, the character written with `WriteChar` will intermingle with data from the serial device and may cause some bizarre behavior and displays.

Note also that the lazy write mode still applies to text written to the terminal with `WriteChar`. If `UseLazyDisplay` is `True`, the text will appear at the appropriate time. You can force the new data to be displayed in this case by calling the `Update` method.

WriteString**method**

```
procedure WriteString(const aSt : string);
```

↳ Writes a string to the terminal.

The string written to the terminal will go through the same steps that characters that have arrived from the serial device would go through. In other words, the characters in the string are first passed to the emulator, which decides what to do with them. If the emulator decides that certain characters are part of a terminal control sequence, it would appear as if the string had not been fully accepted—it would not appear on the display—when in reality it had. You can therefore use `WriteString` to send terminal control sequences to the terminal to alter its behavior. Note that the host computer would be unaware of this change in behavior.

The terminal will accept a string written with `WriteString` at any time, even when it is actively receiving data from the serial device. Be aware that under these circumstances, the characters written with `WriteString` will intermingle with data from the serial device and may cause some bizarre behavior and displays.

Note also that the lazy write mode still applies to text written to the terminal with `WriteString`. If `UseLazyDisplay` is `True`, the text will appear at the appropriate time. You can force the new data to be displayed in this case by calling the `Update` method.

Chapter 9: IP Telephony

Async Professional includes components to transmit data over phone lines and networks, and components to transmit wave recordings through voice modems, APRO also includes a component to transmit audio and video through the network through IP Telephony.

IP (Internet Protocol) Telephony is a technology where audio and video can be streamed across a network. As its name implies, IP Telephony uses IP as the transport layer. This transport layer is provided through TCP/IP. IP Telephony, in APRO, also uses TAPI 3.x, which is only available for Windows 2000 (future Windows versions should also support this implementation). TAPI 3.x, in turn, uses H.323 and the Microsoft H.323 TAPI Service Provider, which is installed by default.

IP Telephony includes the transmission of voice and video over the network. The network can be a local area network (LAN), wide area network (WAN) or the Internet. This technology offers the capability of establishing real-time conferencing around the globe using existing network topology. Using this technology you can hold a real-time conference (complete with audio and video), telecommuting, distance learning, and video on demand, to name a few of the possibilities.

IP Telephony in Async Professional

The TapdVoIP component implements IP Telephony (also known as “Voice over IP”) through the services of TAPI 3.x. TAPI 3.x is available for Windows 2000. Documentation on the Microsoft web site confirms that future Windows versions will support the TAPI 3.x architecture. The TapdVoIP component implements a small subset of TAPI 3.x, limited to the functionality required for Voice over IP.

Since the TapdVoIP component requires TAPI 3.x, and TAPI 3.x is available only on Windows 2000 and later, the TapdVoIP component requires Windows 2000 or later and the Microsoft H.323 TAPI Service Provider. An application that contains the TapdVoIP component can be instantiated on an unsupported Windows version, however all TapdVoIP methods will raise the EVoIPNotSupported exception. To prevent this exception, the VoIPAvailable property can be checked, this property will be True if the required TAPI components are available and False if TAPI does not support it.

H.323

TAPI 3.x implements IP Telephony through the H.323 protocol. This protocol is optimized for the transmission of voice and video over connectionless networks that do not provide guaranteed quality of service (such as IP-based networks and the Internet). H.323 is a comprehensive ITU (International Telecommunications Union) standard for multimedia communications. This protocol defines call control techniques, multimedia and bandwidth management, and mandates for standard audio and video codecs. The H.323 protocol is also platform independent, which means an H.323 instance on a Windows machine can communicate with an H.323 instance on another operating system.

Detailed discussions, and implementation specifications, are available on the ITU Web site. Knowledge of H.323 is not required to use the IP Telephony components in Async Professional. TAPI 3 and APRO will handle all of the details for you.

Configuration for VoIP

As previously stated, the Voice over IP functionality in the TApdVoIP component is available in TAPI 3.x. This TAPI version is available only for Windows 2000, although future Windows versions will most likely include TAPI 3.x compatibility.

The TApdVoIP component also automatically selects the H.323 TAPI device. If this device is not installed (it is installed by default in Windows 2000), the VoIPAvailable property is set to False when the component is created, and the TApdVoIP methods will raise the EVoIPNotSupported exception when called.

The audio and video streaming in VoIP are provided through H.323 media services (also installed by default in Windows 2000). These media services support selection of the audio input, audio output, video input and video output devices. When the TApdVoIP component is created, the media services are queried for supported media terminals. As the terminals are being queried, their capabilities are also queried. All of the supported media terminals are stored in the AvailableTerminalDevices property. This property is a TStrings instance, the Strings value is the name of the terminal; the Objects value is a TApdVoIPTerminal object. See the TApdVoIPTerminal description later in this chapter for details.

Audio and video device selection

There are four properties available in the TApdVoIP component to select which media terminal to use. The AudioInDevice property determines which device to use for audio input (usually a microphone). The AudioOutDevice property determines which device to use for audio output (usually external speakers or headset speakers). The VideoInDevice property determines which device to use for video input (usually a digital video camera or web cam installed on the system). The property values for these properties contain the name of the selected device. The final property for terminal selection is the VideoOutDevice property. This property specifies a TWinControl that is used to render the video stream received through the call. Any TWinControl descendent can be used, although a TPanel is probably the most applicable.

The default values for these terminal properties are "" (empty string) for the AudioInDevice, AudioOutDevice and VideoInDevice, and nil for the VideoOutDevice. The default values disable that media type on the call. For example, if VideoInDevice is an empty string, the call will not be configured to support video input.

The AudioInDevice, AudioOutDevice and VideoInDevice properties are strings, which correspond to the DeviceName property of the associated TApdVoIPTerminal object. These properties can be set directly by assigning the DeviceName of the terminal to the property.

These properties can also be set through the property editor. This property editor is also available at run time through the ShowMediaSelectDialog method. The property editor for these properties contains a combo box for each property as shown in Figure 9.1.

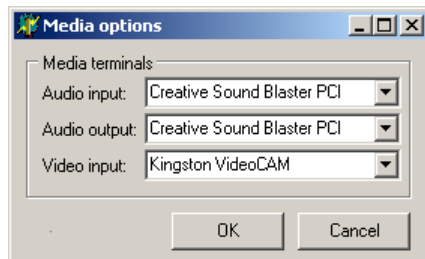


Figure 9.1: Media device property editor.

Each combo box contains the terminal types associated with the given property. The “Audio input” control contains only those devices that support audio input; the “Audio output” control contains only those devices that support audio output; the “Video input” control contains only those devices that support video input. When OK is clicked, the AudioInDevice, AudioOutDevice and VideoInDevice properties are updated to reflect the changes. If Cancel is clicked, the changes are disregarded.

Originating a VoIP call

When originating a call, the TAPdVoIP component initializes the underlying TAPI and H.323 layers, and then attempts to connect to a VoIP implementation elsewhere on the network. The Connect method of the TAPdVoIP component begins the connection attempt. The destination address is determined by the DestAddr string parameter of the Connect method. VoIP addresses can be dotted quad Internet addresses or machine names. For example, the following snippets will connect to a specified IP address and a specified machine:

```
ApdVoIP1.Connect('192.168.12.131');  
ApdVoIP1.Connect('john_work.turbopower.com');
```

An attempt to create the connection is made immediately. If the connection attempt succeeds, the OnConnect event is generated. If the connection attempt fails, the OnFail event is generated.

Receiving a VoIP call

When receiving a call, the TApdVoIP component initializes the underlying TAPI and H.323 layers and waits for the incoming call. The Connect method of the TApdVoIP component begins the answering process. When that method is called with a DestAddr of '' (empty string), the TAPI and H.323 layers are activated and configured to provide status notifications. When an incoming call is detected, the OnIncomingCall event is generated. If the Accept parameter of that event is set to True (the default), the call is answered and the OnConnect event is generated. The following example shows how to receive a VoIP call:

```
...
TApdVoIP1.Connect('');
...

procedure TForm1.TApdVoIP1IncomingCall(VoIP : TApdCustomVoIP;
CallerAddr: string; var Accept : Boolean);
begin
    Accept := MessageDlg('Accept incoming call from ' +
        CallerAddr, mtConfirmation, [mbYes, mbNo], 0) = mrOK;
end;
```

Terminating a VoIP call

Once a call is established through the Connect method of the TApdVoIP component it can be terminated locally or from the remote station. To terminate the call locally, call the CancelCall method of the TApdVoIP component. If an active call is present (the OnConnect event has been generated and the Connected property is True) the OnDisconnect event will be generated shortly after CancelCall returns. If an active call is not present (after calling Connect to enter the answer mode, but before the OnConnect event has been generated), the CancelCall method will not return until the answer mode has been terminated.

At any point in the call, the remote side could terminate, or the network could become disconnected. Regardless of the reason, the OnDisconnect event will be generated if the OnConnect event has been generated previously for this call. If the OnConnect event has not been generated for this call, the OnFail event will be generated and the Reason parameter of that event will contain the reason for the failure.

TApdVoIPTerminal Class

The TApdVoIPTerminal class defines a media terminal. This class is used in the TApdVoIP component's AvailableTerminalDevices property. The AvailableTerminalDevices property is a TStrings type. Each Strings value is the name of a media terminal, each Objects value is a TApdVoIPTerminal class that defines the capabilities of the given terminal.

Hierarchy

TStrings (VCL)
 TApdVoIPTerminal (AdVoIP)

Properties

DeviceClass	DeviceName	MediaType
DeviceInUse	MediaDirection	

Reference Section

DeviceClass

read-only, run-time property

```
property DeviceClass : TApdTerminalDeviceClass

TApdTerminalDeviceClass = (
    dcHandsetTerminal, dcHeadsetTerminal, dcMediaStreamTerminal,
    dcMicrophoneTerminal, dcSpeakerphoneTerminal, dcSpeakersTerminal,
    dcVideoInputTerminal, dcVideoWindowTerm)
```

↪ Indicates the terminal class of the media terminal.

TAPI 3.x defines several terminal classes. The following table shows which terminal classes are applicable to which TApdVoIP terminal properties:

TApdVoIP Property	DeviceClass
AudioInDevice	dcHandsetTerminal
	dcHeadsetTerminal
	dcMicrophoneTerminal
	dcSpeakerphoneTerminal
AudioOutDevice	dcHandsetTerminal
	dcHeadsetTerminal
	dcSpeakerphoneTerminal
	dcSpeakersTerminal
VideoInDevice	dcVideoInput

See also: TApdVoIP.AudioInDevice, TApdVoIP.AudioOutDevice, TApdVoIP.VideoInDevice

DeviceInUse

read-only, run-time property

```
property DeviceInUse : Boolean
```

↪ Indicates whether the terminal is currently being used.

This property is True if the terminal is being used by another process, and False if the terminal is available.

DeviceName**read-only, run-time property**

```
property DeviceName : string
```

↳ Indicates the name of the media terminal.

This property reflects the name of the media terminal as defined by the media installer. This property is used to define the Strings value associated with this Object when added to the AvailableTerminalDevices property.

MediaDirection**read-only, run-time property**

```
property MediaDirection : TApdMediaDirection
```

```
TApdMediaDirection = (mdCapture, mdRender, mdBidirectional);
```

↳ Indicates the directional capabilities of the media terminal.

A media terminal can either receive media data, transmit media data, or transmit and receive media data. The given media terminal's directional capabilities are indicated by this property according to the following:

MediaDirection	Description
mdCapture	Can receive (capture) media stream, corresponds to "In" devices.
mdRender	Can transmit (render) media stream, corresponds to "Out" devices.
mdBidirectional	Can receive (capture) and transmit (render) media streams.

MediaType**read-only, run-time property**

```
property MediaType : TApdTerminalMediaType
```

```
TApdTerminalMediaType = (mtStatic, mtDynamic);
```

↳ Indicates how the terminal is instantiated.

A media terminal can be predefined or created as needed. mtStatic terminals usually refer to hardware devices (speakers, microphones, etc.). mtDynamic terminals are dynamically created or assigned (video display).

TApdVoIP Component

The TApdVoIP component implements Voice over IP (IP Telephony) through the TAPI 3.x and H.323 interfaces. This component is compatible with Windows 2000, although later Windows versions will most likely support TAPI 3.x to some extent.

Hierarchy

TOleServer (VCL)	
❶ TApdBaseOleServer (OOMisc)	8
TApdCustomVoIP (AdVoIP)	
TApdVoIP (AdVoIP)	

Properties

AudioInDevice	Connected	VoIPAvailable
AudioOutDevice	❶ Version	WaitingForCall
AvailableTerminalDevices	VideoInDevice	
CallInfo	VideoOutDevice	

Methods

CancelCall	Connect	ShowMediaSelectDialog
------------	---------	-----------------------

Events

OnConnect	OnFail
OnDisconnect	OnIncomingCall

Reference Section

AudioInDevice

property

```
property AudioInDevice : string
```

↳ Determines the media terminal device to use for audio input.

This property is the DeviceName of the TAPdVoIPTerminal object that will be used to provide audio input. Audio input is usually provided through a microphone connected to the “mic” jack of a sound card.

This property supports the “Media options” property editor, which filters media terminals by type. See “Audio and video device selection” on page 295 for details.

See also: AudioOutDevice, AvailableTerminalClasses, VideoInDevice, VideoOutDevice

AudioOutDevice

property

```
property AudioOutDevice : string
```

↳ Determines the media terminal device to use for audio output.

This property is the DeviceName of the TAPdVoIPTerminal object that will be used to provide audio output. Audio output is usually provided through speakers connected to the “speaker” jack of a sound card, although it can also be a headset.

This property supports the “Media options” property editor, which filters media terminals by type. See “Audio and video device selection” on page 295 for details.

See also: AudioInDevice, AvailableTerminalClasses, VideoInDevice, VideoOutDevice

AvailableTerminalDevices

read-only, run-time property

```
property AvailableTerminalDevices : TStrings
```

↳ Contains references to the available media terminal devices.

When the TAPdVoIP component is created, all supported media terminals are enumerated and their capabilities are retrieved. This information is placed in the AvailableTerminalDevices property. The Strings value contains the DeviceName of the terminal, the Objects value contains the TAPdVoIPTerminal object.

See the TAPdVoIPTerminal description for details on the TAPdVoIPTerminal object. See “Audio and video device selection” on page 295 for details on how AvailableTerminalDevices can be used.

See also: AudioInDevice, AudioOutDevice, VideoInDevice, VideoOutDevice

```
property CallInfo : TApdVoIPCallInfo
```

↪ A structure containing details about the current call.

The TApdVoIP component contains properties for basic call information. TAPI maintains much more information about the call. Although this information may be of limited use, CallInfo is provided for projects that may require the extended information.

CallInfo is allocated immediately after the Connect method is called, is updated periodically by TAPI, and is valid for the duration of the call.

```
procedure CancelCall;
```

↪ Terminates the current call.

CancelCall is the universal method for terminating the current call. CancelCall can be used while waiting for incoming calls, originating a call, negotiating a call and during an established call. The TApdVoIP component terminates the current TAPI process, releases any resources allocated for the call, then returns to an idle state.

If an active call is present (the OnConnect event has been generated and the Connected property is True), the OnDisconnect event will be generated shortly after CancelCall returns. If an active call is not present (after calling Connect("") to enter the answer mode, but before the OnConnect event has been generated), the CancelCall method will not return until the answer mode has been terminated.

See also: Connect, OnConnect, OnDisconnect

```
procedure Connect(DestAddr : string);
```

↳ Establishes a VoIP connection.

The Connect method is used to establish a Voice over IP call. If DestAddr is an empty string, Connect waits for an incoming call to be received. If DestAddr is not an empty string, Connect originates a call to the destination specified by DestAddr.

When originating a call, DestAddr can specify either an IP address or a machine name. See “Originating a VoIP call” on page 296 for details. The underlying TAPI layer requires an address-type flag, which indicates whether the address is an IP address or a machine name. The TApdVoIP component will parse DestAddr to determine the type of address.

When receiving a call, DestAddr must be an empty string. See “Receiving a VoIP call” on page 297 for details. When receiving a VoIP call, it is not known when that call will be originated on the remote system. The OnIncomingCall event will be generated when the incoming call is detected. This event provides the address of the originating system, which can be used to accept or reject the call.

When originating or receiving a call, the OnConnect event is generated when the VoIP call is connected.

See also: CancelCall, Connected, OnConnect, OnIncomingCall, WaitingForCall

Connected

read-only, run-time property

```
property Connected
```

↳ Indicates whether a connection is currently active or not.

This property is set to True when a connection is established (when the OnConnect event is generated) and False when the connection is terminated (when the OnDisconnect event is generated). This property can be used to determine whether a call is in progress or not.

See also: Connect, OnConnect, OnDisconnect, WaitingForCalls

OnConnect

event

```
property OnConnect : TApdVoIPNotifyEvent
```

```
TApdVoIPNotifyEvent = procedure (VoIP : TApdCustomVoIP) of object;
```

↳ Defines an event handler that is generated when a VoIP connection is established.

The Connect method begins a background process that can take some time before actually establishing a connection. The OnConnect event is generated when a VoIP call is actually connected.

When a VoIP call is originated, the OnConnect event is generated after the called station accepts the call and the H.323 negotiations are complete. When Connect is called to answer calls, the OnConnect event is generated after the OnIncomingCall event is generated and the Accept parameter is set to True.

VoIP is the TApdCustomVoIP component that generated the event.

See also: CancelCall, Connect, Connected, OnDisconnect, OnIncomingCall

OnDisconnect event

```
property OnDisconnect : TApdVoIPNotifyEvent  
TApdVoIPNotifyEvent = procedure (VoIP : TApdCustomVoIP) of object;
```

↳ Defines an event handler that is generated when a VoIP connection is terminated.

This event handler is generated after calling the CancelCall method to terminate the connection from the local side and when the connection is terminated from the remote side or by the network. This event will only be generated if the call has been connected (the OnConnect event has been generated).

VoIP is the TApdCustomVoIP component that generated the event.

See also: CancelCall, Connect, OnConnect

OnFail event

```
property OnFail : TApdVoIPFailEvent  
TApdVoIPFailEvent = procedure(  
    VoIP : TApdCustomVoIP; ErrorCode : Integer) of object;
```

↳ Defines an event handler that is generated when a VoIP call fails.

The Connect method can fail for a variety of reasons, the remote could reject the call, the network could be congested, the destination address could be invalid, etc. This event handler is generated when the Connect method fails. This event is also generated when an established call fails.

VoIP is the TApdCustomVoIP component that generated the event. ErrorCode is a non-zero value indicating the type of failure.

See also: Connect, OnDisconnect

```
property OnIncomingCall : TApdVoIPIncomingCallEvent

TApdVoIPIncomingCallEvent = procedure(
    VoIP : TApdCustomVoIP; CallerAddr: string; var Accept : Boolean);
```

↳ Defines an event handler that is generated when an incoming call is detected.

After calling the `Connect` method with an empty string parameter, the `TApdVoIP` component begins a background process that monitors for incoming calls. When an incoming call is detected, the `OnIncomingCall` event is generated. This event provides an opportunity to accept or reject the call, either based on the `CallerAddr` or other criteria.

`VoIP` is the `TApdCustomVoIP` component that received the incoming call notification and generated the event. `CallerAddr` is a form of Caller ID specific to IP Telephony, this string is usually an IP address or machine name that identifies the system that originated the call. `Accept` determines whether the `TApdVoIP` component accepts or rejects the incoming call. If `Accept` is `True` when this event exits, the call is accepted and the `OnConnect` event is generated when the H.323 negotiations are complete. If `Accept` is `False` when this event exits, the call is rejected and the `TApdVoIP` component will continue monitoring for new incoming calls. The `OnFail` and `OnDisconnect` events will not be generated if the call is not accepted.

See “Receiving a VoIP call” on page 297 for details and an example.

See also: `Connect`, `OnConnect`, `OnDisconnect`

ShowMediaSelectDialog**method**

```
function ShowMediaSelectDialog : Boolean;
```

↳ Displays a dialog where the audio and video devices can be selected.

The `ShowMediaSelectDialog` method displays the Media options dialog discussed in the “Audio and video device selection” on page 295. This dialog box allows the user to select the media terminals for the `AudioInDevice`, `AudioOutDevice`, and `VideoInDevice` properties.

If `OK` is clicked, the `AudioInDevice`, `AudioOutDevice` and `VideoInDevice` properties are updated to reflect the new media terminals and the return value of `ShowMediaSelectDialog` is `True`. If `Cancel` is clicked, the properties are not changed and this method returns `False`.

The dialog displayed from the `ShowMediaSelectDialog` method is the same dialog used for the `AudioInDevice`, `AudioOutDevice` and `VideoInDevice` property editors.

See also: `AudioInDevice`, `AudioOutDevice`, `VideoInDevice`

VideoInDevice

property

```
property VideoInDevice : string
```

↪ Determines the media terminal device to use to for video input.

This property is the DeviceName of the TApdVoIPTerminal object that will be used to provide video input. Video input is usually provided through a digital camera or web camera installed on the system.

This property supports the Media options property editor, which filters media terminals by type. See “Audio and video device selection” on page 295 for details.

See also: AudioInDevice, AudioOutDevice, AvailableTerminalClasses, VideoOutDevice

VideoOutDevice

property

```
property VideoOutDevice : TWinControl
```

↪ Determines the control where video output is rendered.

This property is a TWinControl descendent where the received video stream is displayed. To display the received video stream, assign the name of a TWinControl descendent, such as a TPanel, to the VideoOutDevice property.

See also: AudioInDevice, AudioOutDevice, AvailableTerminalClasses, VideoInDevice

VoIPAvailable

read-only, run-time property

```
property VoIPAvailable : Boolean
```

↪ Indicates whether Voice over IP is supported or not.

The TApdVoIP component implements Voice over IP (IP Telephony) using TAPI 3.x and the H.323 TAPI service provider. TAPI 3.x is available for Windows 2000 and later, it is not available for earlier versions of Windows. When the TApdVoIP component is created, the TAPI version and H.323 TSP are verified, if these TAPI components are not available the VoIPAvailable property is set to False; if these TAPI components are available the VoIPAvailable property is set to True.

If VoIPAvailable is False, calling the Connect method will raise the EVoIPNotSupported exception.

See also: Connect

property `WaitingForCall` : Boolean

↳ Indicates whether the `TApdVoIP` component is waiting for incoming calls or not.

When the `Connect` method is called with an empty string, the `TApdVoIP` component will begin monitoring for incoming calls. `WaitingForCall` indicates whether the `TApdVoIP` component is waiting for incoming calls or not.

See also: `Connect`, `Connected`

Chapter 10: SAPI Components

The SAPI (Speech API) components in Async Professional provide an easy means to incorporate speech synthesis and recognition into your programs. The components act as an interface between the Microsoft SAPI 4 and Delphi and C++ Builder.

SAPI Overview

Async Professional provides speech synthesis, speech recognition and voice telephony capabilities. These aspects will be explored in further in the following sections.

Speech synthesis

Speech synthesis will take plain ASCII text and convert it to into digital audio. There are three primary ways this is done as described in Table 10.1.

Table 10.1: *Speech synthesis schemes*

Scheme	Description
Concatenated Word	This technique stitches together recordings of individual words as provided by the developer of the speech synthesis engine. This method can provide high quality output, but is limited to the words that have been provided with the engine.
Subword Concatenation	This speech synthesis engine concatenates short prerecorded phonemes to spell out the words. The small pieces of audio are smoothed out to improve the quality of the spoken text.
Synthesis	The speech synthesis engine will simulate the human vocal chords when it generates the digital audio. This technique has the advantage of the ability to adjust the sound of the voice via a few simple parameters.

In Async Professional, generic speech synthesis is provided through the `TAPdSapiEngine` component. The generated speech can either go directly to the sound card or be saved as a .WAV file for future usage.

While great strides have been made in the quality of computer-synthesized speech, it is still easily recognizable as computer generated. As of yet, a computer cannot duplicate the inflection, timing and emotion of human speech.

The speech synthesis engine will do it's best to pronounce the text given to it. However, some words and names it will have problems with. The best way to handle this is to provide the proper phonemes to the speech synthesis engine.

Some speech synthesis engines support input using the International Phonetic Alphabet (IPA). This is a standardized system of representing phonemes in a Unicode character set. If your engine supports IPA, this can be used to precisely control the pronunciation of a word.

You can verify if an engine supports IPA by checking if `tfIPAUnicode` is in the features set of the engine. This can be found in the `Features` subproperty of the `SSVoices` property of the `TApdSapiEngine` component.

Speech recognition

Speech recognition converts a spoken audio data stream into ASCII text. In Async Professional, generic speech recognition is provided through the `TApdSapiEngine` component.

Speech recognition requires an object known as a grammar (a list of known words and possibly how they relate to each other). In Async Professional, the grammar is handled automatically. Either a list of expected words can be provided in the `WordList` property, or the `Dictation` property can be set to `True`. In the latter case, the speech synthesis engine will be provided with a dictation grammar. Most speech recognition engines will then use a much larger list of known words. Additional words can be specified via the `WordList` property.

Using a specific word list is generally faster and more accurate than using the full dictation grammar.

As words and phrases are recognized the `OnPhraseHypothesis` and `OnPhraseFinish` events will fire to let the application know what words were spoken.

The Microsoft SAPI SDK documentation provides a full list of supported phonemes.

Voice telephony

Voice telephony support is provided via the `TApdSapiPhone` component. This component allows for speech synthesis and recognition to take place over a voice call. The component is a descendent of the `TApdCustomTAPIDevice` and provides all the functionality of a TAPI based voice call with the capabilities of the `TApdCustomSapiEngine`.

The voice telephony component has several methods for asking the user for information and then returning that information in a usable format. For example, the `AskForDate` method will prompt the user for a date and then return that date as a `TDateTime` parameter. Other methods exist for asking for phone numbers, times, items from a list, extensions and yes or no replies.

Requirements for speech synthesis and recognition

For speech synthesis and recognition, you will need to have the following:

- A 486/33 (DX or SX) or better CPU is the bare minimum required. A Pentium is recommended.
- Speech synthesis will require an additional 1Mb of RAM to that which your program already needs. Speech recognition will require an additional 8Mb.
- A sound card is needed for both speech synthesis and recognition. For speech recognition, a microphone is required. Headset microphones are less susceptible to background noise and are preferable to desktop microphones.
- Microsoft Windows 95 or NT 4.0 or better.
- The Microsoft SAPI controls and a speech recognition and synthesis engine is needed. A good place to search both for the SAPI software and speech engines is <http://www.microsoft.com/speech/>.
- For voice telephony, you will need TAPI, a voice capable modem and speech synthesis and recognition engines that have been optimized for telephony applications.

Considerations of speech synthesis and recognition

For speech synthesis and recognition to work properly, there are several things that should be taken into consideration.

For speech recognition, the most significant source of problems will be from the microphone. The quality of the microphone and the training that the user has done with the microphone directly affect the ability of the speech recognition engine to interpret what was said to it. Headset microphones are generally preferable to desktop models.

Half-duplex sound cards should be avoided. A sound card is “half-duplex” if it is incapable of playing and recording audio at the same time. While the card is playing, speech recognition cannot occur.

Feedback between sound card’s output and the microphone can dramatically reduce the quality of the spoken text.

Background noise will also affect the speech recognition. Background conversations, a radio, door slams or other ambient noise can be picked up by the speech recognition engine and interpreted in unexpected ways. The use of a higher quality microphone can reduce the problems, but will not completely eliminate them.

Speech recognition is not perfect. Words will be misinterpreted, or not interpreted at all., and background noise will be interpreted as words.

Considerations for voice enabling your application

There are numerous things that should be taken into account when developing voice-enabled applications. Here are a few items to keep in mind:

- Speech recognition does make mistakes. If you are using speech recognition to handle commands to your application, you will want confirmation before doing anything that may be harmful.
- Speech synthesis should be used for short phrases. Computer generated voices can quickly annoy your users.
- Speech recognition control of an application as well as speech synthesis should be optional. Speech recognition should not be used to replace the standard means of input (that is, the keyboard and mouse).
- In voice telephony, other options should be provided for when speech recognition fails. DTMF is usually a good backup option.
- Mixing prerecorded text with synthesized speech generally sounds bad. As a rule, either prerecord everything or nothing.
- Visual feedback should be provided for both speech recognition and synthesis.

Distribution of speech synthesis and speech recognition engines

There are two options for distributing voice-enabled applications. The first is to bundle the needed speech engines with the application. This will require negotiation with the speech engine vendor and for the appropriate royalties to be paid.

The second option is to require the user to provide the speech engine. Many sound cards come bundled with speech engines. The quality of these engines varies dramatically.

Async Professional does not come bundled with speech engines.

TapdAudioOutDevice Class

The TapdAudioOutDevice class is used by the TapdSapiEngine component to encapsulate the engines used for speech recognition. This class allows for the easy selection of a speech recognition engine as well as the ability to look at the details of the engines.

The properties of this class are laid out as arrays in which each speech recognition engine is accessed by its index in the array. The CurrentEngine property is used to set the speech recognition engine. Setting this property to the index of a speech recognition engine will activate that engine.

Hierarchy

TObject (VCL)
 TapdAudioOutDevice (AdSapiEn)

Properties

Age	Features	ModeName
Count	Gender	ProductName
CurrentVoice	Interfaces	Speaker
Dialect	LanguageID	Style
EngineFeatures	MfgName	
EngineID	ModeID	

Methods

Find

Reference Section

Age read-only, run-time property

```
property Age[x : Integer] : TApdTTSAge
TApdTTSAge = (tsBaby, tsToddler, tsChild,
    tsAdolescent, tsAdult, tsElderly, tsUnknown);
```

↳ Indicates the age of the specified voice.

The Age property indicates the age of the voice currently specified . The following table indicates values for Age:

Value	Approximate Age
tsBaby	1
tsToddler	3
tsChild	6
tsAdolescent	14
tsAdult	20 - 60
tsElderly	Over 60
tsUnknown	Unknown

The index of the array indicates the voice that the property refers to. To get the Age property for the currently active voice, use the CurrentVoice property as the index.

See also: CurrentVoice

Count read-only, run-time property

```
property Count : Integer
```

↳ Specifies the number of installed speech synthesis voices.

This property indicates the number of speech synthesis voices that are installed. The voices are numbered from 0 to Count – 1. The properties of the voices are accessed as an array where the index indicates the voice in question.

The active voice can be activated by setting the CurrentVoice property.

See also: CurrentVoice

property CurrentVoice

↳ Specifies the current speech synthesis voice.

This property is used to activate a specific voice for use in speech synthesis. This value should be set between 0 and Count – 1. This property should not be modified while text is being spoken.

See also: Count, Speak

Dialect**read-only, run-time property**

property Dialect[x : Integer] : string

↳ Indicates the dialect of the specified voice.

Examples of dialects are “Texas” or “New York City.” The actual dialects vary from engine to engine.

If no specific dialect is used, this property may be either “Standard” or blank.

The index of the array indicates the voice that the property refers to. To get the Dialect property for the currently active voice, use the CurrentVoice property as the index.

See also: CurrentVoice

EngineFeatures**read-only, run-time property**

property EngineFeatures[x : Integer] : Integer

↳ Indicates speech synthesis engine specific features for the specified voice.

This property specifies features specific to the speech synthesis engine. The meaning of the value of this property will depend on the speech synthesis engine.

The index of the array indicates the voice that the property refers to. To get the EngineFeatures property for the currently active voice, use the CurrentVoice property as the index.

See also: CurrentVoice

```
property EngineID[x : Integer] : string
```

↳ Identifies the GUID of the speech synthesis engine.

This property identifies the GUID for the speech synthesis engine. This GUID may be used for multiple voices and languages.

The index of the array indicates the voice that the property refers to. To get the EngineID property for the currently active voice, use the CurrentVoice property as the index.

See also: CurrentVoice

Features**read-only, run-time property**

```
property Features[x : Integer] : TApdTTSFeatures
```

```
TApdTTSFeatures = set of (tfAnyWord, tfVolume, tfSpeed, tfPitch,  
    tfTagged, tfIPAUnicode, tfVisual, tfWordPosition, tfPCOptimized,  
    tfPhoneOptimized, tfFixedAudio, tfSingleInstance, tfThreadSafe,  
    tfIPATextData, tfPreferred, tfTransplanted, tfSAPI4);
```

↳ Identifies what features the speech synthesis engine supports.

The most common speech synthesis engine features are listed in the following table:

Feature	Description
tfAnyWord	The speech synthesis engine will attempt to read any word.
tfFixedAudio	The audio device that the speech synthesis engine will speak to is fixed and cannot be changed.
tfIPATextData	The speech synthesis engine supports the IPA
tfIPAUnicode	The speech synthesis engine supports the International Phonetic Alphabet Unicode character set.
tfPCOptimized	The speech synthesis engine's voice is optimized to output to the computer.
tfPhoneOptimized	The speech synthesis engine's voice is optimized for the telephone. This is required for voice telephony.
tfPitch	The speech synthesis engine can adjust the pitch while it is speaking.
tfSAPI4	The engine is designed for SAPI 4.
tfSingleInstance	Only one instance of the speech synthesis engine can exist at a time.

Feature	Description
tfSpeed	The speech synthesis engine can adjust the speaking rate while it is speaking.
tfTagged	The speech synthesis engine can recognize and handle embedded control tags in the text.
tfThreadSafe	The speech synthesis engine is thread safe.
tfTransplanted	The speech synthesis engine supports transplanted prosody.
tfVisual	The speech synthesis engine provides mouth position information while it is speaking.
tfVolume	The speech synthesis engine can adjust the speaking volume while it is speaking.
tfWordPosition	The speech synthesis engine can send notification of word position while it is speaking.

The index of the array indicates the voice that the property refers to. To get the Features property for the currently active voice, use the CurrentVoice property as the index.

See also: CurrentVoice

Find method

```
function Find(Criteria : string) : Integer;
```

↳ Finds the speech synthesis engine and voice that is the closest match for the input criteria.

This method is used to find the best matched speech engine and voice for the requested features. The format of the Criteria parameter is:

```
<field>=<value>;<field=value>
```

Legal values for field and value are:

Field	Value
Age	Can be one of the following constants: ApdTTSAGE_BABY, ApdTTSAGE_TODDLER, ApdTTSAGE_CHILD, ApdTTSAGE_ADOLESCENT, ApdTTSAGE_ADULT, ApdTTSAGE_ELDERLY. Note: These constants are integers and will have to be converted to string values for usages with the Find method.
Dialect	String
EngineFeatures	Integer
EngineId	String

Field	Value
Features	Can be one of the following constants: ApdTTSFEATURE_ANYWORD, ApdTTSFEATURE_VOLUME, ApdTTSFEATURE_SPEED, ApdTTSFEATURE_PITCH, ApdTTSFEATURE_TAGGED, ApdTTSFEATURE_IPAUNICODE, ApdTTSFEATURE_VISUAL, ApdTTSFEATURE_WORDPOSITION, ApdTTSFEATURE_PCOPTIMIZED, ApdTTSFEATURE_PHONEOPTIMIZED, ApdTTSFEATURE_FIXEDAUDIO, ApdTTSFEATURE_SINGLEINSTANCE, ApdTTSFEATURE_THREADSAFE, ApdTTSFEATURE_IPATEXTDATA, ApdTTSFEATURE_PREFERRED, ApdTTSFEATURE_TRANSPLANTED, ApdTTSFEATURE_SAPI4. Note: These constants are integers and will have to be converted to string values for use with the Find method.
Gender	Can be one of the following constants: ApdGENDER_NEUTRAL, ApdGENDER_FEMALE, ApdGENDER_MALE. Note: These constants are integers and will have to be converted to string values for use with the Find method.
Interfaces	Can be one of the following constants: ApdTTSI_ILEXPRONOUNCE, ApdTTSI_ITTSATTRIBUTES, ApdTTSI_ITTSCENTRAL, ApdTTSI_ITTSDIALOGS, ApdTTSI_ATTRIBUTES, ApdTTSI_IATTRIBUTES, ApdTTSI_ILEXPRONOUNCE2. Note: These constants are integers and will have to be converted to string values for use with the Find method.
LanguageID	Integer
MfgName	String
ModeID	String
ModeName	String
ProductName	String
Speaker	String
Style	String

The following code example shows how to use Find:

```
var
    EngineIdx : Integer;

begin
    EngineIdx := ApdSapiEngine1.SSVoices.Find(
        'Gender=' + IntToStr (ApdGENDER_FEMALE));
    EngineIdx := ApdSapiEngine1.SSVoices.Find(
        'Gender=' + IntToStr(ApdGENDER_FEMALE) + ';Dialect=Texas');
```

See also: Count, CurrentEngine

Gender **read-only, run-time property**

```
property Gender[x : Integer] : TApdTTSGender
TApdTTSGender = (tgNeutral, tgFemale, tgMale, tgUnknown);
```

↪ Indicates the gender of the specified voice.

The index of the array indicates the voice that the property refers to. To get the Features property for the currently active voice, use the CurrentVoice property as the index.

See also: CurrentVoice

Interfaces **read-only, run-time property**

```
property Interfaces[x : Integer] : TApdTTSInterfaces
TApdTTSInterfaces = set of (
    tiLexPronounce, tiTTSSAttributes, tiTTSCentral, tiTTSDialogs,
    tiAttributes, tiIAttributes, tiLexPronounce2);
```

↪ Indicates the COM interfaces supported by the speech synthesis engine.

The index of the array indicates the speech synthesis voice that the property refers to. To get the Interfaces property for the currently active speech synthesis voice, use the CurrentVoice property as the index.

See also: CurrentVoice

LanguageID

read-only, run-time property

```
property LanguageID[x : Integer] : Integer
```

↳ Specifies the language identifier for the specified voice.

Bits 0 through 9 indicate the primary language. Bits 10-15 indicate a sublanguage (or locale).

More information on this can be found in the Windows SDK help under the MAKELANGID topic.

The index of the array indicates the speech synthesis voice that the property refers to. To get the LanguageID property for the currently active speech synthesis voice, use the CurrentVoice property as the index.

See also: CurrentVoice

MfgName

read-only, run-time property

```
property MfgName[x : Integer] : string
```

↳ Indicates the name of the manufacturer of the specified voice.

The index of the array indicates the speech synthesis voice that the property refers to. To get the MfgName property for the currently active speech synthesis voice, use the CurrentVoice property as the index.

See also: CurrentVoice

ModeID

read-only, run-time property

```
property ModeID[x : Integer] : string
```

↳ Specifies the GUID for the specified voice.

This property indicates the GUID that uniquely identifies each voice installed on the computer.

The index of the array indicates the speech synthesis voice that the property refers to. To get the ModeID property for the currently active speech synthesis voice, use the CurrentVoice property as the index.

See also: CurrentVoice

ModeName**read-only, run-time property**

```
property ModeName[x : Integer] : string
```

↳ Specifies the name of the specified text to speech mode.

This is the default property of the `TApdAudioOutDevice` class. Specifying an array index without any further field specifications will return this property.

The index of the array indicates the speech synthesis mode that the property refers to. To get the `ModeName` property for the currently active speech synthesis mode, use the `CurrentVoice` property as the index.

See also: `CurrentVoice`

ProductName**read-only, run-time property**

```
property ProductName[x : Integer] : string
```

↳ Provides the product name of the specified voice.

The index of the array indicates the speech synthesis voice that the property refers to. To get the `ProductName` property for the currently active speech synthesis voice, use the `CurrentVoice` property as the index.

See also: `CurrentVoice`

Speaker**read-only, run-time property**

```
property Speaker[x : Integer] : string
```

↳ Indicates the name of the specified selected speech synthesis voice.

This property specifies the name of the voice of the specified speech synthesis voice. This property may be blank for some voices.

The index of the array indicates the speech synthesis voice that the property refers to. To get the `Speaker` property for the currently active speech synthesis voice, use the `CurrentVoice` property as the index.

See also: `CurrentVoice`

```
property Style[x : Integer] : string
```

↪ Indicates the speaking style of the specified voice.

Common speaking styles include “Angry”, “Business”, “Calm”, “Casual”, “Computer”, “Depressed”, “Excited”, “Falsetto”, “Happy”, “Loud”, “Monotone”, “Perky”, “Quiet”, “Sarcastic”, “Scared”, “Shout”, “Singsong”, “Tense” and “Whisper.”

The index of the array indicates the speech synthesis voice that the property refers to. To get the Style property for the currently active speech synthesis voice, use the CurrentVoice property as the index.

See also: CurrentVoice

TapdAudioInDevice Class

The TapdAudioInDevice class is used by the TapdSapiEngine component to encapsulate the engines, modes, and voices used by speech synthesis. This class allows for the easy selection of a speech synthesis engine as well as the ability to look at the details of the speech synthesis engine.

The properties of this are laid out as arrays in which each speech synthesis engine is accessed by its index in the array. The CurrentVoice property is used to set the speech synthesis engine. Setting this property to the index of a speech synthesis engine will activate that engine.

Hierarchy

TObject (VCL)
 TapdAudioInDevice (AdSapiEn)

Properties

Count	Grammars	ModeID
CurrentEngine	Interfaces	ModeName
Dialect	LanguageID	ProductName
EngineFeatures	MaxWordsState	Sequencing
EngineID	MaxWordsVocab	
Features	MfgName	

Reference Section

Count

read-only, run-time property

```
property Count : Integer
```

↳ Specifies the number of installed speech recognition engines.

The engines are numbered from 0 to Count – 1. The properties of the speech recognition engines are accessed as an array where the index indicates the engine in question.

The active speech recognition engine can be activated by setting the CurrentEngine property.

See also: CurrentEngine

CurrentEngine

run-time property

```
property CurrentEngine : Integer
```

↳ Specifies the current speech recognition engine.

This property is used to activate a specific speech recognition engine for use in speech recognition. This value should be set between 0 and Count – 1.

See also: Count, Listen

Dialect

read-only, run-time property

```
property Dialect[x : Integer] : string
```

↳ Indicates the dialect of the specified language used by the speech recognition engine.

Examples of dialects are “Texas” or “New York City.” The allowed dialects vary from engine to engine.

If no specific dialect is used, this property may be either “Standard” or blank.

The index of the array indicates the speech recognition engine that the property refers to. To get the Dialect property for the currently active speech recognition engine, use the CurrentEngine property as the index.

See also: CurrentEngine

EngineFeatures

read-only, run-time property

```
property EngineFeatures[x : Integer] : Integer
```

↳ Indicates speech recognition engine specific features for the specified engine.

The meaning of the value of this property will depend on the speech recognition engine.

The index of the array indicates the speech recognition engine that the property refers to. To get the EngineFeatures property for the currently active speech recognition engine, use the CurrentEngine property as the index.

See also: CurrentEngine

EngineID

read-only, run-time property

```
property EngineID[x : Integer] : string
```

↳ Identifies the GUID of the speech synthesis engine.

This property identifies the GUID for the speech recognition engine. This GUID may be used for multiple voices and languages.

The index of the array indicates the speech recognition engine to which the property refers. To get the EngineID property for the currently active speech recognition engine, use the CurrentEngine property as the index.

See also: CurrentEngine

10

Features

read-only, run-time property

```
property Features[x : Integer] : TApdSRFeatures
```

```
TApdSRFeatures = set of (sfIndepSpeaker, sfIndepMicrophone,  
    sfTrainWord, sfTrainPhonetic, sfWildcard, sfAnyWord,  
    sfPCOptimized, sfPhoneOptimized, sfGramList, sfGramLink,  
    sfMultiLingual, sfGramRecursive, sfIPAUnicode, sfSingleInstance,  
    sfThreadSafe, sfFixedAudio, sfIPAWord, sfSAPI4);
```

↳ Identifies what features the speech recognition engine supports.

The following table details the meanings of the more common engine features:

Feature	Description
<code>sfAnyWord</code>	The speech recognition engine will attempt to recognize any word.
<code>sfFixedAudio</code>	The audio input device is fixed and cannot be changed.
<code>sfGramLink</code>	The speech recognition engine supports automatic linking between grammars.
<code>sfGramList</code>	The grammars used by the speech recognition engine support recognition lists.
<code>sfGramRecursive</code>	Context-free grammars support recursive rules.
<code>sfIndepMicrophone</code>	The speech recognition engine is microphone independent. Retraining is not required if the microphone changes.
<code>sfIndepSpeaker</code>	The speech recognition engine is speaker independent. It will work well without training.
<code>sfIPAUnicode</code>	The engine supports the Unicode International Phonetic Alphabet.
<code>sfMultiLingual</code>	The speech recognition engine is capable of recognizing multiple languages at a time.
<code>sfPCOptimized</code>	The speech recognition engine is optimized for use on a computer.
<code>sfPhoneOptimized</code>	The speech recognition engine is optimized for use over a phone. This is required for voice telephony.
<code>sfSAPI4</code>	The engine is designed for SAPI 4.
<code>sfSingleInstance</code>	Only a single instance of the voice recognition engine can be in memory at a time.
<code>sfThreadSafe</code>	The speech recognition engine is thread safe.
<code>sfTrainPhonetic</code>	The user can train the speech recognition engine on a known set of words specifically to train all the phonemes.
<code>sfTrainWord</code>	The speaker can train the speech recognition engine on individual words.
<code>sfWildcard</code>	Wildcards are supported in context-free grammars.

The index of the array indicates the speech recognition engine that the property refers to. To get the Features property for the currently active speech recognition engine, use the CurrentEngine property as the index.

See also: CurrentEngine

Grammars

read-only, run-time property

```
property Grammars[x : Integer] : TApdSRGrammars
TApdSRGrammars = set of (sgCFG, sgDictation, sgLimitedDomain);
```

↳ Indicates the types of grammars supported by the speech recognition engine.

They types of grammars supported by the speech recognition engine are listed in the following table:

Grammar	Description
sgCFG	Context-free grammar
sgDictation	Dictation grammar
sgLimitedDomain	Limited-domain grammar

A speech recognition engine may support multiple grammar types.

The index of the array indicates the speech recognition engine that the property refers to. To get the Grammars property for the currently active speech recognition engine, use the CurrentEngine property as the index.

See also: CurrentEngine

Interfaces

read-only, run-time property

```
property Interfaces[x : Integer] : TapdSRInterfaces

TapdSRInterfaces = set of (siLexPronounce, siSRAttributes,
    siSRCentral, siSRGramCommon, siSRDialogs, siSRGramCFG,
    siSRGramDictation, siSRGramInsertionGui, siSREsBasic,
    siSREsMerge, siSREsAudio, siSREsCorrection, siSREsEval,
    siSREsGraph, siSREsMemory, siSREsModifyGui,
    siSREsSpeaker, siSRSpeaker, siSREsScores, siSREsAudioEx,
    siSRGramLexPron, siSREsGraphEx, siLexPronounce2,
    siAttributes, siSRSpeaker2, siSRDialogs2);
```

↳ Indicates the COM interfaces supported by the speech recognition engine.

The index of the array indicates the speech recognition engine that the property refers to. To get the Interfaces property for the currently active speech recognition engine, use the CurrentEngine property as the index.

See also: CurrentEngine

LanguageID

read-only, run-time property

```
property LanguageID[x : Integer] : Integer
```

↳ Specifies the language identifier for the specified speech recognition engine.

Bits 0 through 9 indicate the primary language. Bits 10-15 indicate a sublanguage (or locale).

More information on this can be found in the Windows SDK help under the MAKELANGID topic.

The index of the array indicates the speech recognition engine that the property refers to. To get the LanguageID property for the currently active speech recognition engine, use the CurrentEngine property as the index.

See also: CurrentEngine

MaxWordsState

read-only, run-time property

```
property MaxWordsState[x : Integer] : Integer
```

↳ Indicates the maximum number of words in any grammar state for the specified engine.

The index of the array indicates the speech recognition engine that the property refers to. To get the MaxWordsState property for the currently active speech recognition engine, use the CurrentEngine property as the index.

See also: CurrentEngine

MaxWordsVocab

read-only, run-time property

```
property MaxWordsVocab[x : Integer] : Integer
```

- ↳ Indicates the maximum number of words in a grammar for the specified speech recognition engine.

The index of the array indicates the speech recognition engine that the property refers to. To get the MaxWordsVocab property for the currently active speech recognition engine, use the CurrentEngine property as the index.

See also: CurrentEngine

MfgName

read-only, run-time property

```
property MfgName[x : Integer] : string
```

- ↳ Indicates the name of the manufacturer of the specified speech recognition engine.

The index of the array indicates the speech recognition engine that the property refers to. To get the MfgName property for the currently active speech recognition engine, use the CurrentEngine property as the index.

See also: CurrentEngine

ModeID

read-only, run-time property

```
property ModeID[x : Integer] : string
```

- ↳ Specifies the GUID for the specified speech recognition engine.

This property indicates the GUID that uniquely identifies each speech recognition engine installed on the computer.

The index of the array indicates the speech recognition engine that the property refers to. To get the ModeID property for the currently active speech recognition engine, use the CurrentEngine property as the index.

See also: CurrentEngine

ModeName

read-only, run-time property

```
property ModeName[x : Integer] : string
```

↳ Specifies the name of the specified speech recognition mode.

This is the default property of the `TApdAudioInDevice` class. Specifying an array index without any further field specifications will return this property.

The index of the array indicates the speech recognition engine that the property refers to. To get the `ModeName` property for the currently active speech recognition engine, use the `CurrentEngine` property as the index.

See also: `CurrentEngine`

ProductName

read-only, run-time property

```
property ProductName[x : Integer] : string
```

↳ Provides the product name of the specified speech recognition engine.

The index of the array indicates the speech recognition engine that the property refers to. To get the `ProductName` property for the currently active speech recognition engine, use the `CurrentEngine` property as the index.

See also: `CurrentEngine`

Sequencing

read-only, run-time property

```
property Sequencing[x : Integer] : TApdSRSequences
```

```
TApdSRSequences = (ssDiscrete, ssContinuous, ssWordSpot,  
    ssContCFGDiscDict, ssUnknown);
```

↳ Indicates the speech recognition scheme.

The supported speech recognition schemes are detailed in the following table:

Scheme	Meaning
ssContCFGDiscDict	Performs continuous recognition for context-free grammars. In some cases, this may behave like the discrete speech recognition scheme.
ssContinuous	Continuous recognition. This will recognize words in a continuous stream of audio with no pauses.
ssDiscrete	Each word must be separated by a pause for the speech recognition engine to recognize it.
ssWordSpot	Searches for known words in a continuous stream of audio input.

The index of the array indicates the speech recognition engine that the property refers to. To get the Sequencing property for the currently active speech recognition engine, use the `CurrentEngine` property as the index.

See also: `CurrentEngine`

TApdCustomSapiEngine Component

The TApdCustomSapiEngine component provides access to the speech synthesis and recognition engines. The details of the installed engines are accessible via the SREngines and SSVoices properties.

The speech recognition and synthesis engines are automatically initialized at the time of creation.

Speech Synthesis Tags

Some speech synthesis engines support tags that allow for the changing of characteristics of the spoken text while it is being spoken.

All tags begin and end with a backslash. Tags are case insensitive, but white space is significant. To include a backslash in the text, use a double backslash.

An example of tagged text would be:

```
\Chr="Business"\This is a \Chr="monotone"\test.
```

The text "This is a" would be spoken using the Business character and the word test would be spoken in a monotone.

The tags for the speech synthesis engine are defined in Table 10.2.

Table 10.2: *Speech synthesis tags*

Tag	Meaning
Chr	Specifies a character for the voice. The speech synthesis engine defines what values are allowed for the Chr tag. Common values are "Angry", "Business", "Calm", "Depressed", "Excited", "Falsetto", "Happy", "Loud", "Monotone", "Perky", "Quiet", "Sarcastic", "Scared", "Shout", "Singsong", "Tense", and "Whisper". The actual values (if any) supported by the engine will vary from engine to engine.
Com	Comment. Comments are not spoken. An example would be \com="comment\".
Ctx	Specifies a context for the following text. The format of the tag is \Ctx="value\". Legal contexts are "Address" for addresses or phone numbers, "C" for C or C++ code, "Document" for text documents, "E-Mail" for email, "Numbers" for numbers, dates and times, "Spreadsheet" for spreadsheet documentation, "Normal", for normal text, and finally "Unknown" for unknown contexts.
Dem	Demphasizes the next word.
Emp	Emphasizes the next word.

Table 10.2: *Speech synthesis tags (continued)*

Tag	Meaning
Eng	Embeds an engine-specific command. The format is <code>\Eng;GUID;command\</code> or <code>\Eng;command\</code> where GUID is the GUID of the engine and the command is the command you want to send.
Mrk	Inserts a bookmark in the text. The format of the tag is <code>\Mrk=number\</code> where number is the number of the bookmark.
Pau	Pause for a number of milliseconds. The format of the tag is <code>\Pau=number\</code> where number is the number of milliseconds to pause.
Pit	Sets the baseline pitch in hertz. The format of the tag is <code>\Pit=number\</code> where number is the pitch you want to use.
Pra	Sets the pitch range. Format of the tag is <code>\Pra=value\</code> where value is the pitch range in hertz.
Prn	Specifies how to pronounce text by passing the phonetic version to the engine. The format of the tag is <code>\Prn=text=phonemes\</code> .
Pro	Turns on and off prosodic rules. <code>\Pro=1\</code> activates prosodic rules, <code>\Pro=0\</code> deactivates them.
Prt	Indicates the part of speech of the next word. The format of the tag is <code>\Prt="type"\</code> where type can be "Abbr" for abbreviations, "Adj" for adjectives, "Adv" for adverbs, "Card" for cardinal numbers, "Conj" for conjunctions, "Cont" for contractions, "Det" for determiners, "Interj" for interjections, "N" for nouns, "Ord" for ordinal numbers, "Prep" for prepositions, "Pron" for pronouns, "Prop" for proper nouns, "Punct" for punctuation, "Quant" for quantifiers, and "V" for verbs.
RmS	Turns on and off spelling mode. Use <code>\RmS=1\</code> to turn on spelling mode and <code>\RmS=0\</code> to turn it off.
RmW	Turn on and off pauses between words. Use <code>\RmW=1\</code> to insert pauses between words and <code>\RmW=0\</code> to turn them off.

Table 10.2: *Speech synthesis tags (continued)*

Tag	Meaning
RPit	Sets the relative pitch. The format is \RPit=number\ where number is the pitch. 100 is the default pitch.
RPrn	Sets the relative pitch range. The format is \RPrn=value\ where value is the pitch range.
RSpd	Sets the relative speed. The format is \RSpd=number\ where number is the relative speed. The default speed is 100.
Rst	Resets the speech engine to the default values.
Spd	Sets the average talking speed. The format is \Spd=number\ where number is speed in words per minute.
Vce	Changes characteristics of the voice. The format is \Vce=property=value\ where property can be "Accent", "Age", "Dialect" "Gender", "Language" or "Style". The value varies depending on the property being set.
Vol	Sets the baseline volume for speaking. The format of the tag is /Vol=number/ where number is a value between 0 (silence) and 65535 (maximum volume).

Note: Not all tags are supported by all engines.

Please note that for the ease of reference, the properties, methods, and events of TApdCustomSapiEngine are documented in the TApdSapiEngine section.

Hierarchy

TWInControl (VCL)
 TApdBaseWinControl (OoMisc)..... 8
 TApdCustomSapiEngine (AdSapiEn)

TApdSapiEngine Component

The TApdSapiEngine component exposes the functionality in the TApdCustomSapiEngine class, but does not change or add functionality to it. For ease of reference, however, the properties of TApdSapiEngine are documented here.

Hierarchy

TWinControl (VCL)

❶ TApdBaseWinControl (OoMisc)	8
TApdCustomSapiEngine (AdSapiEn)	333
TApdSapiEngine (AdSapiEn)	

Properties

CharSet	SRAutoGain	TTSOptions
Dictation	SREngines	❶ Version
SRAmplitude	SSVoices	WordList

Methods

Listen	ShowGeneralDlg	SpeakFileToFile
PauseListening	ShowLexiconDlg	SpeakStream
PauseSpeaking	ShowTrainGeneralDlg	SpeakToFile
ResumeListening	ShowTrainMicDlg	StopListening
ResumeSpeaking	Speak	StopSpeaking
ShowAboutDlg	SpeakFile	

Events

OnInterference	OnSpeakStop	OnSSError
OnPhraseFinish	OnSRError	OnSSWarning
OnPhraseHypothesis	OnSRWarning	OnTrainingRequested
OnSpeakStart	OnSSAttributeChanged	OnVUMeter

Reference Section

CharSet

property

```
property CharSet : TApdCharacterSet  
TApdCharacterSet = (csText, csIPAPhonetic, csEnginePhonetic);
```

↳ Specifies the character set used in speed synthesis.

csText indicates ASCII text, csIPAPhonetic indicates the International Phonetic Alphabet and csEnginePhonetic indicates an engine specific phonetic alphabet.

Dictation

property

```
property Dictation
```

↳ Controls whether or not dictation mode is on.

When dictation mode is on (provided the engine supports it), a larger set of words is used as the base grammar. When it is off, only the words in the WordList property will be recognized. Dictation mode may slow down speech recognition.

See Also: WordList

Listen

method

```
procedure Listen;
```

↳ Starts the speech recognition.

Calling Listen will start the speech recognition engine recognizing spoken text. The OnPhraseHypothesis and OnPhraseFinish events will fire when words are recognized. The PauseListening, and StopListening events will turn off the speech recognition.

See also: OnPhraseFinish, OnPhraseHypothesis, PauseListening, StopListening

```
property OnInterference : TApdSRInterferenceEvent
TApdSRInterferenceEvent = procedure(Sender : TObject;
  InterferenceType : TApdSRInterferenceType) of object;
TApdSRInterferenceType = (itAudioStarted, itAudioStopped,
  itDeviceOpened, itDeviceClosed, itNoise, itTooLoud,
  itTooQuiet, itUnknown);
```

↳ Indicates when the audio input to the speech recognition is garbled.

This event will fire when the audio input to the speech recognition less than optimal. The `InterferenceType` parameter indicates how the input stream was damaged.

The possible values for `OnInterference` are shown in the following table:

Value	Meaning
<code>itAudioStarted</code>	The speech recognition engine has started or resumed receiving audio data.
<code>itAudioStopped</code>	The speech recognition engine has stopped receiving audio data.
<code>itDeviceOpened</code>	The speech recognition engine has opened the input audio device.
<code>itDeviceClosed</code>	The speech recognition engine has closed the input audio device.
<code>itNoise</code>	Background noise has interfered with the speech recognition.
<code>itTooLoud</code>	The user is speaking too loud.
<code>itTooQuiet</code>	The user is speaking too quietly.
<code>itUnknown</code>	The input audio stream was garbled for an unknown reason.

OnPhraseFinish**event**

```
property OnPhraseFinish : TApdSRPhraseFinishEvent
TApdSRPhraseFinishEvent = procedure(
    Sender : TObject; const Phrase : WideString) of object;
```

↳ Indicates when the user has finished speaking a phrase.

The words the user spoke are in the Phrase parameter of the event. Multiple words may be in this value.

See also: OnPhraseHypothesis

OnPhraseHypothesis**event**

```
property OnPhraseHypothesis : TApdSRPhraseHypothesisEvent
TApdSRPhraseHypothesisEvent = procedure(
    Sender : TObject; const Phrase : WideString) of object;
```

↳ Indicates when the user has finished speaking a phrase, but the speech recognition engine is not positive on what was spoken.

The best guess of what the user said will be indicated in the Phrase parameter. Multiple words may be in this value.

See also: OnPhraseFinish

OnSpeakStart**event**

```
property OnSpeakStart : TApdSapiNotifyEvent
TApdSapiNotifyEvent = procedure(Sender : TObject) of object;
```

↳ Defines an event handler for when speech synthesis starts speaking.

This event will fire when the speech synthesis engine starts speaking to the default audio device or creates an audio file.

OnSpeakStop**event**

```
property OnSpeakStop : TApdSapiNotifyEvent
TApdSapiNotifyEvent = procedure(Sender : TObject) of object;
```

↳ Defines an event handler for when the speech synthesis stops speaking.

This event will fire when the speech synthesis engine stops speaking to the default audio device or finishes creating an audio file.

OnSRError

event

```
property OnSRError : TApdOnSapiError  
  
TApdOnSapiError = procedure(  
    Sender : TObject; Error : LongWord;  
    const Details : string; const Message : string) of object;
```

↳ Defines an event handler that is called when a speech recognition error occurs.

An error message was generated at some point during speed recognition. Error indicates the error code generated by the speech recognition engine. Details provides additional details about the error and Message provides the error messages as text.

OnSRWarning

event

```
property OnSRWarning : TApdOnSapiError  
  
TApdOnSapiError = procedure(  
    Sender : TObject; Error : LongWord;  
    const Details : string; const Message : string) of object;
```

↳ Defines an event handler that is called when a speech recognition warning occurs.

A warning message was generated at some point during speed recognition. Error indicates the error code generated by the speech recognition engine. Details provides additional details about the warning and Message provides the error messages as text.

OnSSAttributeChanged

event

```
property OnSSAttributeChanged : TApdSSAttributeChanged  
  
TApdSSAttributeChanged = procedure(  
    Sender : TObject; Attribute: Integer) of object;
```

↳ Indicates when an speech synthesis engine attribute has changed.

Multiple applications may be using the same speech synthesis engine. Any one of those can change attributes. This event exists to let the application know that an attribute has changed.

OnSSError**event**

```
property OnSSError : TApdOnSapiError

TApdOnSapiError = procedure(
    Sender : TObject; Error : LongWord; const Details : string;
    const Message : string) of object;
```

↳ Defines an event handler that is called when a speech synthesis error occurs.

An error message was generated at some point during the speech synthesis. Error indicates the error code generated by the speech synthesis. Details provides additional details about the error and Message provides the error messages as text.

OnSSWarning**event**

```
property OnSSWarning : TApdOnSapiError

TApdOnSapiError = procedure(
    Sender : TObject; Error : LongWord; const Details : string;
    const Message : string) of object;
```

↳ Defines an event handler that is called when a speech synthesis warning occurs.

A warning message was generated at some point during the speech synthesis. Error indicates the error code generated by the speech synthesis. Details provides additional details about the warning and Message provides the error messages as text.

OnTrainingRequested**event**

```
property OnTrainingRequested : TApdSRTrainingRequestedEvent

TApdSRTrainingRequestedEvent = procedure(
    Sender : TObject; Training : TApdSRTrainingType) of object;

TApdSRTrainingType = set of (
    ttCurrentMic, ttCurrentGrammar, ttGeneral);
```

↳ Indicates when the speech recognition engine needs further training.

The OnTrainingRequested event will fire when the speech recognition requests further training. The Training parameter indicates the type of training required. The ShowTrainGeneralDlg and ShowTrainMicDlg methods can be called to provide the necessary training.

See also: ShowTrainGeneralDlg, ShowTrainMicDlg

```
property OnVUMeter : TApdSRVUMeterEvent  
  
TApdSRVUMeterEvent = procedure(  
    Sender : TObject; Level : Integer) of object;
```

↳ Indicates the volume of the spoken data.

This event provides a rough indicator of the volume of the spoken data heard by the speech recognition engine. While the speech recognition engine is listening, this event will fire approximately 8 times a second.

See also: Listen, StopListening, SRAmplitude

PauseListening**method**

```
procedure PauseListening;
```

↳ Pauses speech recognition.

PauseListening pauses the speech recognition. ResumeListening must be called to continue speech recognition. Pauses are nested. For example, if the engine is paused twice, it will take two resumes to restart the listening.

The speech recognition engine may lose data while it is paused.

See also: ResumeListening

PauseSpeaking**method**

```
procedure PauseSpeaking;
```

↳ Pauses speaking.

This method pauses speech synthesis. To resume speaking, call ResumeSpeaking.

See also: ResumeSpeaking

ResumeListening**method**

```
procedure ResumeListening;
```

↳ Resumes speech recognition.

This method resumes speech recognition after it was paused using PauseListening. Pauses are nested. For example, if the engine is paused twice, it will take two resumes to restart the listening.

See also: PauseListening

ResumeSpeaking

method

```
procedure ResumeSpeaking;
```

- ↳ Resumes speaking after speaking was paused with `PauseSpeaking`.

This method will resume speech synthesis after it was paused using `PauseSpeaking`.

See also: `PauseSpeaking`

ShowAboutDlg

method

```
procedure ShowAboutDlg(const Caption : string);
```

- ↳ Displays the speech synthesis about dialog box.

The caption parameter provides a caption to the dialog box.

ShowGeneralDlg

method

```
procedure ShowGeneralDlg(const Caption : string);
```

- ↳ Displays the speech synthesis general settings dialog box.

The speech synthesis general settings dialog allows the user to access various aspects of the speech synthesis engine. This may include speaker specific information like gender and age, various engine optimizations, and access to other speech synthesis dialog boxes.

The caption parameter provides a caption to the dialog box.

ShowLexiconDlg

method

```
procedure ShowLexiconDlg(const Caption : string);
```

- ↳ Displays a dialog box for editing mispronounced words.

This method will display a dialog box that allows the user to edit the pronunciation lexicon used by speech synthesis.

The caption parameter provides a caption to the dialog box.

ShowTrainGeneralDlg

method

```
procedure ShowTrainGeneralDlg(const Caption : string);
```

↳ Displays the general training dialog box.

This method will display the general training dialog box. This dialog allows the user to train the speech recognition engine using a preselected set of words. The speech recognition engine may request this dialog to be displayed via the OnTrainingRequested event.

The caption parameter provides a caption to the dialog box.

See also: OnTrainingRequested

ShowTrainMicDlg

method

```
procedure ShowTrainMicDlg(const Caption : string);
```

↳ Displays the microphone training dialog box.

This method will display a dialog box in which the user can train the speech recognition engine for a specific microphone. The speech recognition engine can request for this training dialog to be displayed via the OnTrainingRequested event.

The Caption parameter allows the application to specify a caption for the title bar of the dialog.

Note: Not all speech recognition events will support this dialog.

See also: OnTrainingRequested

Speak

method

```
procedure Speak(Text : string);
```

↳ Speaks the specified text to the default audio device.

Calling this method will cause the text specified by the Text parameter to be spoken to the default audio device.

If toTagged is specified in the TTsoptions property, the Text parameter can contain embedded tags to control the behavior of the speech synthesis engine.

The format of the Text parameter is controlled by the CharSet property.

The speech synthesis engine will store the text in a series of buffers. Since each buffer is spoken independently, complete sentences should be passed to the speech synthesis engine.

Normally, the speech synthesis engine will start speaking the text as soon as it passed into one of the `Speak` methods (`Speak`, `SpeakFile`, `SpeakFileToFile`, `SpeakStream`, or `SpeakToFile`). However, if `PauseSpeaking` has been called before one of the `Speak` methods, the speaking will not occur until `ResumeSpeaking` is called.

`StopSpeaking` can be called at any time to halt the flow of speech. When `StopSpeaking` is called, the text that is spoken and any text that is queued will not be spoken. Text that is queued by using one of the `Speak` methods after the call to `StopSpeaking` will be spoken in the usual fashion.

See also: `CharSet`, `PauseSpeaking`, `ResumeSpeaking`, `SpeakFile`, `SpeakFileToFile`, `SpeakStream`, `SpeakToFile`, `StopSpeaking`, `TTSOptions`

SpeakFile

method

```
procedure SpeakFile(FileName : string);
```

↳ Speaks the contents of the specified file to the default audio device.

Calling this method will cause the contents of the specified file to be read over the default audio device.

Note: Reading binary files may cause unexpected behavior.

If `toTagged` is specified in the `TTSOptions` property, the `Text` parameter can contain embedded tags to control the behavior of the speech synthesis engine.

The format of the `Text` parameter is controlled by the `CharSet` property.

The speech synthesis engine will store the text in a series of buffers. Since each buffer is spoken independently, complete sentences should be passed to the speech synthesis engine.

Normally, the speech synthesis engine will start speaking the text as soon as it passed into one of the `Speak` methods (`Speak`, `SpeakFile`, `SpeakFileToFile`, `SpeakStream`, or `SpeakToFile`). However, if `PauseSpeaking` has been called before one of the `Speak` methods, the speaking will not occur until `ResumeSpeaking` is called.

`StopSpeaking` can be called at any time to halt the flow of speech. When `StopSpeaking` is called, the text that is spoken and any text that is queued will not be spoken. Text that is queued by using one of the `Speak` methods after the call to `StopSpeaking` will be spoken in the usual fashion.

See also: `CharSet`, `PauseSpeaking`, `ResumeSpeaking`, `Speak`, `SpeakFileToFile`, `SpeakStream`, `SpeakToFile`, `StopSpeaking`, `TTSOptions`

```
procedure SpeakFileToFile(const InFile, OutFile : string);
```

↳ Converts the contents of the specified file into an audio file.

This method will create an audio file named by the OutFile parameter containing the spoken contents of the InFile parameter. The output file is formatted as xxxxxx.

Note: Reading binary files may cause unexpected behavior.

If toTagged is specified in the TTSTOptions property, the Text parameter can contain embedded tags to control the behavior of the speech synthesis engine.

The format of the Text parameter is controlled by the CharSet property.

The speech synthesis engine will store the text in a series of buffers. Since each buffer is spoken independently, complete sentences should be passed to the speech synthesis engine.

Normally, the speech synthesis engine will start speaking the text as soon as it passed into one of the Speak methods (Speak, SpeakFile, SpeakFileToFile, SpeakStream, or SpeakToFile). However, if PauseSpeaking has been called before one of the Speak methods, the speaking will not occur until ResumeSpeaking is called.

StopSpeaking can be called at any time to halt the flow of speech. When StopSpeaking is called, the text that is spoken and any text that is queued will not be spoken. Text that is queued by using one of the Speak methods after the call to StopSpeaking will be spoken in the usual fashion.

See also: CharSet, PauseSpeaking, ResumeSpeaking, Speak, SpeakFile, SpeakStream, SpeakToFile, StopSpeaking, TTSTOptions

SpeakStream**method**

```
procedure SpeakStream(Stream : TStream; FileName : string);
```

↳ Speaks the contents of the specified stream.

This method will speak the contents of the specified stream.

Note: Reading binary files may cause unexpected behavior.

If toTagged is specified in the TTSTOptions property, the Text parameter can contain embedded tags to control the behavior of the speech synthesis engine.

The format of the Text parameter is controlled by the CharSet property.

The speech synthesis engine will store the text in a series of buffers. Since each buffer is spoken independently, complete sentences should be passed to the speech synthesis engine.

Normally, the speech synthesis engine will start speaking the text as soon as it passed into one of the `Speak` methods (`Speak`, `SpeakFile`, `SpeakFileToFile`, `SpeakStream`, or `SpeakToFile`). However, if `PauseSpeaking` has been called before one of the `Speak` methods, the speaking will not occur until `ResumeSpeaking` is called.

`StopSpeaking` can be called at any time to halt the flow of speech. When `StopSpeaking` is called, the text that is spoken and any text that is queued will not be spoken. Text that is queued by using one of the `Speak` methods after the call to `StopSpeaking` will be spoken in the usual fashion.

See also: `CharSet`, `PauseSpeaking`, `ResumeSpeaking`, `Speak`, `SpeakFile`, `SpeakFileToFile`, `SpeakToFile`, `StopSpeaking`, `TTSOptions`

SpeakToFile

method

```
procedure SpeakToFile (const Text, FileName : string);
```

↳ Converts the specified text into an audio file.

Calling this method will cause the text specified by the `Text` parameter to be converted into an audio file.

Note: Reading binary files may cause unexpected behavior.

If `toTagged` is specified in the `TTSOptions` property, the `Text` parameter can contain embedded tags to control the behavior of the speech synthesis engine.

The format of the `Text` parameter is controlled by the `CharSet` property.

The speech synthesis engine will store the text in a series of buffers. Since each buffer is spoken independently, complete sentences should be passed to the speech synthesis engine.

Normally, the speech synthesis engine will start speaking the text as soon as it passed into one of the `Speak` methods (`Speak`, `SpeakFile`, `SpeakFileToFile`, `SpeakStream`, or `SpeakToFile`). However, if `PauseSpeaking` has been called before one of the `Speak` methods, the speaking will not occur until `ResumeSpeaking` is called.

`StopSpeaking` can be called at any time to halt the flow of speech. When `StopSpeaking` is called, the text that is spoken and any text that is queued will not be spoken. Text that is queued by using one of the `Speak` methods after the call to `StopSpeaking` will be spoken in the usual fashion.

See also: `CharSet`, `PauseSpeaking`, `ResumeSpeaking`, `Speak`, `SpeakFile`, `SpeakFileToFile`, `SpeakStream`, `StopSpeaking`, `TTSOptions`

SREngines

run-time property

```
property SREngines : TApdAudioInDevice
```

- ↳ Lists the available text to speech engines and specifies the current engine.

SSVoices

run-time property

```
property SSVoices : TApdAudioOutDevice
```

- ↳ Lists the available speech recognition engines and specifies the current engine.

StopListening

method

```
procedure StopListening;
```

- ↳ Stops speech recognition.

This method will halt the speech recognition engine. If you call this method before calling Listen, an error will be generated.

See also: Listen, PauseListening, ResumeListening

SRAmplitude

read-only, run-time property

```
property SRAmplitude : Word
```

- ↳ Provides the loudness of words spoken to the speech recognition engine.

Provides the current volume of the audio data spoken to the speech recognition engine. This value is the same as the Level parameter passed into the OnVUMeter event.

See also: Listen, OnVUMeter

SRAutoGain

property

```
property SRAutoGain : Integer
```

- ↳ Controls the automatic gain settings of the speech recognition engine.

This property controls the degree to which the speech recognition engine can automatically adjust the gain. Legal values range from 0, in which the speech recognition engine cannot adjust the gain, to 100, in which the gain is entirely controlled by the speech recognition engine.

Values between 0 and 100 set the percentage of gain adjustment that the speech engine can make. For example, a SRAutoGain value of 50 would cause the speech recognition engine to only adjust the gain half as much as it normally would.

StopSpeaking

method

```
procedure StopSpeaking;
```

↳ Stops speaking.

StopSpeaking halts the flow of speech being generated by the speech synthesis engine. This will also cancel any queued speech requests.

See also: Speak, PauseSpeaking, ResumeSpeaking

TTSOptions

property

```
property TTSOptions : TApdTTSOptions
```

```
TApdTTSOptions = set of (toTagged);
```

↳ Specifies options used during speech synthesis.

This property sets the speech synthesis options that will be used when text is spoken. The toTagged value allows the text to be spoken to contain embedded tags.

WordList

property

```
property WordList : TStringList
```

↳ Specifies expected words to be used by the speech recognition engine.

This property specifies those words that the speech recognition will be listening for. If Dictation is False, only the words in this property will the speech engine listen for. Otherwise, the speech engine will listen for the specified words in addition to those words in the default dictation word list. When Dictation is True, this property is optional.

See Also: Dictation

TApdSapiPhonePrompts

The `TApdSapiPhonePrompts` class stores default prompts for use by the `TApdCustomSapiPhone` component. These values are spoken to the users when information is requested by one of the `AskFor` methods in the `TApdCustomSapiPhone` component.

Hierarchy

TPersistent (VCL)

 TApdSapiPhonePrompts (AsSapiPh)

Properties

AskAreaCode	Help2	TooManyDigits
AskLastFour	Main	Unrecognized
AskNextThree	Main2	Where
Help	TooFewDigits	Where2

Reference Section

AskAreaCode

property

```
property AskAreaCode : string
```

↳ Prompt for user to enter an area code.

AskAreaCode specifies the spoken prompt used to ask the user for their area code. AskForPhoneNumber and AskForPhoneNumberEx use this when the speech recognition engine is unable to understand the full phone number. If the speech recognition engine fails to recognize the full phone number, the user will first be prompted for their area code using the AskAreaCode property, followed by the middle three digits of the phone number using the AskNextThree property and finally the last four digits using the prompt in the AskLastFour property.

See also: AskLastFour, AskNextThree

AskLastFour

property

```
property AskLastFour : string
```

↳ Prompt for user to enter the last four digits of the phone number.

AskLastFour specifies the spoken prompt used to ask the user for the last four digits of their phone number. AskForPhoneNumber and AskForPhoneNumberEx use this when the speech recognition engine is unable to understand the full phone number. If the speech recognition engine fails to recognize the full phone number, the user will first be prompted for their area code using the AskAreaCode property, followed by the middle three digits of the phone number using the AskNextThree property and finally the last four digits using the prompt in the AskLastFour property.

See also: AskAreaCode, AskNextThree

AskNextThree

property

```
property AskNextThree : string
```

- ↳ Prompt for user to enter the middle three digits of the phone number.

AskNextThree specifies the spoken prompt used to ask the user for middle three digits of their phone number. AskForPhoneNumber and AskForPhoneNumberEx use this when the speech recognition engine is unable to understand the full phone number. If the speech recognition engine fails to recognize the full phone number, the user will first be prompted for an area code using the AskAreaCode property, followed by the middle three digits of the phone number using the AskNextThree property and finally the last four digits using the prompt in the AskLastFour property.

See also: AskAreaCode, AskLastFour

Help

property

```
property Help : string
```

- ↳ Specifies help text that will be spoken if the user asks for help.

If the user asks for help during a call, first the text in the Help property will be spoken. If the user asks for help a second time, the text in Help2 will be spoken.

See also: Help2

Help2

property

```
property Help2 : string
```

- ↳ Specifies help text that will be spoken if the user asks for help a second time.

If the user asks for help during a call, first the text in the Help property will be spoken. If the user asks for help a second time, the text in Help2 will be spoken.

See also: Help

Main**property**`property Main : string`

↳ Specifies the main prompt spoken to the user.

Main is the first prompt spoken to the user by the AskFor methods in the TApdCustomSapiPhone component. The contents of this property should reflect what the user is being asked for. If the main prompt is repeated, the value in Main2 will be spoken instead of Main.

See also: Main2

Main2**property**`property Main2 : string`

↳ Specifies the main prompt spoken to the user if the main prompt needs to be repeated.

Main2 is alternative and generally shorter text spoken if the main prompt should need to be repeated. The contents of this property should reflect what the user is being asked for. If the main prompt is repeated, the value in Main2 will be spoken instead of Main.

See also: Main

TooFewDigits**property**`property TooFewDigits : string`

↳ Prompt spoken to the user if too few digits were specified for an extension.

See also: TooManyDigits

TooManyDigits**property**`property TooManyDigits : string`

↳ Prompt spoken to the user if too many digits were specified for an extension.

See also: TooFewDigits

Unrecognized

property

```
property Unrecognized : string
```

- ↳ Prompt spoken to the user to indicate the speech recognition engine did not understand what they said.

Where

property

```
property Where : string
```

- ↳ Help text spoken to the user if they ask where they are.

The Where prompt is spoken in response when the user asks, “Where Am I?” If the user asks where they are again, the text in Where2 will be spoken.

See also: Where2

Where2

property

```
property Where2 : string
```

- ↳ Help text spoken to the user if they ask where they are a second time.

If the user asks where they are again for a second time, the text in Where2 will be spoken.

See also: Where

TApdCustomSapiPhone Component

The TApdCustomSapiPhone component is a speech-aware TAPI device. This component will automatically link to a TApdCustomSapiEngine component. When a call is active, speech synthesis and recognition will take place over the voice TAPI connection.

For this to work you will need a voice capable modem and speech synthesis and recognition engines that have been optimized for telephone usage.

Note: Speech recognition may lag behind what is actually spoken. Setting the Dictation property of the TApdCustomSapiEngine component to False helps with this, but the operating system, modem type, amount of memory and other factors influence the speech recognition lag.

☛ **Caution:** When dialing a voice call, the standard TAPI Service Provider will return that it is connected as soon as it has dialed. This is a known problem in the standard TAPI Service Provider provided with Windows.

Most SAPI calls consist of a series of questions for the user. These are handled by several methods: AskForDate, AskForTime, AskForPhoneNumber and related methods. These methods will prompt the user for some information and then collect the voice reply and interpret the reply into a usable value.

The return code for the AskFor methods lets you know what the state of the call is. The reply codes are detailed in Table 10.3. Several of these values are controlled by the Options property.

Table 10.3: *AskFor methods return codes*

Return code	Description
prOk	The method was able to get a reply from the user.
prAbort	An unrecoverable error occurred in the method.
prNoResponse	The user did not respond.
prOperator	The user asked to speak to an operator. This can be disabled in the Options property.
prHangUp	The user either asked to hang up or did hang up. Asking to hang up can be disabled in the Options property.
prBack	The user asked to go back. This can be disabled in the Options property.

Table 10.3: *AskFor methods return codes (continued)*

Return code	Description
<code>prWhere</code>	The user asked where he or she is. This is not normally returned.
<code>prHelp</code>	The user asked for help. This is not normally returned.
<code>prRepeat</code>	The user asked for the current prompt to be repeated. This is not normally returned.
<code>prSpeakFaster</code>	The user asked for the speech synthesis engine to speak faster. This is not normally returned. This can be disabled in the <code>Options</code> property.
<code>prSpeakSlower</code>	The user asked for the speech synthesis engine to speak slower. This is not normally returned. This can be disabled in the <code>Options</code> property.
<code>prCheck</code>	A reply was returned from the user, but the method had difficulty in interpreting it. The returned value may be unreliable. This most applies to asking for dates and times. For these methods, the textual data is returned to aid in interpreting the value.
<code>prError</code>	The reply from user was entirely unintelligible.
<code>prUnknown</code>	An unexpected value was returned.

Full and half duplex telephony connections

Most voice modems are half duplex—that is, they cannot listen and talk at the same time. In addition, the TAPI Service Provider may also be half duplex. UnimodemV, even under Windows 2000, has half duplex wave drivers. Even if you do have a full duplex modem, the wave drivers will prevent you from operating in full duplex mode.

Please note that for the ease of reference, the properties methods and events of `TApdCustomSapiPhone` are documented in the `TApdSapiPhone` section.

Hierarchy

TComponent

 TApdBaseComponent (OoMisc) 8

 TApdCustomTapiDevice (AdTapi)

 TApdCustomSapiPhone (AdSapiPh)

TApdSapiPhone Component

The TApdSapiPhone component exposes the functionality in the TApdCustomSapiPhone class, but does not change or add functionality to it. For ease of reference, however, the properties of TApdSapiEngine are documented here.

Hierarchy

TComponent	
❶ TApdBaseComponent (OoMisc)	8
TApdCustomTapiDevice (AdTapi)	
TApdCustomSapiPhone (AdSapiPh)	
TApdSapiPhone (AdSapiPh)	

Properties

NoAnswerMax	Options	❶ Version
NoAnswerTime	Prompts	
NumDigits	SapiEngine	

Methods

AskForDate	AskForListEx	AskForTime
AskForDateEx	AskForPhoneNumber	AskForTimeEx
AskForExtension	AskForPhoneNumberEx	AskForYesNo
AskForExtensionEx	AskForSpelling	AskForYesNoEx
AskForList	AskForSpellingEx	Speak

Reference Section

AskForDate

method

```
function AskForDate(  
    var OutDate : TDateTime; var ParsedText : string;  
    NewPrompt1 : string) : TApdSapiPhoneReply;  
  
TApdSapiPhoneReply = (prOk, prAbort, prNoResponse,  
    prOperator, prHangUp, prBack, prWhere, prHelp, prRepeat,  
    prSpeakFaster, prSpeakSlower, prCheck, prError, prUnknown);
```

↳ Asks the user for a date.

The user will be prompted for the date with the value in NewPrompt1. If this value is blank, then the default main prompt in the Prompts property will be used. Other prompts used in asking for the date are obtained from the Prompts property. If you want to override the prompts other than the main prompt, use the AskForDateEx method.

The date is returned in OutDate as a TDateTime. The date returned may be unreliable. If this is the case, the method will return prCheck and the parsed text spoken by the user will be returned in ParsedText.

See also: AskForDateEx, Prompts

AskForDateEx

method

```
function AskForDateEx(var OutDate : TDateTime;  
    var ParsedText : string; NewPrompt1 : string;  
    NewPrompt2 : string; NewHelp1 : string;  
    NewHelp2 : string; NewWhere1 : string;  
    NewWhere2 : string) : TApdSapiPhoneReply;  
  
TApdSapiPhoneReply = (prOk, prAbort, prNoResponse,  
    prOperator, prHangUp, prBack, prWhere, prHelp, prRepeat,  
    prSpeakFaster, prSpeakSlower, prCheck, prError,  
    prUnknown);
```

↳ Asks the user for a date.

The user will be prompted for the date using the prompts passed into the method. These prompts correspond to the values in the Prompts property. If a value is blank, then the value from the corresponding prompt in the Prompts property will be used. If you only need to override the main prompt, use the AskForDate method.

The date is returned in OutDate as a TDateTime. The date returned may be unreliable. If this is the case, the method will return prCheck and the parsed text spoken by the user will be returned in ParsedText.

See also: AskForDate, Prompts

AskForExtension

method

```
function AskForExtension(var Extension : string;
    NewPrompt1 : string) : TApdSapiPhoneReply;

TApdSapiPhoneReply = (prOk, prAbort, prNoResponse,
    prOperator, prHangUp, prBack, prWhere, prHelp, prRepeat,
    prSpeakFaster, prSpeakSlower, prCheck, prError,
    prUnknown);
```

🔗 Asks the user for a telephone extension.

The user will be prompted for the telephone extension with the value in NewPrompt1. If this value is blank, then the default main prompt in the Prompts property will be used. Other prompts used in asking for the extension are obtained from the Prompts property. If you want to override the prompts other than the main prompt, use the AskForExtensionEx method.

The extension is assumed to be a series of digits and is returned as a string in the Extension parameter.

See also: AskForExtensionEx, Prompts

```
function AskForExtensionEx(var Extension : string;
    NewPrompt1 : string; NewPrompt2 : string;
    NewTooManyDigits : string; NewTooFewDigits : string;
    NewNumDigits : Integer; NewHelp1 : string;
    NewHelp2 : string; NewWhere1 : string;
    NewWhere2 : string) : TApdSapiPhoneReply;

TApdSapiPhoneReply = (prOk, prAbort, prNoResponse,
    prOperator, prHangUp, prBack, prWhere, prHelp, prRepeat,
    prSpeakFaster, prSpeakSlower, prCheck, prError,
    prUnknown);
```

↳ Asks the user for a telephone extension.

The user will be prompted for the telephone extension using the prompts passed into the method. These prompts correspond to the values in the Prompts property. If a value is blank, then the value from the corresponding prompt in the Prompts property will be used. If you only need to override the main prompt, use the AskForExtension method.

The extension is assumed to be a series of digits and is returned as a string in the Extension parameter.

See also: AskForExtension, Prompts

AskForList

method

```
function AskForList(List : TStringList; var OutIndex : Integer;
    NewPrompt1 : string) : TApdSapiPhoneReply;

TApdSapiPhoneReply = (prOk, prAbort, prNoResponse,
    prOperator, prHangUp, prBack, prWhere, prHelp, prRepeat,
    prSpeakFaster, prSpeakSlower, prCheck, prError,
    prUnknown);
```

↳ Asks the user for a value in a list.

The user will be prompted for an entry in a list with the value in NewPrompt1. If this value is blank, then the default main prompt in the Prompts property will be used. Other prompts used in asking for the value are obtained from the Prompts property. If you want to override the prompts other than the main prompt, use the AskForListEx method.

The list of items the user is asked to select an item from is provided in the List parameter. If the user replies with one of the items in the list, that value will be returned in the OutIndex parameter. If no item is selected, OutIndex will be set to -1.

See also: AskForListEx, Prompts

```
function AskForListEx(List : TStringList;
    var OutIndex : Integer; NewPrompt1 : string;
    NewPrompt2 : string; NewHelp1 : string;
    NewHelp2 : string; NewWhere1 : string;
    NewWhere2 : string) : TApdSapiPhoneReply;

TApdSapiPhoneReply = (prOk, prAbort, prNoResponse,
    prOperator, prHangUp, prBack, prWhere, prHelp, prRepeat,
    prSpeakFaster, prSpeakSlower, prCheck, prError,
    prUnknown);
```

↳ Asks the user for a value in a list.

The user will be prompted for the value in a list using the prompts passed into the method. These prompts correspond to the values in the Prompts property. If a value is blank, then the value from the corresponding prompt in the Prompts property will be used. If you only need to override the main prompt, use the AskForList method.

The list of items the user is asked to select an item from is provided in the List parameter. If the user replies with one of the items in the list, that value will be returned in the OutIndex parameter. If no item is selected, OutIndex will be set to -1.

See also: AskForList, Prompts

AskForPhoneNumber**method**

```
function AskForPhoneNumber(var PhoneNumber : string;
    NewPrompt1 : string) : TApdSapiPhoneReply;

TApdSapiPhoneReply = (prOk, prAbort, prNoResponse,
    prOperator, prHangUp, prBack, prWhere, prHelp, prRepeat,
    prSpeakFaster, prSpeakSlower, prCheck, prError,
    prUnknown);
```

↳ Asks the user for a phone number.

The user will be prompted for the phone number with the value in NewPrompt1. If this value is blank, then the default main prompt in the Prompts property will be used. Other prompts used in asking for the phone number are obtained from the Prompts property. If you want to override the prompts other than the main prompt, use the AskForPhoneNumberEx method.

If `AskForPhoneNumber` is unable to obtain the phone number, it will ask for the number in pieces using the `AskAreaCode`, `AskNextThree` and `AskLastFour` prompts.

The phone number is returned as a string in the `PhoneNumber` property.

See also: `AskForPhoneNumberEx`, Prompts

AskForPhoneNumberEx

method

```
function AskForPhoneNumberEx(var PhoneNumber : string;
    NewPrompt1 : string; NewPrompt2 : string;
    NewAskAreaCode : string; NewAskNextThree : string;
    NewAskLastFour : string; NewHelp1 : string;
    NewHelp2 : string; NewWhere1 : string;
    NewWhere2 : string) : TApdSapiPhoneReply;

TApdSapiPhoneReply = (prOk, prAbort, prNoResponse,
    prOperator, prHangUp, prBack, prWhere, prHelp, prRepeat,
    prSpeakFaster, prSpeakSlower, prCheck, prError,
    prUnknown);
```

👉 Asks the user for a phone number.

The user will be prompted for the phone number using the prompts passed into the method. These prompts correspond to the values in the Prompts property. If a value is blank, then the value from the corresponding prompt in the Prompts property will be used. If you only need to override the main prompt, use the `AskForPhoneNumber` method.

If `AskForPhoneNumber` is unable to obtain the phone number, it will ask for the number in pieces using the `NewAskAreaCode`, `NewAskNextThree` and `NewAskLastFour` parameters or the corresponding values in Prompts.

The phone number is returned as a string in the `PhoneNumber` property.

See also: `AskForPhoneNumber`, Prompts

```
function AskForSpelling(  
    var SpelledWord : string;  
    NewPrompt1 : string) : TApdSapiPhoneReply;  
  
TApdSapiPhoneReply = (prOk, prAbort, prNoResponse,  
    prOperator, prHangUp, prBack, prWhere, prHelp, prRepeat,  
    prSpeakFaster, prSpeakSlower, prCheck, prError,  
    prUnknown);
```

↳ Asks the user for a spelled word.

The user will be prompted using the value in NewPrompt1 to spell out a reply. If NewPrompt1 is blank, then the default main prompt in the Prompts property will be used. Other prompts used in asking for the spelled item are obtained from the Prompts property. If you want to override the prompts other than the main prompt, use the AskForSpellingEx method.

The reply from the user is returned in the SpelledWord parameter.

See also: AskForSpellingEx, Prompts

AskForSpellingEx**method**

```
function AskForSpellingEx(  
    var SpelledWord : string; NewPrompt1 : string;  
    NewPrompt2 : string; NewHelp1 : string; NewHelp2 : string;  
    NewWhere1 : string; NewWhere2 : string) : TApdSapiPhoneReply;  
  
TApdSapiPhoneReply = (prOk, prAbort, prNoResponse,  
    prOperator, prHangUp, prBack, prWhere, prHelp, prRepeat,  
    prSpeakFaster, prSpeakSlower, prCheck, prError,  
    prUnknown);
```

↳ Asks the user for a spelled word.

The user will be prompted using the prompts passed into the method to spell out a reply. The prompts correspond to the values in the Prompts property. If a value is blank, then the value from the corresponding prompt in the Prompts property will be used. If you only need to override the main prompt, use the AskForSpelling method.

The reply from the user is returned in the SpelledWord parameter.

See also: AskForSpelling, Prompts


```
function AskForTime(
    var OutTime : TDateTime; var ParsedText : string;
    NewPrompt1 : string) : TApdSapiPhoneReply;

TApdSapiPhoneReply = (prOk, prAbort, prNoResponse,
    prOperator, prHangUp, prBack, prWhere, prHelp, prRepeat,
    prSpeakFaster, prSpeakSlower, prCheck, prError,
    prUnknown);
```

↳ Asks the user for a time.

The user will be prompted for the time with the value in NewPrompt1. If this value is blank, then the default main prompt in the Prompts property will be used. Other prompts used in asking for the time are obtained from the Prompts property. If you want to override the prompts other than the main prompt, use the AskForTimeEx method.

The date is returned in OutTime as a TDateTime. The date returned may be unreliable. If this is the case, the method will return prCheck and the parsed text spoken by the user will be returned in ParsedText.

See also: AskForTimeEx, Prompts

AskForTimeEx

method

```
function AskForTimeEx(
    var OutTime : TDateTime; var ParsedText : string;
    NewPrompt1 : string; NewPrompt2 : string; NewHelp1 : string;
    NewHelp2 : string; NewWhere1 : string;
    NewWhere2 : string) : TApdSapiPhoneReply;

TApdSapiPhoneReply = (prOk, prAbort, prNoResponse,
    prOperator, prHangUp, prBack, prWhere, prHelp, prRepeat,
    prSpeakFaster, prSpeakSlower, prCheck, prError,
    prUnknown);
```

↳ Asks the user for a time.

The user will be prompted for the time using the prompts passed into the method. These prompts correspond to the values in the Prompts property. If a value is blank, then the value from the corresponding prompt in the Prompts property will be used. If you only need to override the main prompt, use the AskForTime method.

The date is returned in OutTime as a TDateTime. The date returned may be unreliable. If this is the case, the method will return prCheck and the parsed text spoken by the user will be returned in ParsedText.

See also: AskForTime, Prompts

```
function AskForYesNo(  
    var Reply : Boolean; NewPrompt1 : string) : TApdSapiPhoneReply;  
  
TApdSapiPhoneReply = (prOk, prAbort, prNoResponse,  
    prOperator, prHangUp, prBack, prWhere, prHelp, prRepeat,  
    prSpeakFaster, prSpeakSlower, prCheck, prError,  
    prUnknown);
```

🔗 Asks the user for a yes or no reply.

The user will be prompted for a yes or no reply using the value in NewPrompt1. If this value is blank, then the default main prompt in the Prompts property will be used. Other prompts used in asking for the yes or no reply are obtained from the Prompts property. If you want to override the prompts other than the main prompt, use the AskForYesNoEx method.

The date is returned in the Reply parameter as a Boolean. A value of True indicates yes and a value of False indicates no.

See also: AskForYesNoEx, Prompts

```
function AskForYesNoEx(  
    var Reply : Boolean; NewPrompt1 : string;  
    NewPrompt2 : string; NewHelp1 : string;  
    NewHelp2 : string; NewWhere1 : string;  
    NewWhere2 : string) : TApdSapiPhoneReply;  
  
TApdSapiPhoneReply = (prOk, prAbort, prNoResponse,  
    prOperator, prHangUp, prBack, prWhere, prHelp, prRepeat,  
    prSpeakFaster, prSpeakSlower, prCheck, prError,  
    prUnknown);
```

🔗 Asks the user for a yes or no reply.

The user will be prompted for the reply using the prompts passed into the method. These prompts correspond to the values in the Prompts property. If a value is blank, then the value from the corresponding prompt in the Prompts property will be used. If you only need to override the main prompt, use the AskForYesNo method.

The date is returned in the Reply parameter as a Boolean. A value of True indicates yes and a value of False indicates no.

See also: AskForYesNo, Prompts

NoAnswerMax

property

```
property NoAnswerMax : Integer
```

↪ Specifies the number of times the user can not answer a prompt before returning an error.

This is used by the AskFor methods. If there is no answer, these methods will return prNoResponse.

See also: NoAnswerTime

NoAnswerTime

property

```
property NoAnswerTime : Integer
```

↪ Specifies in seconds the amount of time for a user to answer a prompt.

This is used by the AskFor methods. If the user does not reply in NoAnswerTime seconds NoAnswerMax times, the prNoResponse error will be returned by the method.

See also: NoAnswerMax

NumDigits

property

```
property NumDigits : Integer
```

↪ Specifies the number of digits in the extension asked for by AskForExtension and AskForExtensionEx.

10

Options

property

```
property Options : TApdSapiPhoneSettings
```

```
TApdSapiPhoneSettings = set of (psVerify, psCanGoBack,  
    psDisableSpeedChange, psEnableOperator,  
    psEnableAskHangup, psFullDuplex);
```

↪ Specifies options supported by the call.

Various options can be enabled for a SAPI call. These are detailed in the following table:

Option	Meaning
psVerify	Every AksFor method will confirm the user's reply automatically.
psCanGoBack	The user can ask to go back. This enables the prBack return value in the AskFor methods.
psDisableSpeedChange	If selected, the user cannot ask for the speech synthesis to speak faster or slower.
psEnableOperator	The user can ask for an operator. This enables the prOperator return value in the AskFor methods.
psEnableAskHangup	The user can ask to hang up. This enables the prHangUp return value in the AskFor methods. This will not automatically hang up.
psFullDuplex	Enables the telephony control to listen at the same time it is speaking. In most cases, this cannot be turned on. Full duplex depends on the modem drivers in addition to the TAPI service provider.

Prompts	property
----------------	-----------------

property Prompts : TApdSapiPhonePrompts

↳ Specifies the default spoken prompts used in getting information from the user.

SapiEngine	property
-------------------	-----------------

property SapiEngine : TApdCustomSapiEngine

↳ Specifies the TApdCustomSapiEngine component to use for speech synthesis and recognition.

Speak	method
--------------	---------------

procedure Speak(const Text : string);

↳ Speaks the specified text to the user.

Chapter 11: Remote Access Service (RAS) Components

Async Professional provides two RAS components that make it easy to add RAS dialing capability to your application. RAS dialing is performed via the Microsoft RAS API.

The TAPdRasDialer provides an easy to use interface to the Microsoft Remote Access Services API to establish a connection to another computer via Dialup Networking. In addition, the component also provides a set of standard functions for manipulating phonebook entries and displaying dial status information.

The TAPdRasStatus component provides a dialing status dialog for use on machines whose RAS DLL does not provide a status display.

Overview

Remote Access Services (RAS) is the Windows service that handles dial-up networking connections via modem. RAS is installed by default on most Win9x machines. On NT machines, however, RAS is not installed until you add a modem device to the system configuration. On Windows 9x/ME, dialup-networking automatically starts when an application attempts to connect to a remote machine. Under Windows NT, however, this does not occur and it's up to the user to explicitly dial with RAS.

Key to RAS operations is Window's concept of a phonebook. An entry in the phonebook contains the user's dialup networking connection settings which includes the modem to use, the phone number to dial, the network connection settings, and so on. In Windows NT, a phonebook is contained in a phonebook file (a file with a .PBK extension) and there can be more than one phonebook. In Windows 9x/ME only one phonebook exists and is stored in the registry.

TapdRasDialer Component

The TapdRasDialer provides an interface to Microsoft's Remote Access Service (RAS) API. The component is used primarily to establish and terminate a connection to a remote machine using Window's dial-up networking, however it can also be used to manipulate RAS phonebook entries and enumerate active connections.

The TapdRasDialer requires that RAS has been installed on the machine that the application is to be run. If RAS is not installed, then an exception `ecRasLoadFail` is raised whenever a TapdRasDialer function is called.

Dialing is performed by the `Dial` and `DialDlg` (WinNT) methods. Both asynchronous and synchronous dialing options are available with `Dial`. The `Hangup` method terminates the call. Phonebook entries are manipulated by the `CreatePhonebookEntry`, `EditPhonebookEntry`, `ListEntries`, and `PhonebookDlg` (WinNT) methods. The dialing parameters for a particular call can be obtained and by the `GetDialParameters`, and `SetDialParameters` methods. For WinNT machines, a monitor can be displayed by `MonitorDlg` to display the status of RAS connections.

Hierarchy

TComponent (VCL)

- TapdBaseComponent (OOMisc) 8
 - TapdCustomRasDialer (AdRas)
 - TapdRasDialer (AdRas)

Properties

CallBackNumber	DialOptions	PlatformID
CompressionMode	Domain	SpeakerMode
Connection	EntryName	StatusDisplay
ConnectState	HangupOnDestroy	UserName
DeviceName	Password	❶ Version
DeviceType	Phonebook	
DialMode	PhoneNumber	

Methods

CreatePhonebookEntry	GetDialParameters	ListEntries
DeletePhonebookEntry	GetErrorText	MonitorDlg
Dial	GetStatusText	PhonebookDlg
DialDlg	Hangup	SetDialParamters
EditPhonebookEntry	ListConnections	

Events

OnConnected	OnDialStatus
OnDialError	OnDisconnected

Reference

CallBackNumber

property

```
property CallBackNumber : string
```

- ↳ Specifies a callback number for the current phonebook entry.

An empty string indicates that callback should not be used. This string is ignored unless the user has “Set By Caller” callback permission on the RAS server. An asterisk indicates that the number stored in the phonebook should be used for callback.

CompressionMode

property

```
property CompressionMode : TApdRasCompressionMode  
TApdRasCompressionMode = (  
    cmDefault, cmCompressionOn, cmCompressionOff);
```

- ↳ Specifies whether software compression is to be used.

If the application is not running under Windows NT, this property is ignored.

Connection

run-time, read-only property

```
property Connection : TRasConnHandle
```

- ↳ Contains the RAS connection handle for the current phonebook entry.

This property is available if needed, but will probably not be of use to the user.

ConnectState

property

```
property ConnectState : Integer
```

- ↳ Contains connection state for the current phonebook entry.

Use ConnectState to determine the connect status for the current phonebook entry. This value can be passed to GetStatusText to obtain the corresponding text string. Connection state constants are defined in ADRASCS.INC

See also: GetStatusText

CreatePhonebookEntry

method

```
function CreatePhonebookEntry : Integer;
```

↳ Creates a new entry in the current phonebook.

A dialog box is displayed in which the user enters the information for the new entry.

A return value other than ecOK indicates that an error occurred and the return value is the error code. This value can then be passed to GetErrorText to obtain a description of the error.

See also: DeletePhonebookEntry, EditPhonebookEntry, GetErrorText

DeletePhonebookEntry

method

```
function DeletePhonebookEntry : Integer;
```

↳ Deletes the current phonebook entry.

If the EntryName property specifies an existing phonebook entry, that entry will be deleted and the EntryName property set to an empty string. If a connection exists for the entry, it will be disconnected.

A return value other than ecOK indicates that an error occurred and the return value is the error code. This value can then be passed to GetErrorText to obtain a description of the error.

See also: CreatePhonebookEntry, EditPhonebookEntry, EntryName, GetErrorText

DeviceName

run-time, read-only property

```
property DeviceName : string
```

↳ Contains the device name for the current active connection.

DeviceName contains the name of the current device, if available. This could be the name of the modem, the name of the PAD, or the name of a switch device.

See also: ConnectState, DeviceType

```
property DeviceType : string
```

↳ Contains the device type name for the current active connection.

DeviceType contains a string that identifies the type of the current device, if available. Common device types supported by RAS remote access service are “modem”, “pad”, “switch”, “ison”, or “null.”

See also: ConnectState, DeviceName

Dial**method**

```
function Dial : Integer;
```

↳ Establishes a Remote Access Service (RAS) connection.

Use Dial to establish a Remote Access Service (RAS) connection between a RAS client and a RAS server. If a connection error occurs then the connection is automatically hung up.

During asynchronous dialing (DialMode = dmAsync), Dial returns immediately before the connection is established. The connection progress is communicated via the OnDialStatus, OnDialError, and OnConnected events. Additionally, if a TApdRasStatus component is specified by StatusDisplay, then the status component displays a dialing status dialog until the connection has been established or cancelled.

During synchronous dialing (DialMode is set to dmSync), Dial does not return until the connection attempt has completed successfully or failed. No events are fired, so the Dial function result must be checked to determine the connection status. A return value other than ecOK indicates that an error occurred and the return value is the error code. This value can then be passed to GetErrorText to obtain a description of the error.

Connection status information is also available via the ConnectState property until the application calls HangUp to terminate the connection. An application must eventually call HangUp after a connection has been successfully established.

Dial does not display a logon dialog box. This is currently done through the Remote Networking application. Your application is responsible for setting the dialing properties.

See also: ConnectState, DialMode, GetErrorText, Hangup, OnConnected, OnDialError, OnDialStatus, StatusDisplay

```
function DialDlg : Integer;
```

↳ Establishes a RAS connection using synchronous Ras dial dialog (WinNT).

If the application is not running under Windows NT, DialDlg will return a `ecFunctionNotSupported` error.

DialDlg commences synchronous dialing and displays dialog boxes during the connection operation to provide feedback to the user about the progress of the operation. The dialog boxes also provide a Cancel button for the user to terminate the connection.

DialDlg returns when the connection is established, or when the user cancels the operation.

No progress events are fired so the function return value must be examined to determine the status of the connection. A return value other than `ecOK` indicates that an error occurred and the return value is the error code. This value can then be passed to `GetErrorText` to obtain a description of the error. If a connection error occurs, the connection is automatically hung-up.

Connection status information is also available via the `ConnectState` property until the application calls `HangUp` to terminate the connection. An application must eventually call `HangUp` after a connection has been successfully established.

See also: `ConnectState`, `DialMode`, `GetErrorText`, `Hangup`

DialMode**property**

```
property DialMode : TApdRasDialMode
```

```
TApdRasDialMode = (dmSync, dmAsync);
```

Default: `dmAsync`

↳ Specifies whether dialing is performed asynchronously or not.

When asynchronous dialing is specified, a call to the `Dial` method returns immediately and connection status is provided via the `OnDialStatus` and `OnConnected` events. When synchronous dialing is specified, a call to the `Dial` method does not return until either a connection has been established, or an error has been detected during dialing.

See also: `Dial`, `OnConnected`, `OnDialStatus`

DialOptions

property

```
property DialOptions : TApdRasDialOptions  
  
TApdRasDialOption = (doPrefixSuffix, doPausedStates,  
    doDisableConnectedUI, doDisableReconnectUI,  
    doNoUser, doPauseOnScript);  
  
TApdRasDialOptions = set of TApdRasDialOption;
```

↳ Specifies dialing options for the current phonebook entry.

If the application is not running under Windows NT, this property is ignored.

Domain

property

```
property Domain : string
```

↳ Specifies a string containing the domain on which authentication is to occur.

An empty string specifies the domain in which the remote access server is a member. An asterisk specifies the domain stored in the phonebook for the entry.

EditPhonebookEntry

method

```
function EditPhonebookEntry : Integer;
```

↳ Edits the current phonebook entry.

A dialog box is displayed in which the user can modify the information for the entry specified by the EntryName property.

A return value other than ecOK indicates that an error occurred and the return value is the error code. This value can then be passed to GetErrorText to obtain a description of the error.

See also: CreatePhonebookEntry, DeletePhonebookEntry, EntryName, GetErrorText

EntryName

property

```
property EntryName : string
```

↳ Specifies a string containing the phonebook entry to use to establish a connection.

An empty string specifies a simple modem connection on the first available modem port, in which case PhoneNumber must contain the number to be dialed.

GetDialParameters

method

```
function GetDialParameters : Integer;
```

↳ Retrieves stored dialing parameters.

The connection information saved by the last successful call to Dial or SetDialParameters for the current phonebook entry is retrieved.

A return value other than ecOK indicates that an error occurred and the return value is the error code. This value can then be passed to GetErrorText to obtain a description of the error.

See also: GetErrorText, SetDialParameters

GetErrorText

method

```
function GetErrorText(Error : Integer) : string;
```

↳ Returns the text string for the specified RAS error.

Use GetErrorText to obtain the text for a given RAS error.

Example:

```
with ApdRasDialer1 do  
  ShowMessage(GetErrorText(DialDlg));
```

Error code constants are defined in ADRASEC.INC

See also: OnDialError

GetStatusText

method

```
function GetStatusText(State : TRasState) : string;
```

↳ Returns the text string for the specified RAS connection state.

Use GetStatusText to obtain the text for a given RAS connection state.

Example:

```
with ApdRasDialer1 do  
  ShowMessage(GetStatusText(ConnectState));
```

Connection state constants are defined in ADRASCS.INC

See also: ConnectState, OnDialStatus

Hangup

method

```
procedure Hangup;
```

- ↳ Terminates the active RAS connection.

Hangup releases all RASAPI32.DLL resources associated with the connection. Hangup must be called eventually if a connection has been made by calling either the Dial or DialDlg methods. It is safe to call Hangup at any time during dialing.

See also: Dial, DialDlg

HangupOnDestroy

property

```
property HangupOnDestroy : Boolean
```

Default: True

- ↳ Specifies whether or not an active connection is terminated when the component is destroyed.

If an active RAS connection has been established by calling either Dial or DialDlg, a connection handle is maintained for the duration of the call and must be used to hangup the call. By default, when the dialing application terminates, it will disconnect an active connection since the connection handle is not saved. By setting HangupOnDestroy to True, you can override this behavior and allow the connection to remain open when the application closes.

ListConnections

method

```
function ListConnections(List : TStrings) : Integer;
```

- ↳ Obtains a list of all active RAS connections.

Use ListConnections to obtain a string list of all active RAS connections. The List argument must be created before calling the function. Upon return, the List.Strings property contains the entry names of the active connections. If List has not been created, an exception is raised.

Example:

```
...
ComboBox1.Clear;
ApdRasDialer1.ListConnections(ComboBox1.Items);
```

A return value other than ecOK indicates that an error occurred and the return value is the error code. This value can then be passed to GetErrorText to obtain a description of the error.

```
function ListEntries(List : TStrings) : Integer;
```

↳ Obtains a list of all entry names in the current RAS phonebook.

Use `ListEntries` to obtain a string list of all entries in the current RAS phonebook. The `List` argument must be created before calling the function. Upon return, the `List.Strings` property contains the entry names contained in the phonebook. If `List` has not been created, an exception is raised.

Example:

```
...
ComboBox1.Clear;
ApdRasDialer1.ListEntries(ComboBox1.Items);
...
ApdRasDialer1.EntryName :=
    ComboBox1.Items[ComboBox1.ItemIndex];
...
```

A return value other than `ecOK` indicates that an error occurred and the return value is the error code. This value can then be passed to `GetErrorText` to obtain a description of the error.

See also: `GetErrorText`

```
function MonitorDlg : Integer;
```

↳ Displays the status of a connection using the Ras status dialog (WinNT).

Use `MonitorDlg` to display a dialog showing the status of a connection. This is the same dialog box you see if you right-click the Dial-Up Networking Monitor application in NT's system tray.

A return value other than `ecOK` indicates that an error occurred and the return value is the error code. This value can then be passed to `GetErrorText` to obtain a description of the error.

See also: `GetErrorText`

OnConnected

event

```
property OnConnected : TApdRasConnectedEvent
```

```
TApdRasConnectedEvent = procedure(Sender : TObject) of object;
```

↳ Defines an event handler that is called when a RAS connection has been established.

During asynchronous dialing (DialMode = dmAsync), OnConnected is called to notify that the connection has been successfully established. If StatusDisplay specifies a TApdRasStatus dialog, the dialog is closed.

During synchronous dialing (DialMode = dmSync), this event is not used and the ConnectState property should be read to determine if a connection has been established.

See also: ConnectState, Dial, DialDlg, DialMode

OnDialError

event

```
property OnDialError : TApdRasErrorEvent
```

```
TApdRasErrorEvent = procedure(  
    Sender : TObject; Error : Integer) of object;
```

↳ Defines an event handler that is called when an error occurs during dialing.

OnDialError is called whenever an error occurs while attempting to complete a RAS connection. If no event handler is defined, the result of the called method should be used to determine if an error has occurred.

The Error parameter may be passed to GetErrorText to obtain the corresponding error text.

Error code constants are defined in ADRASUTL.PAS

See also: GetErrorText

OnDialStatus

event

```
property OnDialStatus : TApdRasStatusEvent
```

```
TApdRasStatusEvent = procedure(  
    Sender : TObject; State : TRasConnState) of object;
```

↳ Defines an event handler that is called to provide status during dialing.

During asynchronous dialing (DialMode = dmAsync), OnDialStatus is called periodically to provide connection status information. During synchronous dialing (DialMode = dmSync), this event is not used and the ConnectionState property should be read to determine the connection status.

The State parameter may be passed to GetStatusText to obtain the corresponding text string.

Connection state constants are defined in ADRASUTL.PAS

See also: ConnectionState, Dial, DialMode, GetStatusText

OnDisconnected

event

```
property OnDisconnected : TNotifyEvent
```

↳ Defines an event handler that is called when an active connection is terminated.

If an active RAS connection has been established by calling either Dial or DialDlg, the OnDisconnected event is fired when the connection is closed.

See also: ConnectionStatus, OnConnected

Password

property

```
property Password : string
```

↳ Specifies a string containing the user's password.

Password is used to authenticate the user's access to the remote access server.

See also: UserName

Phonebook

property

```
property Phonebook : string
```

↳ Specifies the full path and filename of the phonebook file.

For applications running under Windows NT, Phonebook can be an empty string, and typically is, in which case the default phonebook file RASPHONE.PBK is used.

For applications running under Windows 95/98, Phonebook is ignored. Dial-up networking stores phonebook entries in the registry rather than in a phonebook file.

PhonebookDlg

method

```
function PhonebookDlg : Integer;
```

↳ Displays the main Dial-Up Networking dialog box. (WinNT).

PhonebookDlg displays the main Dial-Up Networking modal dialog box from which the user can edit, or delete a selected phonebook entry, create a new phonebook entry, or specify user preferences.

A return value other than ecOK indicates that an error occurred and the return value is the error code. This value can then be passed to GetLastError to obtain a description of the error.

See also: GetLastError

PhoneNumber

property

```
property PhoneNumber : string
```

↳ Specifies a string containing an overriding phone number.

An empty string indicates that the phonebook entry's phone number should be used. If EntryName contains an empty string, PhoneNumber must contain a phone number to dial.

See also: EntryName

PlatformID

run-time, read-only property

property PlatformID : DWord

Identifies the platform supported by the operating system.

PlatformID can have one of the following values:

Value	Operating System
VER_PLATFORM_WIN32_WINDOWS	Windows 95/98/ME
VER_PLATFORM_WIN32_NT	Windows NT

SetDialParameters

method

function SetDialParameters : Integer;

Updates stored dialing parameters.

This function changes the connection information saved by the last call to Dial or SetDialParameters for the current phonebook entry. Only the UserName, Password, Domain name, and Callback number can be changed.

A return value other than ecOK indicates that an error occurred and the return value is the error code. This value can then be passed to GetErrorText to obtain a description of the error.

See also: GetDialParameters, GetErrorText

SpeakerMode

property

property SpeakerMode : TApdRasSpeakerMode

TApdRasSpeakerMode = (smDefault, smSpeakerOn, smSpeakerOff);

Specifies the modem speaker setting for the current phonebook entry.

For applications not running under Windows NT, this property is ignored.

StatusDisplay

property

```
property StatusDisplay : TApdRasStatus
```

↳ Specifies an attached dialing status dialog.

The StatusDisplay provides a mechanism to display a TApdRasStatus dialog while establishing a connection via the Dial function. If the DialDlg function is used then this property is ignored.

See also: Dial, DialDlg

UserName

property

```
property UserName : string
```

↳ Specifies a string containing the user's identification name.

UserName is used to authenticate the user's access to the remote access server.

See also: Password

TApdRasStatus Component

The `TApdRasStatus` provides a standard RAS dialing status dialog with a Cancel button to abort dialing at any time. To use it, just create an instance and assign it to the `StatusDisplay` property of your `TApdRasDialer` component.

`TApdRasStatus` has no methods that you must call or properties that you must adjust. You might want to change the settings of the `Ctl3D` and `Position` properties to modify the appearance and placement of the window.

Figure 11.1 shows the display that is associated with a `TApdRasStatus` component.

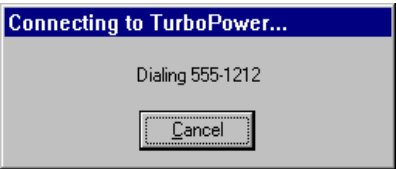


Figure 11.1: `TApdRasStatus` component display.

Hierarchy

`TComponent` (VCL)

`TApdBaseComponent` (OOMisc) 8

`TApdCustomRasStatus` (AdRas)

`TApdRasStatus` (AdRStat)

Chapter 12: TAPI Components

The Telephony Application Programming Interface (TAPI) is a collection of DLLs and a documented programming interface for centralizing and controlling telephony communications services. TAPI was developed by Microsoft primarily for Computer Telephony Integration (CTI) applications. TAPI provides the services that telephone equipment and system providers need to integrate Windows programming and telephone hardware.

TAPI also provides a smaller, though much more visible, service in managing modems as system devices. This is a tremendous boon to communications programmers. No longer do communications applications need to search serial ports for modems, try to identify modems, burden the user with questions about their modem, or try any of the other traditional approaches to supporting modems. Under TAPI, that task is handled by the operating system. Programs make a few simple TAPI calls to determine what modems are available.

Another advantage of TAPI is that applications can share serial ports. For example, assume that an application opens a TAPI device to accept incoming fax or data calls. A second application, if it also uses TAPI, can safely open the same TAPI device for an outgoing call. When the outgoing call is over, TAPI resumes monitoring incoming calls without any further action required by either application.

The TAPI components do have some drawbacks. TAPI may not be installed, or properly configured, on all operating systems. TAPI relies upon several device drivers and INF files to properly configure the modems. If these supporting files are not present, or if they are not adequate, TAPI operations may fail or behave erratically. Consider making a data connection through TAPI; if the wrong modem driver is selected for an installed modem, TAPI could configure the modem to a state where it is incompatible with the remote device. The modem driver may also be written poorly, most notably by not requiring responses to modem commands.

The second major disadvantage of TAPI is that it doesn't provide the level of visibility or control that is provided by TAdModem. TAdModem provides more detailed status information and more control over failures such as "no dialtone" and "called number busy" conditions. It is also much easier to customize TAdModem to provide unique configurations.

The third major disadvantage of TAPI is that TAPI doesn't provide support for direct, modemless connections. TAPI applications usually must still include logic for the direct opening of serial ports.

The fourth major disadvantage of TAPI is the difficulty of assuring proper modem configuration. TAPI modems are detected and installed by Windows during the installation process. If a new modem is added later, the instructions provided with the modem usually direct the user to run one of the Control Panel applets “Add New Hardware” or “Modems.” The “Add New Hardware” applet found in Windows scans all available serial ports for attached modems. When it finds a modem, it sends a variety of commands to the modem and compares the responses against a large database of known modems, usually resulting in an unambiguous choice. The “Modems” applet (or a setup program provided by the modem vendor), can skip this detection process if the modem type is already known.

No matter how the modem is installed, the end result is that TAPI now knows everything it needs to about that modem (serial port, baud rate, and the specific configuration commands/responses). Property sheets are available to the user for changing the configuration of the modem. For example, the user can turn the speaker on or off, change the attached serial port, enable/disable flow control, and so on.

The default property values for the modem chosen by Microsoft or the modem vendor are the values that provide the best results in the widest variety of situations. These properties, however, are available to the user via the Modem applet’s modem property sheets. If the user changes a critical value (say, the serial port number or perhaps flow control) it’s likely that your application won’t operate properly when using that modem. Unfortunately, there isn’t much you can do to protect against this. The responsibility for assuring the modem is properly configured is in the user’s hands, not the application.

The final major disadvantage of TAPI is that resources often do not get properly released following an abnormal termination of an application using TAPI. To counter this problem, TAPdTapiDevice includes a TAPI crash recovery mechanism which will automatically detect this situation and release TAPI resources.

All of these issues are likely to improve over time, making TAPI the choice for current and future communications programs. The best recommendation at this point is to use TAPI for applications targeted for Windows.

Async Professional provides components that make use of TAPI modem management. These components provide an alternate to TAdModem for configuring modems and dialing and answering calls. Async Professional negotiates for TAPI 1.4 in Windows applications.

Async Professional also supports TAPI 1.3 functions in 16-bit applications for Windows 3.1 and Windows for Workgroups, but those environments lack a general purpose modem service provider, which renders most of the TAPI functions useless for serial communications programs.

The Async Professional TAPI components provide all the services necessary for selecting TAPI devices, dialing and answering calls, and receiving status information about TAPI calls in progress. The TAPI system itself provides even more services. For more information about TAPI, see the following sources:

- Sells, *Windows Telephony Programming: A Developer's Guide to TAPI*, Addison-Wesley, ISBN 0/201/63450-3.
- *TAPI Reference Manual*, included with the TAPI SDK and with the Windows 95/98 SDK.
- “Create Communications Programs for Windows 95 with the Win32 Comm API”, Microsoft Systems Journal, 1994 #12 (December).
- *Programming Windows 95 Unleashed* (SAMS Publishing). Although only one out of 37 chapters is devoted to TAPI, it provides a nicely condensed version of much of the information in the TAPI Reference Manual, plus some helpful C++ example programs.
- The C++ sample program TAPICOMM, available on the Microsoft Developer Network CD.

TAPI Device Control from an Application

Without TAPI, the `TApdComPort` component opens the physical serial port directly using the appropriate Windows API call, which returns a handle to that port. The `TApdComPort` then uses the handle to send and receive data and otherwise control the serial port.

Configuring, dialing or answering the modem requires sending explicit `ATXxx` commands to the modem, interpreting the responses, and dealing with the myriad of differences among currently available modems.

With TAPI, a `TApdComPort` is still required, but it is not initially involved in establishing the modem connection—a `TApdTapiDevice` is used for that purpose. The application calls `Dial` to place an outgoing call or calls `AutoAnswer` to wait for an incoming call. TAPI sends the appropriate `ATXxx` commands, interprets the responses, and establishes the modem connection.

Once the connection is established, TAPI's role is essentially over. TAPI remains in charge of the call until the modem connection is broken, but the `TApdTapiDevice` is not used again. From this point on you use the `TApdComPort` to control the port and send/receive data, just as if TAPI is not involved. The `TApdComPort` is automatically opened by the `TApdTapiDevice` when the modem connection is established. Several `TApdComPort` properties are updated with appropriate information from the TAPI device, notably `Baud` and `ComNumber`.

When the modem connection is broken, `TApdTapiDevice` automatically closes the associated `TApdComPort`. The `TApdComPort` cannot be used for input/output unless the modem connection is re-established by the TAPI device or unless the program bypasses TAPI and opens and uses the `TApdComPort` directly.

The `TApdComPort` property `TapiMode` determines whether the port is in charge of the physical serial port or whether TAPI is in charge of the port:

```
TapiMode : TTapiMode;  
TTapiMode = (tmNone, tmAuto, tmOn, tmOff);
```

When `TapiMode` is `tmAuto` (the default), the `TApdComPort` is in charge unless a `TApdTapiDevice` is added to the form. The `TApdTapiDevice` automatically forces the `TapiMode` property to `tmOn` and thereafter the `TApdTapiDevice` is in charge. Attempts to open the `TApdComPort` directly (by setting its `Open` property to `True`) are ignored.

You can give control back to the `TApdComPort` by setting `TapiMode` to `tmOff`, meaning that the port is not to be used in TAPI mode even if a `TApdTapiDevice` is present. To give control back to the `TApdTapiDevice` again, set `TapiMode` to `tmAuto` or `tmOn`.

TAPI events

TAPI dials outgoing calls and waits for incoming calls in the background. Applications are informed of progress through a callback procedure. TAPdTapiDevice installs a hidden callback and translates these progress callbacks into the following VCL events:

OnTapiStatus

```
procedure(  
    CP : TObject; First, Last : Boolean; Device, Message,  
    Param1, Param2, Param3 : DWORD) of object;
```

Generated at various intervals while dialing an outgoing call or answering an incoming call. The parameters mirror the parameters passed directly to the TAPI callback. You will usually need to reference only the Message and Param1 fields. The other fields are supplied for applications that extend the services provided by TAPdTapiDevice. See the OnTapiStatus event on page 425. Also see “TAPI status processing” on page 394.

OnTapiLog

```
procedure(CP : TObject; Log : TTapiLogCode) of object;
```

Generated at the start and finish of each TAPI call (either outgoing or incoming) and at various points during the call. See the OnTapiLog event on page 424. Also see “TAPI logging” on page 397.

OnTapiPortOpen

```
procedure(CP : TObject) of object;
```

Generated immediately after TAPI has established a connection and has the serial port handle available for handoff to the TAPdComPort component. See the OnTapiPortOpen event on page 425.

OnTapiPortClose

```
procedure(CP : TObject) of object;
```

Generated immediately after TAPI closes the serial port due to a broken connection. See the OnTapiPortClose event on page 424.

OnTapiConnect

```
procedure(CP : TObject) of object;
```

Generated immediately after TAPI establishes a connection. See the OnTapiConnect event on page 393.

OnTapiFail

```
procedure(CP : TObject) of object;
```

Generated immediately after TAPI tries but fails to establish a connection. See the OnTapiFail event on page 394.

TAPI status processing

TAPI handles the details of controlling, dialing, and answering the modem. The OnTapiStatus event is provided to let you know what's happening as the connection progresses.

TAPI actually uses a callback procedure to inform an application program of its progress. TApdTapiDevice installs a hidden callback and translates all calls into OnTapiStatus events for easier processing. The format of the event handler is identical to the internal TAPI callback. The following illustrates the format and use of OnTapiStatus:

```
...
ApdTapiDevice1: TApdTapiDevice;
Msg: TLabel;
Num: TLabel;
...
procedure ApdTapiDevice1TapiStatus(
  CP : TObject; First, Last : Boolean; Device, Message, Param1,
  Param2, Param3 : LongInt);
end;

procedure TForm1.ApdTapiDevice1TapiStatus(
  CP : TObject; First, Last : Boolean; Device, Message,
  Param1, Param2, Param3 : LongInt);
begin
  if First then
    ...do setup stuff
  else if Last then
    ...do cleanup stuff
  else begin
    {Update status}
    Msg.Caption := ApdTapiDevice1.TapiStatusMsg(Message, Param2);
    Num.Caption := ApdTapiDevice1.Number;
  end;
end;
```

ApdTapiDevice1TapiStatus handles the OnTapiStatus event by updating a form at each call. First is True for the first status event of the current call. Last is True for the last status event of the current call. These parameters can be used to setup and cleanup forms and other resources used when displaying status information.

The remaining parameters mirror the parameters that TAPI sends to the hidden callback procedure. Only Message and Param1 are used by TapdTapiDevice. The other parameters are provided in case you extend TapdTapiDevice beyond making and answering calls.

Async Professional contains resource strings for all of the values of Message and Param1 that TapdTapiDevice can generate. Most applications can simply pass those parameters to TAPIStatusMsg to return the appropriate string, as shown in the code above. If returning an appropriate string is sufficient for your purposes, you can skip the following discussion of the parameters. Only Message and Param1 are described in detail; the rest are mentioned only briefly.

Message is a constant that describes the class of change since the previous OnTapiStatus event. The possible values are shown in Table 12.1.

Table 12.1: *Possible TAPI messages*

TAPI Message	Value	Explanation
Line_CallState	2	The state of the call changed.
Line_LineDevState	8	The device state changed.
Line_Reply	12	The previous request was accepted.
Line_APDSpecific	32	Async Professional-specific status.

The first three values are a subset of the possible TAPI messages. These are the only values that dialing and answering generate. The final value, Line_APDSpecific isn't really a TAPI status message. It's a pseudo state change that TapdTapiDevice generates to provide more information about the progress of a call.

Line_CallState indicates that the progress of the call, what TAPI calls the "state" of the call, changed. For example, the line was idle, but now it is dialing or the line was dialing but now it is proceeding (the TAPI term for waiting for the connection).

Line_LineDevState indicates that the state of the device (modem, phone, or whatever) changed. TapdTapiDevice dial/answer actions generate this message only to indicate that the modem is ringing.

Line_Reply indicates that TAPI has accepted, but not necessarily completed, the requested background. For example, it is generated just after a request to dial a number.

Line_APDSpecific is generated during periods when TAPI does not generate events, such as after dialing and waiting for a connection, or when answering and waiting for a connection. The primary purpose of Line_APDSpecific is to give the status event an opportunity to update a timer.

Param1 provides additional information about Message. For example, when Message is Line_CallState, Param1 contains a constant describing the change in state. Table 12.2 shows the values of Param1 that are generated by TApdTapiDevice for each value of Message.

Table 12.2: *Corresponding Param1 values to Message values*

Param1 Value	Explanation
For Message = Line_CallState	
LineCallState_Idle	No call in progress.
LineCallState_Offering	Call starting.
LineCallState_Accepted	Incoming call accepted.
LineCallState_DialTone	Dialtone detected.
LineCallState_Dialing	Dialing the outgoing number.
LineCallState_Proceeding	Handshaking with the remote modem.
LineCallState_RingBack	Detected a remote ring.
LineCallState_Busy	Detected a busy signal.
LineCallState_Connected	Connected with the remote modem.
LineCallState_Disconnected	Disconnected from remote modem.
For Message = Line_DevState	
LineDevState_Ringing	Ring detected for incoming call.
For Message = Line_Reply	
(Param1 not used for Line_Reply)	
For Message = Line_APDSpecific	
APDSpecific_DialFail	Dial failed due to busy or other error.
APDSpecific_RetryWait	Waiting for next retry.
APDSpecific_TAPIChange	Unknown state change.

These are only subsets of the possible values of Param1 for each of the Message states, but they are the only values generated for the dial/answer actions performed by TApdTapiDevice.

Other `TApdTapiDevice` properties can also be used in `OnTapiStatus`. Some examples are `Number`, which contains the number just dialed, and `SelectedDevice`, which contains the name of the TAPI device.

Automatic status display

Async Professional includes a mechanism for providing automatic TAPI status display without programming, through the `StatusDisplay` property of `TApdTapiDevice`:

```
property StatusDisplay : TApdAbstractTapiStatus
```

The `TApdAbstractTapiStatus` class is described in more detail on page 440. For each `OnTapiStatus` event, `TApdTapiDevice` checks whether `StatusDisplay` is assigned. If it is, the `UpdateDisplay` method of `StatusDisplay` is called to update the display. `TApdTapiDevice` then calls the `OnTapiStatus` event, if one is implemented.

When a `TApdTapiDevice` component is created, either dynamically or when dropped on a form, it searches the form for a `TApdAbstractTapiStatus` instance and updates the `StatusDisplay` property with the first one it finds. `StatusDisplay` is also filled in if a `TApdAbstractTapiStatus` component is added to the form later. You can also change `StatusDisplay` at design time or run time to point to a different `TApdAbstractStatusDisplay` component.

TAPI logging

Dialing and answering calls is often an automated process. For example, an application might automatically dial a list of numbers every night to upload or download the day's transaction files. Or, an application might serve a BBS-like role where customers dial in to get the latest updated files or information.

The `TApdTapiDevice` provides an event that is ideal for logging automated dial or answer applications. The `OnTapiLog` event provides an opportunity to log information about each call. The information that is logged includes the time the call starts, the time it ends, and additional information about events during the call (connected, busy/retry, and so on). This example that handles the `OnTapiLog` event:

```
procedure TForm1.ApdTapiDevice1TapiLog(
  CP : TObject; Log : TTapiLogCode);
var
  HisFile : Text;
begin
  ...open HisFile
  {Write the log entry}
  with TapiDevice do begin
    case Log of
      ltapiNone : ;
      ltapiCallStart :
        WriteLn(HisFile, DateTimeToStr(Now), ' : call started');
      ltapiCallFinish :
        WriteLn(HisFile, DateTimeToStr(Now), ' :
          call finished'^M^J);
      ltapiDial :
        WriteLn(
          HisFile, DateTimeToStr(Now), ' : dialing ', Number);
      ltapiAnswer :
        WriteLn(HisFile, DateTimeToStr(Now), ' : answering');
      ltapiConnect :
        WriteLn(HisFile, DateTimeToStr(Now), ' : connected');
      ltapiCancel :
        WriteLn(HisFile, DateTimeToStr(Now), ' : cancelled');
      ltapiDrop :
        WriteLn(HisFile, DateTimeToStr(Now), ' : dropped');
      ltapiBusy:
        WriteLn(HisFile, DateTimeToStr(Now), ' : busy');
      ltapiDialFail :
        WriteLn(HisFile, DateTimeToStr(Now), ' : dial failed');
    end;
  end;
  ...close HisFile
end;
```

This example shows every possible logging value. The meanings of the various logging conditions are described in Table 12.3.

Table 12.3: *TAPI logging conditions*

Logging Condition	Explanation
ltapiAccept	Accepting an incoming call.
ltapiAnswer	An incoming ring was detected and the call is being answered.
ltapiBusy	The called number was busy.
ltapiCallFinish	The call is finished, either successfully or due to an error.
ltapiCallStart	A call, either answer or dial, was started.
ltapiCancel	The dial or answer operation was cancelled before a connection was established.
ltapiConnect	A successful modem connection was established after a dial or answer.
ltapiDial	An outgoing call was just dialed. The Number property contains the number that was dialed.
ltapiDialFail	A modem connection was not established due to no dialtone, no answer, or some other error.
ltapiDrop	A connection was established but was subsequently dropped. The drop can be due to an error or due to the normal completion of the session.
ltapiReceivedDigit	A DTMF digit has been received.

A call is always started with an ltapiCallStart event and finished with an ltapiCallFinish event. There may be one or more other events in between. A typical dial operation might generate the following sequence of log events:

ltapiCallStart	call started
ltapiDial	number dialed
ltapiBusy	called number was busy
ltapiDial	number re-dialed
ltapiConnect	a connection was established
ltapiDrop	the connection was dropped
ltapiCallFinish	the call is finished

Automatic TAPI logging

Async Professional includes a mechanism for providing automatic TAPI logging without programming, through the `TapiLog` property of the `TApdTapiDevice`:

```
property TapiLog : TApdTapiLog
```

The `TApdTapiLog` class is described in more detail on page 445. For each `OnTapiLog` event, `TApdTapiDevice` checks whether `TapiLog` is assigned. If it is, the `UpdateLog` method of `TapiLog` is called to update the log. `TApdTapiDevice` then calls the `OnTapiLog` event, if one is implemented.

When a `TApdTapiDevice` component is created, either dynamically or when dropped on a form, it searches the form for a `TApdTapiLog` instance and updates the `TapiLog` property with the first one it finds. `TapiLog` is also filled in if a `TApdTapiLog` component is added to the form later. You can also change `TapiLog` at design time or run time to point to a different `TApdTapiLog` component.

Making calls

`TApdTapiDevice` provides a method for placing outgoing calls. When `Dial` is called, TAPI sends the appropriate modem configuration commands to the modem, then dials the number passed to the `Dial` method.

The number passed to `Dial` should not contain any modem commands. It should contain the telephone number to dial, exactly as it would be dialed from a telephone handset.

`TApdTapiDevice` generates `OnTapiStatus` events during the dialing process. If a connection is not established, an `OnTapiFail` event is generated. If a connection is established, an `OnTapiConnect` event is generated, the associated `TApdComPort` is opened, and the `OnTapiPortOpen` event is generated.

Once the connection is established, the `TApdTapiDevice` is no longer directly used. All subsequent port control and input/output operations use the `TApdComPort`. The exception to this occurs when the call is terminated. The application should not simply close the `TApdComPort` because that would not disconnect the modem connection. Instead, the application must direct TAPI to close the connection by calling the `CancelCall` method of `TApdTapiDevice`. The `TApdTapiDevice` then breaks the connection, closes the associated `TApdComPort`, and generates the `OnTapiPortClose` event.

If a dial attempt fails due to a busy signal or other error, `TApdTapiDevice` can try the call again. This is controlled by the `MaxAttempts` property, which determines how many times `Dial` tries the call, and `RetryWait`, which determines how long (in seconds) `Dial` waits before retrying a failed call.

Dialing example

This example demonstrates how to construct and use the TApdTapiDevice to dial a number. This example includes a terminal window and emulator so that you can dial and log on to a BBS or information service.

Create a new project, add the following components, and set the property values as indicated in Table 12.4.

Table 12.4: *Example components and property values*

Component	Property
TApdComPort	
TAdEmulator	
TAdTerminal	
TApdTapiDevice	SelectedDevice
TApdTapiStatus	
TApdTapiLog	
TButton	Name

Double-click on the Dial button's OnClick event handler within the Object Inspector and modify the generated method to match this:

```
procedure TForm1.DialClick(Sender : TObject);
begin
    ApdTapiDevice1.Dial('1-847-262-6000');
end;
```

The phone number passed to Dial is the number of the U.S. Robotics BBS. Modify it if you want to dial a different BBS or service. This method tells TAPI to initialize the modem and dial the number.

Next, double-click on the Hangup button's OnClick event handler within the Object Inspector and modify the generated method to match this:

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

This method cancels the dial operation, or hangs up the phone after the connection is established.

Finally, double-click on the `OnTapiPortOpen` event handler within the Object Inspector and modify the generated method to match this:

```
procedure TForm1.ApdTapiDevice1TapiPortOpen(Sender : TObject);  
begin  
    ApdTerminal1.SetFocus;  
end;
```

This event handler gives the focus to the terminal window as soon as TAPI establishes a connection with the BBS.

This example is in the EXTAPID project in the `\ASYNCPRO\EXAMPLES` directory.

Answering calls

The process of answering the modem is very similar to dialing. The `TApdTapiDevice` component generates the same events as it does when dialing.

The `TApdTapiDevice` waits for incoming calls in the background. No events are generated while waiting for calls. When an incoming call is detected, `TApdTapiDevice` begins generating `OnTapiStatus` events at regular intervals. If a connection is not established, an `OnTapiFail` event is generated. If a connection is established, an `OnTapiConnect` event is generated, the associated `TApdComPort` is opened, and the `OnTapiPortOpen` event is generated.

Once the connection is established, the `TApdTapiDevice` is no longer directly used. All subsequent port control and input/output operations use the `TApdComPort`. The exception to this occurs when the call is terminated. The application should not simply close the `TApdComPort` because that would not disconnect the modem connection. Instead, the application must direct TAPI to close the connection by calling the `CancelCall` method of `TApdTapiDevice`. The `TApdTapiDevice` then breaks the connection, closes the associated `TApdComPort`, and generates the `OnTapiPortClose` event.

Answering example

The following example demonstrates how to construct and use the `TApdTapiDevice` to answer an incoming call.

Create a new project, add the following components, and set the property values as indicated in the Table 12.5.

Table 12.5: *Example components and property values*

Component	Property	Value
TApdComPort		
TAdEmulator		
TAdTerminal		
TApdTapiDevice	SelectedDevice	<set as needed for your PC>
TApdTapiStatus		
TApdTapiLog		
TButton	Name	Answer
TButton	Name	Hangup

Double-click on the Answer button's OnClick event handler within the Object Inspector and modify the generated method to match this:

```
procedure TForm1.AnswerClick(Sender : TObject);
begin
    ApdTapiDevice1.AutoAnswer;
end;
```

AutoAnswer instructs TAPI to listen for incoming calls. It does not immediately begin answering an incoming call. In fact, if an incoming call is ringing before you call AutoAnswer, AutoAnswer will most likely not pick up that call. This happens because TAPI alerts applications of an incoming call on the first ring of that call. If no applications are listening at that point, TAPI does not attempt to answer the call.

Next, double-click on the Hangup button's OnClick event handler within the Object Inspector and modify the generated method to match this:

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

This method tells TAPI to stop listening for incoming calls. If a call is in the process of being answered, it is aborted immediately. If a connection was already established, it is disconnected.

Finally, double-click on the OnTapiPortOpen event handler within the Object Inspector and modify the generated method to match this:

```
procedure TForm1.ApdTapiDevice1TapiPortOpen(Sender : TObject);  
begin  
    ApdTerminal1.SetFocus;  
end;
```

This event handler gives the focus to the terminal window as soon as TAPI establishes a connection.

This example is in the EXTAPIA project in the \ASYNCPRO\EXAMPLES directory.

TAPI Service Providers

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

Even if TAPI is properly installed, it will not function unless a service provider is also installed. TSP modules are typically provided by telephony vendors along with their telephony hardware. Windows 95/98 and Windows NT 4.0 install a general-purpose service provider named UNIMDM.TSP, which provides basic dial and answer support for modems. It is this service provider that makes TAPI available to communications programs in Windows.

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

- UNIMDM does not provide support for caller identification (caller ID). The CallerID property of TApdTapiDevice always returns an empty string when using UNIMDM.
- UNIMDM does not support “no dialtone” detection. TAPI will attempt to dial whether a dialtone is detected or not.

Microsoft has released an extension for Unimodem called UNIMODEM/V. UNIMODEM/V provides additional TAPI services (including support for caller ID and “no dialtone” detection). As of this writing UNIMODEM/V is only for Windows 95/98. UNIMODEM/V for Windows NT 4.0 is not available yet. See the README.TXT file for updated news on UNIMDM supported services. You can also view the UNIMODEM/V README.TXT file at <http://support.microsoft.com/support/kb/articles/q140/3/23.asp>.

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

Using TAPI for configuration only

Although UNIMDM.TSP provides basic dial and answer services it does not provide all of the modem services an application might need. UNIMDM.TSP cannot be used to place a faxmodem in fax answer mode or in adaptive answer mode (in which both incoming fax and incoming data calls are accepted).

However, TAPI (along with UNIMDM and modem information files) contains a wealth of configuration information and it is worthwhile to use TAPI to configure the modem and control the call, even if TAPI doesn't dial or answer the modem. This is provided by a feature called "passthrough mode." In passthrough mode, TAPI immediately enters the "connected" state and opens the associated serial port.

Although in passthrough mode, TAPI doesn't send any modem initialization commands, TApdTapiDevice uses a two-step process to enter passthrough mode, which forces TAPI to send its modem initialization commands. When the ConfigAndOpen method is called, TApdTapiDevice first initializes TAPI in answer mode, which forces TAPI to send its initialization commands. TApdTapiDevice then immediately closes the port and reopens it in passthrough mode.

After calling ConfigAndOpen, TAPI is in control of the call, just as though it had dialed or answered the modem. No OnTapiStatus or OnTapiLog events are generated, but OnTapiPortOpen is generated. To close the call, use CancelCall. If TAPI ever aborts or closes the call itself, the TApdTapiDevice generates the OnTapiPortClose event.

You should use TAPI passthrough mode if you need to support TAPI, but require modem operations that UNIMDM.TSP doesn't provide. An example of this arises in Async Professional when a TAPI-based program needs to send or receive faxes. See EXTAPIF for an example of a TAPI-based fax program.

Wave file support

The TApdTapiDevice class now includes the ability to play and record wave files through a TAPI device (over the phone line). This feature, along with the new DTMF feature, allows you to create an automated voice answering system with Async Professional. To play and record wave files through the TAPI device, you must have the following:

- Windows 95/98/ME or Windows 2000.
- UNIMODEM/V or UNIMODEM/5.
- A voice modem with a wave driver.
- A wave file.

UNIMODEM/V is a set of DLLs that provides voice support for voice modems under Windows 95/98. Voice support includes DTMF tone detection and generation, and wave file playback and recording. UNIMODEM/V is currently available only for Windows 95/98/ME. You can get UNIMODEM/V for Windows 95 from the Microsoft web site.

To use the voice extensions provided by UNIMODEM/V, you must have a voice modem. For wave support, it is important that you have the wave driver for the modem installed. Consult your modem documentation to install the wave device properly.

The `TapdTapiDevice` component allows you to set the wave file format used for playback and recording. The default wave format is PCM, 8KHz, 16 bit, mono. This format was chosen because it is supported by the majority of voice modems. Some voice modems support other wave file formats.

Wave files used for playback with Async Professional can be created with the Microsoft Sound Recorder program. Wave files for use with TAPI which will be played over general telephone lines (POTS) must be recorded in a PCM format compatible with your voice modem (here again, the attributes 8,000 Hz (8Khz), 16 Bit, mono are a good bet). Sound Recorder also allows for the conversion of existing wave files. TAPI requires a call handle for recording. For this reason, you cannot use the `TapdTapiDevice` for recording messages unless you call your modem from another phone.

Recording options include the ability to detect silence on the line and take action when silence is detected (such as hanging up the call). This is desirable because TAPI does not have the ability to detect a hangup for a voice call. This option allows you to save disk space by saving only the portion of the call which contains data.

Dual Tone Multiple Frequency (DTMF)

Dual Tone Multiple Frequency (DTMF) tones are generated by a telephone touch pad over telephone lines. With compatible drivers and modems, Async Professional can detect (receive) and generate these tones. Async Professional notifies an application when it receives a tone by generating an `OnTapiDTMF` event. Tones are generated using the `SendTone` method. See “TAPI Voice Support” in the Developer's Guide for information about supported operating systems.

TApdTapiDevice Component

TApdTapiDevice provides modem dialing, answering and configuration services using Windows built-in TAPI support.

Hierarchy

TComponent (VCL)

- TApdBaseComponent (OOMisc) 8
 - TApdCustomTapiDevice (AdTapi)
 - TApdTapiDevice (AdTapi)

Properties

AnswerOnRing	EnableVoice	ShowTapiDevices
ApiVersion	FailureCode	StatusDisplay
Attempt	FilterUnsupportedDevice	TapiLog
AvgWaveInAmplitude	FilterUnsupportedDevices	TapiState
BPSRate	InterruptWave	TrimSeconds
CallerID	MaxAttempts	SilenceThreshold
CallerIDName	MaxMessageLength	UseSoundCard
ComPort	Number	● Version
Cancelled	RetryWait	WaveFileName
DeviceCount	SelectedDevice	WaveState
Dialing	ShowPorts	

Methods

AutoAnswer	GetDevConfig	ShowConfigDialog
AutomatedVoiceToComms	PlayWaveFile	ShowConfigDialogEdit
CancelCall	SaveWaveFile	StartWaveRecord
ConfigAndOpen	SelectDevice	StopWaveFile
CopyCallInfo	SendTone	StopWaveRecord
Dial	SetDevConfig	TapiStatusMsg
FailureCodeMsg	SetRecordingParams	TranslateAddress

Events

OnTapiCallerID	OnTapiLog	OnTapiWaveNotify
OnTapiConnect	OnTapiPortClose	OnTapiWaveSilence
OnTapiDTMF	OnTapiPortOpen	
OnTapiFail	OnTapiStatus	

Reference Section

AnswerOnRing

property

property AnswerOnRing : Byte

Default: 2

- ↳ The number of times the TAPI device should allow the incoming call to ring before answering it.

The default for AnswerOnRing is two rings because problems can occur with caller-ID enabled modems if the call is answered before the first ring.

ApiVersion

read-only, run-time property

property ApiVersion : LongInt

- ↳ Returns the negotiated TAPI version level.

When an application initializes TAPI, it negotiates for a supported version of TAPI. Features and behaviors differ among the versions. An application requests the highest level of TAPI for which it was designed. The built-in TAPI services, even if they support a higher release level, will behave as the requested version behaves.

The TApdTapiDevice always attempts to negotiate for version 1.4, but can use 1.3 if that is all that is available. With operating systems supporting TAPI 2.0 or greater, TApdTapiDevice will continue to operate at the 1.4 release level. See the README.TXT file for TAPI version information.

Attempt

read-only, run-time property

property Attempt : Word

- ↳ Indicates the number of times the current number has been dialed.

If the dialed number is busy, TAPI waits briefly and calls the number again. It tries up to MaxAttempts times. The Attempt property returns the number of the current attempt. Attempt is incremented immediately upon encountering a busy line. Attempt is primarily for use in OnTapiStatus event handlers.

See also: MaxAttempts, OnTapiStatus, RetryWait

```
property AvgWaveInAmplitude : Integer
```

↳ Indicates the average relative amplitude of a recorded wave sample.

When a wave file is being recorded the AvgWaveInAmplitude property is updated to reflect the average wave input amplitude of the recording. This value can be used to determine spikes in the wave amplitude, which could mean the remote phone's ringback, someone saying "hello", or it could just be an indicator of the speaker's volume. When loud noises are recorded AvgWaveInAmplitude will be larger than when silence is recorded. This property is updated approximately every second while recording.

This property can also be used to determine a usable value for the SilenceThreshold property. SilenceThreshold is the amplitude of recorded wave data to consider silence, and can vary quite a bit between phone lines and modems. To attain a usable SilenceThreshold value, record a few seconds of silence on the line. The AvgWaveInAmplitude property will contain the value for SilenceThreshold for this modem/line combination.

This property is valid while a wave file is being recorded, which is after the StartWaveRecord method has been called, up to the termination of the recording.

See also: OnTapiWaveSilence, SilenceThreshold, StartWaveRecord, StopWaveRecord, TrimSeconds

AutoAnswer**method**

```
procedure AutoAnswer;
```

↳ Instructs TAPI to listen for and answer incoming calls.

AutoAnswer returns immediately after instructing TAPI to listen for calls. TAPI listens for calls in the background. When an incoming call is detected, it answers the call, generating appropriate events as it does.

See "Answering calls" on page 402 for more information and an example.

```
procedure AutomatedVoiceToComms;
```

- ↪ AutomatedVoiceToComms changes the TAPI media mode from voice to data communications.

Call AutomatedVoiceToComms when switching from an automated voice mode (DTMF tone detection, wave recording/playing) to a fax send/receive mode.

The OnTapiPortOpen event will be generated once TAPI successfully switches media modes (from AutomatedVoice to DataModem). Note that Windows 2000 does not allow switching media modes on an active call, so this method will fail under this operating system.

See also: OnTapiPortOpen

BPSRate**read-only, run-time property**

```
property BPSRate : DWORD
```

- ↪ The rate of the current call in bits per second.

BPSRate is the rate negotiated between the local and remote modems for the current call. If a call is not in progress, BPSRate returns zero.

The following example shows an OnTapiConnect event handler that updates a TLabel on the current form with the negotiated bps rate:

```
TForm1 = class(TForm)
...
  ApdTapiDevice1 : TApdTapiDevice;
  Connect        : TLabel;
...
end;

procedure TForm1.ApdTapiDevice1TapiConnect(Sender : TObject);
begin
  Connect.Caption :=
    'Connected at ' + IntToStr(ApdTapiDevice1.BPSRate);
...
end;
```

```
property CallerID : string
```

↳ Contains the caller identification string of the current incoming call.

Many telephony environments make a caller identification string available. This string usually contains the phone number of the incoming call, but can contain other information as well, if supplemented by an office telephony system.

If the telephony environment doesn't supply caller identification information, CallerID is an empty string.

Note: Caller ID requires voice capabilities (i.e., UnimodemV, voice modem).

The following example shows an OnTapiConnect event handler that updates a TLabel on the current form with the caller ID information:

```
TForm1 = class(TForm)
...
  ApdTapiDevice1 : TApdTapiDevice;
  CallerID       : TLabel;
...
end;

procedure TForm1.ApdTapiDevice1TapiConnect(Sender : TObject);
begin
  CallerID.Caption := 'Caller: ' + ApdTapiDevice1.CallerID;
...
end;
```

```
property CallerIDName : string
```

↳ Contains the caller identification name information, if available.

The name information is not available everywhere that caller ID is available. Some localities only provide the caller ID phone number.

See also: CallerID

```
procedure CancelCall;
```

✚ Disconnects the current call.

CancelCall is the TApdTapiDevice universal method for terminating the current call. It can be used while waiting for incoming call, answering an incoming call, dialing a call, or during an established connection. TAPI aborts the current process, assures that the modems have disconnected, and places the modem into an idle state (not waiting for calls).

CancelCall is usually a background operation. It instructs TAPI to cancel the call and TAPI performs the work of canceling and cleaning up. OnTapiPortClose will fire when TAPI closes the port (assuming the port was open). OnTapiFail will always be generated when CancelCall completes. To determine whether the OnTapiFail was generated due to CancelCall, use the Cancelled property.

See also: Cancelled, OnTapiFail, OnTapiPortClose

Cancelled**run-time, read-only property**

```
property Cancelled : Boolean
```

✚ Returns True if an OnTapiFail event fires due to user action.

An OnTapiFail event is generated any time a call is terminated before the final connection is made—even if CancelCall is used to terminate the call. If CancelCall was used to terminate the connection, Cancelled will be True.

See also: CancelCall, OnTapiFail

ComPort**property**

```
property ComPort : TApdCustomComPort
```

✚ Determines the TApdComPort component used by the TApdTapiDevice.

A properly initialized TApdComPort must be assigned to this property before dialing or answering calls.

ComPort is usually set automatically at design time to the first TApdComPort component the TApdTapiDevice finds on the form. If needed, use the Object Inspector to select a different TApdComPort component.

Setting the ComPort property at run time is necessary only when using a dynamically created TApdComPort or when selecting among several TApdComPort components.


```
procedure ConfigAndOpen;
```

↳ Configures the modem and leaves the port open in passthrough mode.

ConfigAndOpen takes advantage of the TAPI modem configuration facilities, even though TAPI isn't used for dialing or answering a call. ConfigAndOpen does, however, require a short period of background processing before the associated TApdComPort component is open.

You could not, for example, use the following logic:

```
var
  ApdComPort1    : TApdComPort;
  ApdTapiDevice  : TApdTapiDevice;
...
  ApdTapiDevice1.ConfigAndOpen;
  ApdComPort1.Output := 'ready';
  ...more port I/O
  ApdComPort1.Open := False;
```

This is incorrect. After the call to ConfigAndOpen you must wait for TAPI to open the TApdComPort, just as when TAPI is dialing or answering a call. When TAPI is ready, (usually just a second or two) it generates the OnTapiPortOpen event which automatically opens the associated TApdComPort.

The following example shows the proper way to use ConfigAndOpen:

```
procedure TForm1.OpenTheLine;
begin
  ApdTapiDevice1.ConfigAndOpen;
end;
...
procedure TForm1.ApdTapiDevice1TapiPortOpen(Sender : TObject);
begin
  ApdComPort1.Output := 'ready';
  ...more port I/O
end;
...
procedure TForm1.CloseTheLine;
begin
  ApdTapiDevice1.CancelCall;
end;
```

See “Using TAPI for configuration only” on page 405 for more information.

```
procedure CopyCallInfo(var CallInfo : PCallInfo);
```



Returns a record containing details about the current call.

The `TApdTapiDevice` contains properties for the basic call information that a typical “dialer” program needs about the current call. TAPI, however, maintains much more information about the call. Although this information is of limited use, `CopyCallInfo` provides it for applications that might need it.

`CallInfo` is a pointer to a `PCallInfo` structure, which is allocated by `CopyCallInfo` and filled in by TAPI. The calling program is responsible for deallocating `CallInfo` when it is no longer needed. The size of the allocated structure varies depending on the information TAPI has about the current call.

The fields in the `TCallInfo` record are not described here. See page 393 for a list of TAPI references.

DeviceCount**read-only, run-time property**

```
property DeviceCount : LongInt
```



The number of currently installed TAPI devices.

`DeviceCount` is the number of TAPI devices installed on the machine and available for use by TAPI applications. Typically, this is the same as the number of modems installed on the machine (usually one).

See also: `FilterUnsupportedDevices`, `SelectedDevice`

Dial**method**

```
procedure Dial(ANumber : string);
```



Dials a phone number in the background.

`Dial` instructs TAPI to prepare the modem for dialing, then to dial `ANumber`. All of these operations take place in the background. `TApdTapiDevice` generates the `OnTapiStatus` event to keep the program apprised of the dialing progress.

If `EnableVoice` is `False` (you are attempting a data connection), the `OnTapiPortOpen` event should be used as your notification that a data connection is established. If `EnableVoice` is `True` (you are attempting a voice connection), the `OnTapiConnect` event should be used as your notification. The `OnTapiConnect` event may be generated for data connections (TSP dependent), but this event does not indicate that the `TApdComPort` has a valid, open serial port.

If a busy signal is detected and MaxAttempts is greater than one, Dial redials the number after waiting RetryWait seconds. This continues until a connection is established or MaxAttempts dial attempts fail.

Due to a limitation of the Microsoft supplied TAPI Service Providers, the OnTapiConnect event is not completely reliable for detecting when the called party answers a voice call. Unimodem/V and Unimodem/5 do not implement the call progress notification techniques required to detect when a voice connection is actually established. As a result of this, the OnTapiConnect event is usually generated immediately after the modem completes dialing, regardless of whether the remote party answered, their phone was busy, or any other conditions. With some TSPs, it is also possible to get an OnTapiConnect event followed by an OnTapiFail event, if the call was busy or no dial tone was detected. The APROFAQ.HLP file contains several tips and tricks to detect when the remote party actually answers. TSPs supplied with dedicated voice boards usually provide much more detailed and accurate call progress notification.

The following example shows how to dial the U.S. Robotics BBS, waiting 5 minutes after a busy signal and retrying up to 10 times:

```
ApdTapiDevice1.RetryWait := 300;  
ApdTapiDevice1.MaxAttempts := 10;  
ApdTapiDevice1.Dial('1-847-262-6000');
```

See also: AutoAnswer, MaxAttempts, Number, RetryWait

Dialing

read-only, run-time property

property Dialing : Boolean

↳ Determines whether TAPI is placing an outgoing call or listening for an incoming call.

Dialing is True when TAPI is placing an outgoing call, False when TAPI is listening for or answering incoming calls. Dialing is intended primarily for use in status routines to distinguish between status events for incoming calls and status events for outgoing calls.

The following example shows an OnTapiStatus event handler that uses Dialing to update a TLabel on the current form:

```
TForm1 = class(TForm)
...
  ApdTapiDevice1 : TApdTapiDevice;
  Direction      : TLabel;
...
end;

procedure TForm1.ApdTapiDevice1TapiStatus(
  CP : TObject; First, Last : Boolean; Device, Message, Param1,
  Param2, Param3 : LongInt);
const
  DirectionStr : array[Boolean] of string = (
    'Incoming', 'Outgoing');
begin
  ...
  Direction.Caption := DirectionStr[ApdTapiDevice1.Dialing];
  ...
end;
```

See also: Number

EnableVoice

property

property EnableVoice : Boolean

Default: False

↪ Determines whether the initial mode of calls is DataModem (Fax or Data) or AutomatedVoice (Voice/DTMF).

If EnableVoice is True and a TAPI device is selected, Async Professional first verifies that AutomatedVoice capabilities exist for the selected device. If so, voice extensions such as DTMF and wave files are supported. Otherwise an ETapiVoiceNotSupported exception is raised and EnableVoice is set to False.

See also: OnTapiDTMF, OnTapiWave, PlayWaveFile, SendTone

```
property FailureCode : Integer
```

✚ FailureCode indicates the last TAPI failure.

During a Dial or AutoAnswer attempt, TAPI could detect a failure and generate the OnTapiFail event. The FailureCode property indicates the most severe error reported by TAPI. The value of FailureCode will be one of the LineCallState_ or LineDisconnectMode_ constants defined in AdTUtil.pas.

The FailureCodeMsg function will convert a FailureCode into a descriptive string based on the string resources.

The following example determines the reason for the failure and displays a corrective action to the user:

```
uses
  AdTUtil; { for the error constants }

procedure TForm1.ApdTapiDevice1TapiFail(Sender: TObject);
begin
  case ApdTapiDevice1.FailureCode of
    LineDisconnectMode_Busy : ShowMessage(
      'The number was busy, try again later');
    LineDisconnectMode_NoAnswer : ShowMessage(
      'No answer, try again later');
    LineDisconnectMode_NoDialtone : ShowMessage(
      'No dialtone, check your phone line');
    else
      ShowMessage(
        ApdTapiDevice1.FailureCodeMsg(ApdTapiDevice1.FailureCode));
  end;
end;
```

See also: FailureCodeMsg, OnTapiFail

```
function FailureCodeMsg(const FailureCode : Integer) : string;
```

✚ FailureCodeMsg converts a FailureCode into a descriptive string.

FailureCodeMsg will return a text message describing the failure code. FailureCode is the failure code, usually provided by the FailureCode property. See FailureCode for an example of how to use this method.

See also: FailureCode, OnTapiFail, TapiStatusMsg

```
property FilterUnsupportedDevices : Boolean
```

Default: True

- ↪ Determines whether unsupported devices are displayed in the TAPI device selection dialog box.

The TApdTapiDevice support TAPI line devices, with the DataModem and AutomatedVoice media modes. Some TAPI devices, notably the IP telephony devices in Windows 2000, are not supported by the TApdTapiDevice. If this property is True (the default) these devices will not be displayed in the TAPI device selection dialog. If this property is False, these devices will be displayed in the dialog.

See also: SelectDevice

GetDevConfig**method**

```
function GetDevConfig : TTapiConfigRec;
```

- ↪ Returns the configuration of the currently selected device.

The Data is binary, and described in the TAPI documentation as an “opaque” structure, meaning that it is not guaranteed to be consistent across different TAPI devices. For this reason, GetDevConfig must be called for a given device at some point before calling SetDevConfig or ShowConfigDialogEdit.

The record used by GetDevConfig (and the following configuration methods) is defined as follows:

```
TTapiConfigRec = record
    DataSize : Cardinal;
    Data : array[0..1023] of Byte;
end;
```

See also: SetDevConfig, ShowConfigDialogEdit

InterruptWave

run-time property

property InterruptWave : Boolean

Default: True

↪ Indicates whether the current wave file should stop when a DTMF tone is detected.

If InterruptWave is True, the currently playing wave file will stop when a DTMF tone is detected. This allows you to have an automated phone answering system in which user is allowed to interrupt the currently playing wave file with a DTMF selection. If InterruptWave is False, DTMF tones do not stop the currently playing wave file. If you want the caller to hear a wave file in its entirety, set InterruptWave to False before you start playing it.

See also: OnTapiDTMF, PlayWaveFile, StopWaveFile

MaxAttempts

property

property MaxAttempts : Word

Default: 3

↪ Determines the number of times Dial automatically dials a number.

This is the number of times a phone number is dialed, it is not the number of retries. When MaxAttempts is one, for example, the number is dialed only once. If the line is busy, it is not tried again.

See also: Attempt, RetryWait

MaxMessageLength

run-time property

property MaxMessageLength : LongInt

Default: 60

↪ The maximum allowed message length, in seconds, for messages recorded over the TAPI waveform audio device.

Use this parameter to specify the maximum length of recorded messages. A 60-second message will require about 950K of disk space given the default recording parameters. When the specified length of time passes, the OnTapiWaveNotify event will be generated with a Msg parameter of waDataReady. You could then save the wave file and terminate the call.

If the TrimSeconds property is set to a non-zero value then wave recording may terminate before MaxMessageLength is reached.

```
property Number : string
```

↳ The last phone number dialed.

Number is intended primarily for use in status routines, to display the last number dialed. Number is updated each time Dial is called.

The following example shows an OnTapiStatus event handler that displays the last number dialed.

```
TForm1 = class(TForm)
...
  ApdTapiDevice1 : TApdTapiDevice;
  NumberDialed : TLabel;
...
end;

procedure TForm1.ApdTapiDevice1TapiStatus(
  CP : TObject; First, Last : Boolean; Device, Message, Param1,
  Param2, Param3 : LongInt);
begin
  ...
  if Dialing then
    NumberDialed.Caption := ApdTapiDevice1.Number
  else
    NumberDialed.Caption := '';
  ...
end;
...
ApdTapiDevice1.Dial('1-847-262-6000');
```

See also: Dial, Dialing


```
property OnTapiCallerID : TTapiCallerIDEvent  
  
TTapiCallerIDEvent = procedure(  
    CP : TObject; ID, IDName : String) of object;
```

- ↳ Defines an event handler that is called after a connection is made and both a Caller ID string and a Caller ID name string are returned.

The OnTapiCallerID event makes it easy to access Caller ID information without having to know when it might be available on a call. Caller ID information is available only if it is supported on the selected device and by the telephone service.

The following example shows how to use the OnTapiCallerID event to get the Caller ID information and store it to edit controls.

```
procedure TForm1.ApdTapiDevice1TapiCallerID(  
    CP : TObject; ID, IDName : string);  
begin  
    CallerId.Text := ID;  
    CallerIdName.Text := IDName;  
end;
```

See also: CallerID, CallerIDName

```
property OnTapiConnect : TNotifyEvent
```

- ↳ Defines an event handler that is called when a connection is established.

Dial and AutoAnswer operations take place in the background. If a connection is established after a call to Dial or AutoAnswer, the TApdTapiDevice generates the OnTapiConnect event.

No parameters are passed to the OnTapiConnect event. The OnTapiConnect event is most useful and reliable when used to indicate when a voice connection has been established. For data connections (EnableVoice = False), the TApdComPort may not have received the serial port handle from TAPI when this event is generated. When you are establishing data connections, use the OnTapiPortOpen event instead.

```
property OnTapiDTMF : TTapiDTMFEvent  
  
TTapiDTMFEvent = procedure(  
    CP : TObject; Digit : Char; ErrorCode : LongInt;) of object;
```

↳ Defines an event handler that is called when a DTMF tone is detected.

Digit is a character that represents the phone button that was pressed on the remote phone device. The possible values are '0' through '9', '*', and '#'. ErrorCode is non-zero if an error occurs when a TAPI connection is made (in this case the OnTapiDTMF is generated just before the OnTapiConnect event).

The following example builds a string of up to ten DTMF tones (characters) in the global variable S.

```
procedure TForm1.ApdTapiDevice1TapiDTMF(  
    CP : TObject; Digits : Char; ErrorCode : LongInt);  
begin  
    if Length(S) < 11 then  
        S := S + Digit;  
    end;
```

See also: EnableVoice

OnTapiFail**event**

```
property OnTapiFail : TNotifyEvent
```

↳ Defines an event handler that is called when a connection attempt fails.

Dial and AutoAnswer operations take place in the background. If an attempt to establish a connection fails, the TApdTapiDevice generates the OnTapiFail event.

No parameters are passed to OnTapiFail. It is a notification to the application that a connection attempt failed.

The FailureCode property will contain the most severe error reported by TAPI. See the description of FailureCode for an example of how to use that property and the OnTapiFail event handler.

See also: FailureCode, FailureCodeMsg

```
property OnTapiLog : TTapiLogEvent  
  
TTapiLogEvent = procedure(  
    CP : TObject; Log : TTapiLogCode) of object;
```

↳ Defines an event handler that is called at designated points during a dial or answer attempt.

The primary purpose of this event is to give the application a chance to log auditing information about telephone calls and whether they succeed or fail. You can also use this event for start-up and cleanup activities, although `OnTapiPortOpen` and `OnTapiPortClose` might be better.

CP is the TAPI component that generated the event. Log is a code that indicates the state of the TAPI connection. The possible states are listed in “TAPI logging” on page 397. No other information is passed with this event, but you can use the `TApdTapiDevice` properties such as `Number` and `Dialing` to get additional information about the TAPI connection.

See “TAPI logging” on page 397 for more information.

See also: `Dialing`, `Number`, `TapiLog`

OnTapiPortClose

```
property OnTapiPortClose : TNotifyEvent
```

↳ Defines an event handler that is called immediately after `TApdTapiDevice` closes its associated `TApdComPort`.

The `TApdTapiDevice` component is responsible for opening and closing the associated `TApdComPort` at the appropriate times (when a connection is established or broken).

The serial port handle is invalid once the `OnTapiPortClose` event is generated, attempts to access the port will raise the `EPortNotOpen` exception.

Applications can use this event to perform additional port cleanup activities.

See also: `OnTapiPortOpen`

```
property OnTapiPortOpen : TNotifyEvent
```

- ↳ Defines an event handler that is called immediately after TApdTapiDevice opens its associated TApdComPort.

The TApdTapiDevice component is responsible for opening and closing the associated TApdComPort at the appropriate times (when a connection is established or broken).

The serial port associated with the selected TAPI device is valid, and available when this event is generated, it is safe to access the port properties and transmit characters.

Note that this event is not generated during a voice connection. When a voice connection is made TAPI retains exclusive access to the serial port. This event will be generated after a call to AutomatedVoiceToComms, once TAPI hands the TApdTapiDevice a valid serial port handle.

Applications can use this event to perform additional port setup activities.

See also: OnTapiPortClose

OnTapiStatus**event**

```
property OnTapiStatus : TTapiStatusEvent
```

```
TTapiStatusEvent = procedure(CP : TObject; First, Last : Boolean;  
    Device, Message, Param1, Param2, Param3 : Cardinal) of object;
```

- ↳ Defines an event handler that is called regularly during a TAPI dial or answer attempt.

TAPI performs dial and answer activities in the background, calling a callback routine whenever the state of the line or call changes. TApdTapiDevice installs a hidden callback routine and translates all callback calls into OnTapiStatus events.

CP is the TApdTapiDevice component that generated the event.

First is True on the first OnTapiStatus event to signal the status routine to perform its start-up activities (e.g., make the status display visible). Last is True on the last OnTapiStatus event to signal the status routine to perform its cleanup activities (e.g., remove the status display). First and Last are False on all other OnTapiStatus events.

The other parameters are the ones passed by TAPI to the callback routine. The only parameters that are of interest to most Async Professional programs are Message and Param1, which indicate the state of the current call. See “TAPI status processing” on page 394 for more information about the values of Message and Param1.

The remaining parameters (Device, Param2, and Param3) are intended for use in applications that extend the features provided by TApdTapiDevice and might need those status parameters.

TAPI generates callbacks only when it perceives a change in the state of the line or call. TAPI, therefore, does not generate callbacks when the modem stays in a single state for an extended period of time. For example, after dialing a number TAPI reports that the call is in the “proceeding” phase. It generates no further status callbacks until the call succeeds or fails. Since this can take many seconds, as much as 20 or even 30 seconds, the user might become concerned about the lack of positive feedback (is it still working?).

To solve this problem, TApdTapiDevice generates additional OnTapiStatus events, based on an internal timer (once per second). These status calls give the status routine an opportunity to update a timer, as the built-in TApdTapiStatus does.

OnTapiWaveNotify

event

```
property OnTapiWaveNotify : TTapiWaveNotify
TTapiWaveEvent = procedure(
    CP : TObject; Msg : TWaveMessage) of object;
TWaveMessage = (waPlayOpen, waPlayDone, waPlayClose,
    waRecordOpen, waDataReady, waRecordClose);
```

↳ Defines an event handler that is called when a wave file status changes.

The possible values for Msg are:

Value	Meaning
waPlayDone	The wave file is finished playing (it either completed normally or was stopped by a call to StopWaveFile).
waPlayOpen	The wave file is open.
waPlayClose	The wave file is closed.
waRecordOpen	The wave device is open for recording.
waRecordClose	The wave device is closed.
waDataReady	The wave device has recorded data and is ready to be saved.

The following example sets the Caption of a label after a wave file has finished playing:

```
procedure TForm1.ApdTapiDevice1TapiWaveEvent(  
    CP : TObject; Msg : TWaveMessage);  
begin  
    if Msg = waPlayDone then  
        Label4.Caption := 'Wave Device Idle...';  
    end;
```

See also: PlayWaveFile, StartWaveRecord, StopWaveFile, StopWaveRecord

OnTapiWaveSilence

event

```
FOnTapiWaveSilence : TTapiWaveSilence  
  
TTapiWaveSilence = procedure(CP : TObject;  
    var StopRecording : Boolean; var Hangup : Boolean) of object;
```

↳ Defines an event handler that is called when silence is detected while recording a wave file.

StopRecording is a var Boolean parameter that determines whether wave recording should stop. This parameter is True by default. Hangup is a var Boolean parameter that determines whether the call should be terminated. The parameter is also True by default.

This event works in conjunction with the TrimSeconds property. If TrimSeconds is 0 then OnTapiWaveSilence will not be generated. If you do not respond to this event then recording will stop and the call will be terminated when silence is detected. This is probably the desired behavior in most applications, so you may not use this event very often.

See also: StartWaveRecord, StopWaveRecord, TrimSeconds

PlayWaveFile

method

```
procedure PlayWaveFile(FileName : String);
```

↳ Plays a wave file.

FileName is the name of the wave file. The wave file starts playing immediately if there is not a wave file currently playing. If another wave file is currently playing and InterruptWave is True, the current wave file is stopped and the new wave file is played. If another wave file is playing and InterruptWave is False, PlayWaveFile returns without playing the new wave file. The wave file is played through the TAPI device if the UseSoundCard property is False (the default), or through the sound card if UseSoundCard is True.

The following example plays a wave file through the TAPI device:

```
ApdTapiDevice1.PlayWaveFile('greeting.wav');
```

See also: InterruptWave, OnTapiWaveNotify, StopWaveFile, UseSoundCard

```
property RetryWait : Word
```

Default: 60

↳ The number of seconds to wait after a busy signal before trying the number again.

After encountering a busy signal, `TApdTapiDevice` checks to see if it should try this number again by comparing `Attempts` to `MaxAttempts`. If more attempts are required, it first waits `RetryWait` seconds before dialing again to give the dialed machine time to complete the current session.

See also: `Attempts`, `MaxAttempts`

SaveWaveFile**method**

```
procedure SaveWaveFile(FileName : String; Overwrite : Boolean);
```

↳ Saves recorded wave data to disk.

`FileName` is the file to save. `Overwrite` indicates whether an existing file should be overwritten. If `Overwrite` is `False` and a file with the same name exists, then an exception is thrown. By default, data is recorded at 8,000 kHz, 16 bits, mono (the maximum sound quality allowed for most TAPI wave devices). You can save the wave data after the `OnTapiWaveNotify` event is received with a `Msg` parameter of `waDataReady`.

The following example starts recording wave data on a button click and saves the recorded data when notified that the recording is done:

```
procedure TForm1.Button1Click(Sender : TObject);
begin
    ApdTapiDevice1.StartWaveRecord;
end;

procedure TForm1.ApdTapiDevice1TapiWaveNotify(
    CP : TObject; Msg : TWaveMessage);
begin
    if Msg = waDataReady then
        ApdTapiDevice1.SaveWaveFile('Call01.wav', True);
    end;
```

See also: `OnTapiWaveNotify`, `StartWaveRecord`, `StopWaveRecord`

SelectDevice

method

```
procedure SelectDevice;
```

↳ Displays a dialog box to select a TAPI device.

SelectDevice uses the same dialog box as the property editor for SelectedDevice. You can call SelectDevice to prompt the user for a TAPI device to use for subsequent dial or answer operations.

See SelectedDevice for the displayed dialog box.

See also: FilterUnsupportedDevices, ShowTapiDevices

SelectedDevice

property

```
property SelectedDevice : string
```

↳ Determines the TAPI device to be used for dialing and answering.

TAPI assigns names to each installed modem. TApdTapiDevice components select among those devices by setting SelectedDevice to the name of the desired TAPI device. Since these names sometimes be rather lengthy and cumbersome to type, a property editor is provided for easier selection of devices.

Because the name specified in SelectedDevice must exactly match a TAPI device name, you should use this component in your application if you need to allow users to select a TAPI device.

SelectedDevice must be set before calling Dial, AutoAnswer, or ShowConfigDialog or they will raise an ETapiNoSelect exception.

SendTone

method

```
procedure SendTone(Digits : string);
```

↳ Sends a DTMF tone to a remote telephone.

SendTone replicates the press of a telephone touch pad button from within an application. Digits should consist of valid telephone touch pad buttons (i.e., '1' through '9', '*', and '#').

You can also use a comma (,) between characters for a short delay between the tones. Multiple comma characters can be used to create a longer delay.

The following example demonstrates how to use SendTone to send multiple tones with a delay.

```
SendTone('123456789,,0');
```

See also: EnableVoice, OnTapiDTMF


```
procedure SetDevConfig(const Config : TTapiConfigRec);
```

↳ Sets the selected device to the configuration defined in Config.

The selected device does not have to be “open” at the time of configuration. Config must originate from a previous call to GetDevConfig or ShowConfigDialogEdit.

See also: GetDevConfig, ShowConfigDialogEdit

SetRecordingParams**method**

```
procedure SetRecordingParams(NumChannels : Byte;  
    NumSamplesPerSecond : Integer; NumBitsPerSample : Byte);
```

↳ Sets the parameters used to record a wave file.

NumChannels is the number of channels to use for recording. A value of 1 indicates mono, and a value of 2 indicates stereo. Due to the nature of telephony, it is unlikely any TAPI devices support stereo recording. NumSamplesPerSecond is the number of samples per second to use for recording. NumBitsPerSample is the number of bits of data to record per sample.

By default recording parameters are set to 1 channel (mono), 8000 samples per second, 16 bits per sample. If your TAPI device supports other recording formats you can use this method to change the recording format. If you set the recording parameters to values not supported by your TAPI device you will get an ETapiWaveError when you attempt to begin recording. It is unlikely that you will need to change the recording parameters.

See also: StartWaveRecord

ShowConfigDialog**method**

```
procedure ShowConfigDialog;
```

↳ Displays the TAPI property sheets for the selected TAPI device.

TAPI maintains a set of port properties for each TAPI device. These property sheets are accessible from the Windows Modem applet in the Control Panel. You can also make them available in your application by calling ShowConfigDialog.

```
function ShowConfigDialogEdit(  
    const Init : TTapiConfigRec) : TTapiConfigRec;
```

↳ Shows a TAPI configuration dialog for the selected device.

Init must be initialized with a TTapiConfigRec obtained from a call to GetDevConfig with the same selected device. The returned TTapiConfigRec reflects any changes the user made to the configuration. The device itself is not updated at this point—the record is merely returned for use in a subsequent call to SetDevConfig.

See also: GetDevConfig, SetDevConfig

```
property ShowPorts : Boolean
```

Default: True

↳ Controls whether serial ports are displayed by the SelectDevice method.

ShowPorts is used in conjunction with ShowTapiDevices to determine what is displayed in the Device Selection dialog. The dialog is used as a property editor for the SelectedDevice property at design time, when you call the SelectDevice method at run time, and when a TAPI command is executed before a TAPI device is selected. See SelectedDevice to see a sample Device Selection dialog box.

If ShowPorts is True, the available serial ports are displayed in the drop-down box in the Device Selection dialog box. If ShowTapiDevices is True, the TAPI devices are displayed. If both ShowPorts and ShowTapiDevices are False, nothing is shown in the drop-down box in the dialog.

When ShowPorts is True, the available serial ports will be displayed in the dialog box. If one of these devices is selected, the TApdComPort.TapiMode will be set to tmOff. Use TapiMode to determine whether to open the port with ConfigAndOpen or whether to open the TApdComPort directly.

See also: SelectedDevice, ShowTapiDevices

ShowTapiDevices

property

```
property ShowTapiDevices : Boolean
```

Default: True

↳ Controls whether TAPI devices are displayed by the SelectDevice method.

If ShowTapiDevices is True, SelectDevice shows both TAPI devices and available serial ports. If ShowTapiDevices is False, SelectDevice shows only serial ports.

See also: SelectDevice, ShowPorts

SilenceThreshold

run-time property

```
property SilenceThreshold : Integer
```

Default: 50

↳ Specifies a value that is used as a measure of silence.

When the TrimSeconds property is set to a non-zero value, the wave data is examined as it is recorded. Silence is determined by comparing the average of the wave data for one second to a silence threshold as defined by SilenceThreshold. PCM data recorded by the TAPI wave driver generally has an amplitude of 400 to 800 for normal speech. A silence threshold of 50 (the default) is conservative. True silence on the phone line is probably less than 20, although anything under 200 could probably be considered silence. Modify the SilenceThreshold property if your phone lines contain more or less noise.

See also: OnTapiWaveSilence, StartWaveRecord, StopWaveRecord, TrimSeconds

StartWaveRecord

method

```
procedure StartWaveRecord;
```

↳ Starts the wave device recording.

Use StartWaveRecord to begin recording a wave file using the TAPI waveform audio device. Recording stops when the StopWaveRecord method is called, when the wave input buffer is full, or when silence is detected on the line. The size (in seconds) of the wave input buffer is determined by MaxMessageLength. Silence detection is controlled through the TrimSeconds property.

When recording stops, the OnTapiWaveNotify event is generated with Msg set to waDataReady. After receiving notification that wave data is ready, you must save the recorded data using SaveWaveFile.

The following example sets the maximum message length to 45 seconds, starts recording wave data on a button click, and then saves the recorded data when notified that the recording is done:

```
procedure TForm1.Button1Click(Sender : TObject);
begin
    ApdTapiDevice1.MaxMessageLength := 45;
    ApdTapiDevice1.StartWaveRecord;
end;

procedure TForm1.ApdTapiDevice1TapiWaveNotify(
    CP : TObject; Msg : TWaveMessage);
begin
    if Msg = waDataReady then
        ApdTapiDevice1.SaveWaveFile('Call01.wav');
    end;
end;
```

See also: `MaxMessageLength`, `PlayWaveFile`, `SaveWaveFile`, `StopWaveRecord`, `TrimSeconds`

StatusDisplay

property

```
property StatusDisplay : TApdAbstractTapiStatus
```

↳ An instance of a TAPI status window.

If `StatusDisplay` is nil (the default), `TApdTapiDevice` does not provide an automatic status window. You can install an `OnTapiStatus` event handler to display status in this case.

If you create an instance of a class derived from `TApdAbstractTapiStatus` or use the supplied `TApdTapiStatus` component (see page 443) and assign it to `StatusDisplay`, the status window is displayed and updated automatically.

`StatusDisplay` is usually set automatically at design time to the first `TApdAbstractStatus` or derived component the `TApdTapiDevice` finds on the form. If necessary, use the Object Inspector to select a different status component.

Setting the `StatusDisplay` property at run time is necessary only when using a dynamically created status display or when selecting among several status components.

```
procedure StopWaveFile;
```

↳ Stops the wave file that is currently playing.

The wave file is halted regardless of the value of the `InterruptWave` property. `StopWaveFile` generates two `OnTapiWave` events. The first has a Code of `WOM_DONE` and the second has a Code of `WOM_CLOSE`. If no wave file is currently playing then `StopWaveFile` returns silently. The following example stops a wave file, if one is currently playing:

```
if ApdTapiDevice1.WaveState = wsPlaying then
  ApdTapiDevice1.StopWaveFile;
```

See also: `InterruptWave`, `OnTapiWaveNotify`, `PlayWaveFile`, `WaveStatus`

```
procedure StopWaveRecord;
```

↳ Stops the wave file that is recording.

It is not always necessary to call this function since wave recording may be halted as the result of the wave buffer filling up, or as a result of silence on the line. The wave record buffer is set to an initial size based on `MaxMessageLength`. If you do not call `StopWaveRecord`, the recording automatically stops after `MaxMessageLength` seconds. The `TrimSeconds` property is set to 2 seconds by default. Wave recording will stop after 2 seconds of silence are detected on the line (unless you respond to the `OnTapiWaveSilence` event and override the default).

See also: `MaxMessageLength`, `OnTapiWaveNotify`, `SaveWaveFile`, `StartWaveRecording`, `TrimSeconds`

```
property TapiLog : TApdTapiLog
```

↳ An instance of a TAPI logging component.

If `TapiLog` is `nil` (the default), `TApdTapiDevice` does not provide automatic logging. You can install an `OnTapiLog` event handler to provide logging services in this case.

`TapiLog` is usually set automatically at design time to the first `TApdTapiLog` or derived component the `TApdTapiDevice` finds on the form. If necessary, use the Object Inspector to select a different a logging component.

Setting the `TapiLog` property at run time is necessary only when using a dynamically created logging component or when selecting among several logging components.

```
property TapiState : TTapiState

TTapiState = (tsIdle, tsOffering, tsAccepted, tsDialTone,
    tsDialing, tsRingback, tsBusy, tsSpecialInfo, tsConnected,
    tsProceeding, tsOnHold, tsConferenced, tsOnHoldPendConf,
    tsOnHoldPendTransfer, tsDisconnected, tsUnknown);
```

Default: tsNone

↪ The state of the TAPI operation.

When TapiState is referenced, APRO retrieves state information from TAPI and returns the result as TapiState. For completeness, the TTapiState enumeration contains all possible returns from TAPI -- which is a superset of values that you will likely see in an APRO application.

Note: Since APRO retrieves this value from TAPI every time you check it, you should avoid calling it too often. In other words, sitting in a loop continuously polling TapiState would not be a good idea.

Value	Meaning
tsIdle	No TAPI operations in progress.
tsOffering	TAPI is offering an incoming call.
tsAccepted	APRO has accepted an incoming call.
tsDialTone	Dial tone detected.
tsDialing	Waiting for dial to complete or fail.
tsRingback	Ringback detected.
tsBusy	Line is busy.
tsSpecialInfo	TAPI service provider specific.
tsConnected	Call is connected.
tsProceeding	Call is proceeding.
tsOnHold	Call has been placed on hold.
tsConferenced	Call is conferenced.

Value	Meaning
tsOnHoldPendConf	Call is being conferenced.
tsOnHoldPendTransfer	Call is being transferred.
tsDisconnected	Call has been disconnected.
tsUnknown	TAPI state unknown to APRO.

TapiStatusMsg **method**

```
function TApdCustomTapiDevice.TapiStatusMsg(
    const Message, State, Reason : DWORD) : string;
```

↳ Returns a text message for the progress of the current TAPI call.

TapiStatusMsg is intended primarily for use in OnTapiStatus event handlers. Message is the Message parameter passed into OnTapiStatus. State is the Param1 parameter passed into OnTapiStatus. TapiStatusMsg combines the values of Message and State to arrive at a unique resource string ID, stored in APW.RC. Reason is the Param2 parameter passed into OnTapiStatus. Reason adds additional information for LineCallState_Disconnected messages. This parameter is ignored for all other messages.

The following example shows an OnTapiStatus event handler that calls TapiStatusMsg and displays the returned string:

```
TForm1 = class(TForm)
...
    ApdTapiDevice1 : TApdTapiDevice;
    StatusStr       : TLabel;
...
end;

procedure TForm1.ApdTapiDevice1TapiStatus(
    CP : TObject; First, Last: Boolean; Device, Message, Param1,
    Param2, Param3 : LongInt);
begin
    ...
    StatusStr.Caption :=
        ApdTapiDevice1.TapiStatusMsg(Message, Param1, Param2);
    ...
end;
```

See also: OnTapiStatus

```
function TranslateAddress(CanonicalAddr : String) : String;
```

↳ Translates a canonical address into a dialable address.

A canonical address is an address that contains the country code as well as the phone number. TAPI also takes into account any settings you have made to your modem properties in the Control Panel. For example, if you have call waiting enabled and the code to disable call waiting is *70, TAPI prepends *70 to the dialable address string when you call TranslateAddress.

See also: Dial

```
property TrimSeconds : Integer
```

Default: 2

↳ Sets the number of seconds of silence to detect when recording wave files.

Wave recording can be terminated in one of three ways. First, you can manually terminate recording by calling StopWaveRecord. Second, recording will automatically terminate when the amount of time specified by MaxMessageLength has passed. Finally, wave recording can terminate as a result of silence detected on the line.

When TrimSeconds is set to a non-zero value, the wave data is examined as it is recorded. Silence is determined by comparing the average of the wave data for one second to a silence threshold as defined by the SilenceThreshold property. If TrimSeconds seconds of silence is detected, the OnTapiWaveSilence event is generated. If no OnTapiWaveSilence event is defined then the recording is stopped and the call is terminated. Even after a hangup, a telephone line contains a good deal of random noise so it is not guaranteed that silence will be detected immediately after a hangup.

See also: OnTapiWaveSilence, SilenceThreshold, StartWaveRecord, StopWaveRecord

property UseSoundCard : Boolean

Default: False

↳ Determines where the output from PlayWaveFile is sent.

UseSoundCard determines whether the output from PlayWaveFile goes to the TAPI device or to the sound card. By default the output is sent to the TAPI waveform audio device (through the phone). Set UseSoundCard to True to play the wave file through the sound card.

The following example plays a wave file through the sound card rather than over the phone line and then resets the device so that subsequent sounds are played through the TAPI device:

```
ApdTapiDevice1.UseSoundCard := True;  
ApdTapiDevice1.PlayWaveFile('Call01.wav');  
ApdTapiDevice1.UseSoundCard := False;
```

See also: PlayWaveFile

property WaveFileName : TFileName

↳ The name of the current wave file.

If a wave file is currently playing, WaveFileName is the name of the file. If no wave file is currently playing, WaveFileName is the name of the last wave file that was played. WaveFileName is automatically set when you use PlayWaveFile to play a file.

The following example sets a label's Caption to the name of the current wave file:

```
Label1.Caption := ApdTapiDevice1.WaveFileName;
```

See also: PlayWaveFile

```
property WaveState : TWaveState  
TWaveState = (wsIdle, wsPlaying, wsRecording, wsData);
```

↪ The current state of the TAPI waveform device.

The possible values for wsData are:

Value	Meaning
wsIdle	The wave device is not in use.
wsPlaying	The wave device is playing a wave file.
wsRecording	The wave device is recording a wave file.
wsData	Data is available in the wave buffer.

The following example stops a wave file if one is currently playing:

```
if ApdTapiDevice1.WaveState = wsPlaying then  
  ApdTapiDevice1.StopWaveFile;
```

See also: PlayWaveFile, StartWaveRecord, StopWaveFile, StopWaveRecord

TapdAbstractTapiStatus Class

TapdAbstractTapiStatus is an abstract class that defines the methods and properties needed by a component that automatically displays status while TapdTapiDevice is dialing or answering a call. You generally won't need to create a descendent class of your own, since Async Professional supplies one, the TapdTapiStatus component (see page 443).

However, TapdTapiStatus shows a particular set of information about a call in a predefined format. If this format is not suitable for your needs, you can create your own descendant of TapdAbstractTapiStatus. The best way to start is to study the source code of TapdTapiStatus (in the AdTStat unit) and its form, TStandardTapiDisplay.

The TapdAbstractTapiStatus class contains an instance of a TForm that holds controls used to display the dial or answer status. You design the form, create an instance, and assign the instance to the Display property of TapdAbstractTapiStatus.

TapdAbstractTapiStatus overrides the standard VCL properties Ctl3D, Position, and Visible and the standard VCL method Show. When these routines are used in the status component, the overridden versions perform the same actions on the associated Display form. Thus you can display the status form by calling Show, erase it by setting Visible to False, adjust its position by assigning to Position, and use 3D effects by setting Ctl3D to True.

Once you create an instance of your TapdAbstractTapiStatus descendant, you must assign it to the StatusDisplay property of your TapdTapiDevice component. When TapdTapiDevice needs to update the status display, it calls the UpdateDisplay method of TapdAbstractTapiStatus, which you must override to update your status window.

Hierarchy

TComponent (VCL)	
❶ TapdBaseComponent (OOMisc)	8
TapdAbstractTapiStatus (AdTapi)	

Properties

Display	TapiDevice	❶ Version
---------	------------	-----------

Methods

CreateDisplay	DestroyDisplay	UpdateDisplay
---------------	----------------	---------------

Reference Section

CreateDisplay

dynamic abstract method

```
procedure CreateDisplay; dynamic; abstract;
```

- ↳ An abstract method that creates a form to display dialing and answering status.

A descendant of `TApdAbstractTapiStatus` must override this method with a routine that creates a `TForm` component that contains various controls (typically of type `TLabel`) for displaying the call progress. The `TForm` should also contain a `TButton` control and associated `OnClick` event handler that allow the user to cancel the dial or answer operation.

`CreateDisplay` must then assign the instance of this form to the `Display` property.

See also: `DestroyDisplay`, `Display`

DestroyDisplay

dynamic abstract method

```
procedure DestroyDisplay; dynamic; abstract;
```

- ↳ An abstract method that destroys the display form.

A descendant of `TApdAbstractTapiStatus` must override this method to destroy the `TForm` instance created by `CreateDisplay`.

See also: `CreateDisplay`, `Display`

Display

run-time property

```
property Display : TForm
```

- ↳ A reference to the form created by `CreateDisplay`.

`CreateDisplay` must assign a properly initialized instance of a `TForm` to this property. `UpdateDisplay` can refer to this property to update the status window.

See also: `CreateDisplay`, `UpdateDisplay`

TapiDevice

property

```
property TapiDevice : TApdCustomTapiDevice
```

- ↳ The `TApdTapiDevice` component that is using the status component.

When you derive components from `TApdAbstractTapiStatus`, you will probably reference `TApdTapiDevice` properties to display information about the progress of the dial or answer operation. Use this property to do so. It is automatically initialized when you assign the status component to the `StatusDisplay` property of `TApdTapiDevice`.

```
procedure UpdateDisplay(First, Last : Boolean;  
    Device, Message, Param1, Param2, Param3 : DWORD);  
    virtual; abstract;
```

↳ An abstract method that writes the contents of the status window.

A descendant of `TApdAbstractTapiStatus` must override this method to update the display form. The `TApdTapiDevice` component calls this method regularly from its `OnTapiStatus` event handler.

On the first call to `UpdateDisplay`, `First` equals `True` and `UpdateDisplay` should call the `Show` method of `Display` to draw the outline and background of the status form. On the last call to `UpdateDisplay`, `Last` equals `True` and `UpdateDisplay` should set the `Visible` property of `Display` to `False` to erase the status window.

For all other calls to `UpdateDisplay`, `First` and `Last` are both `False`. During these calls `UpdateDisplay` should update the various labels in the `Display` form. To get information about the dial or answer operation, use the `TapiDevice` field of `TApdAbstractTapiStatus` to read the values of various properties such as `Number` and `Dialing`.

The `CancelClick` event handler, if one is provided, should call the `CancelCall` method of `TApdTapiDevice` to abort the dial or answer operation.

TapdTapiStatus Component

TapdTapiStatus is a descendant of TapdAbstractTapiStatus that implements a standard TAPI status display. To use it, just create an instance and assign it to the StatusDisplay property of your TapdTapiDevice component. TapdTapiStatus includes all of the most frequently used information about a call and it provides a Cancel button so that the user can abort the call at any time.

TapdTapiStatus overrides all the abstract methods of TapdAbstractTapiStatus. TapdTapiStatus has no methods that you must call or properties that you must adjust. You might want to change the settings of the Ctl3D and Position properties to modify the appearance and placement of the window.

Figure 12.1 shows the TStandardTapiDisplay form that is associated with a TapdTapiStatus component.

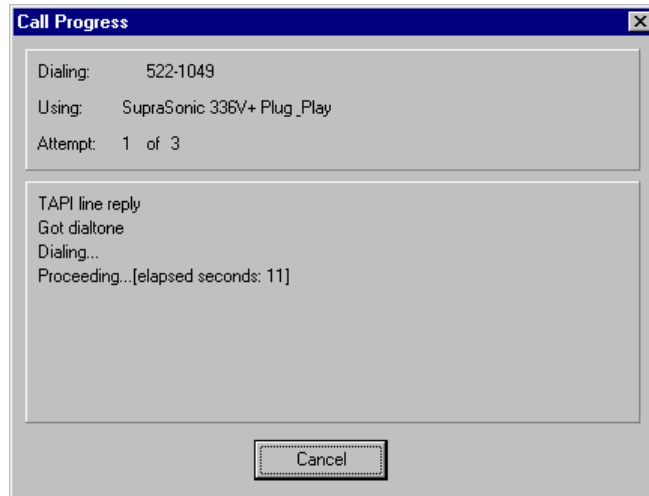


Figure 12.1: TStandardTapiDisplay form.

For an example of using a TapdTapiStatus component, see either the dial or answer examples in “Making calls” on page 400 and “Answering calls” on page 402.

Hierarchy

- TComponent (VCL)
 - TApdBaseComponent (OOMisc) 8
 - TApdAbstractTapiStatus (AdTapi) 440
 - TApdTapiStatus (AdTStat)

TApdTapiLog Class

TApdTapiLog is a small class that can be associated with a TApdTapiDevice to provide automatic TAPI logging services. Simply create an instance of TApdTapiLog and assign it to the TapiLog property of the TApdTapiDevice.

TApdTapiLog creates or appends to a text file whose name is given by the TapiHistoryName property. Each time the OnTapiLog event is generated, the associated TApdTapiLog instance opens the file, writes a new line to it, and closes the file.

Following is a sample of the text file created by TApdTapiLog:

```
5/7/96 10:11:34 PM : call started
5/7/96 10:11:34 PM :   dialing 262-6000
5/7/96 10:11:53 PM :   cancelled
5/7/96 10:11:53 PM : call finished

5/7/96 10:50:50 PM : call started
5/7/96 10:50:50 PM :   dialing 262-6000
5/7/96 10:51:02 PM :   busy
5/7/96 10:51:02 PM :   dial failed
5/7/96 10:51:07 PM :   dialing 262-6000
5/7/96 10:51:11 PM :   cancelled
5/7/96 10:51:11 PM : call finished

5/7/96 11:11:34 PM : call started
5/7/96 11:11:34 PM :   dialing 262-6000
5/7/96 11:11:53 PM :   connected
5/7/96 11:30:07 PM : call finished
```

Hierarchy

TComponent (VCL)

- TApdBaseComponent (OOMisc) 8
 - TApdTapiLog (AdTapi)

Properties

TapiHistoryName TapiDevice ● Version

Methods

UpdateLog

Reference Section

TapiDevice

property

```
property TapiDevice : TApdCustomTapiDevice
```

↳ The TAPI component that is using the log component.

TapiDevice is automatically initialized when the TapiLog property of the owning TAPI component is set. TapiDevice can be changed to assign the log component to a different TAPI component.

TapiHistoryName

property

```
property TapiHistoryName : string
```

Default: "APROTAPI.HIS"

↳ Determines the name of the TAPI log file.

The value of TapiHistoryName should be set before calling Dial or AutoAnswer. However, because the log file is opened and closed for each update, TapiHistoryName can be changed at any time. If TapiHistoryName is set to an empty string, automatic logging is disabled until a non-empty string is assigned.

See also: TApdTapiDevice.AutoAnswer, TApdTapiDevice.Dial

UpdateLog

virtual method

```
procedure UpdateLog(const Log : TTapiLogCode); virtual;
```

```
TTapiLogCode = (ltapiNone, ltapiCallStart,  
    ltapiCallFinish, ltapiDial, ltapiAnswer, ltapiConnect,  
    ltapiCancel, ltapiDrop, ltapiBusy, ltapiDialFail);
```

↳ Called for each TAPI logging event.

The Log parameter has the same values passed to the OnTapiLog event handler of TApdTapiDevice. UpdateLog creates or appends to the log file, builds and writes a text string for each event, and closes the log file.

TApdTapiLog contains a field named TapiDevice that UpdateLog uses to obtain additional information (e.g., Number and Dialing) about the dial or answer operation.

See also: TApdTapiDevice.OnTapiLog

Chapter 13: Modem Components

Async Professional provides a modem database and several components for configuring, dialing, and answering modems and managing phone books. These components are ideal for controlling modems since they include a modem database and all of the features a modem-based program might need. Your users simply select their modem (or the closest match) from the database and your application loads the associated configuration strings for that modem from the database.

The modem components provide routines for retrieving modem information from the libmodem database, manipulating a modem by sending commands through TApdComPort components, and processing response data, a dialing engine for making repeated dial attempts, and a few user-interface components for maintaining phonebooks and dialing the modem. The following units and components are described in this chapter.

For projects ported from previous versions of Async Professional, the TAdModem, TApdSModem and supporting components are still available as deprecated components.

AdMdm

Contains a component (TAdModem) that provides a simple interface for accessing the most commonly used modem operations. It combines the features of most of the other modem components into one component.

AdLibMdm

Contains TApdLibModem, a VCL component that provides an interface to the libmodem modem database.

AdMdmDlg

Contains a modem status dialog for use with the TAdModem component.

modemcap and libmodem

The TAdModem component uses a Windows version of libmodem to obtain modem configuration information from the modemcap database. Modemcap is a set of XML documents that defines configuration, response and identification information for a wide range of modems. The following section describes the TAdLibModem class that retrieves information from the modemcap database. Refer to the libmodem help pages for a more detailed description of modemcap and libmodem.

TApdLibModem Component

The TApdLibModem component is a Windows implementation of libmodem. This class provides several routines for access and maintaining lists of known modems and details on individual modems.

Detailed configuration, response and feature information for a wide variety of modems is stored in numerous XML documents. By default, these documents are stored in the /etc/modemcap directory. The modemcap.xml document contains an index of all available modem definitions.

Several data structures are used to describe the modem cap index and the modem detail files. These types are defined in the AxLibMdm.pas unit. The structures and methods that are most applicable to the TAdModem are listed here with a brief description of the structure. Refer to the libmodem source code (AdLibMdm.pas) and help pages for a more detailed discussion of these structures.

The TApdLibModem class uses the following structures. Several fields are TLists, each item points to an item in a sequence, the type of item is provided following the field name.

```
{ an entry from modemcap.xml describing the location and
identification of a single modem }
PLmModemName = ^TLmModemName;
TLmModemName = record
    ModemName      : string;
    Manufacturer   : string;
    Model          : string;
    ModemFile      : string;
end;

{ a modem response }
PLmResponseData = ^TLmResponseData;
TLmResponseData = record
    Response                : string;
    ResponseType            : string;
end;
```

```

{ lots of modem responses }
PLmResponses = ^TLmResponses;
TLmResponses = record
    OK                                     : TList;    // LmResponseData
    NegotiationProgress                   : TList;    // LmResponseData
    Connect                               : TList;    // LmResponseData
    Error                                 : TList;    // LmResponseData
    NoCarrier                             : TList;    // LmResponseData
    NoDialTone                            : TList;    // LmResponseData
    Busy                                  : TList;    // LmResponseData
    NoAnswer                              : TList;    // LmResponseData
    Ring                                  : TList;    // LmResponseData
    VoiceView1                           : TList;    // LmResponseData
    VoiceView2                           : TList;    // LmResponseData
    VoiceView3                           : TList;    // LmResponseData
    VoiceView4                           : TList;    // LmResponseData
    VoiceView5                           : TList;    // LmResponseData
    VoiceView6                           : TList;    // LmResponseData
    VoiceView7                           : TList;    // LmResponseData
    VoiceView8                           : TList;    // LmResponseData
    RingDuration                          : TList;    // LmResponseData
    RingBreak                             : TList;    // LmResponseData
    Date                                  : TList;    // LmResponseData
    Time                                  : TList;    // LmResponseData
    Number                                : TList;    // LmResponseData
    Name                                  : TList;    // LmResponseData
    Msg                                    : TList;    // LmResponseData
    SingleRing                            : TList;    // LmResponseData
    DoubleRing                            : TList;    // LmResponseData
    TripleRing                            : TList;    // LmResponseData
    Voice                                  : TList;    // LmResponseData
    Fax                                    : TList;    // LmResponseData
    Data                                  : TList;    // LmResponseData
    Other                                  : TList;    // LmResponseData
end;

{ a modem command }
PLmModemCommand = ^TLmModemCommand;
TLmModemCommand = record
    Command                               : string;
    Sequence                              : Integer;
end;

{ fax commands and responses }
TLmFaxClassDetails = record
    ModemResponseFaxDetect                : string;
    ModemResponseDataDetect               : string;

```

```

    SerialSpeedFaxDetect      : string;
    SerialSpeedDataDetect    : string;
    HostCommandFaxDetect     : string;
    HostCommandDataDetect    : string;
    ModemResponseFaxConnect  : string;
    ModemResponseDataConnect : string;
    AnswerCommand            : TList;
end;

{ more fax commands and responses }
TLmFaxDetails = record
    ExitCommand              : string;
    PreAnswerCommand         : string;
    PreDialCommand           : string;
    ResetCommand             : string;
    SetupCommand             : string;
    EnableV17Recv            : string;
    EnableV17Send            : string;
    FixModemClass            : string;
    FixSerialSpeed           : string;
    HighestSendSpeed         : string;
    LowestSendSpeed          : string;
    HardwareFlowControl      : string;
    SerialSpeedInit          : string;
    Cl1FCS                   : string;
    Cl2DC2                   : string;
    Cl2lsEx                  : string;
    Cl2RecvBOR               : string;
    Cl2SendBOR               : string;
    Cl2SkipCtrlQ             : string;
    Cl2SWBOR                 : string;
    Class2FlowOff            : string;
    Class2FlowHW             : string;
    Class2FlowSW             : string;
    FaxClass1                : TLmFaxClassDetails;
    FaxClass2                : TLmFaxClassDetails;
    FaxClass2_0              : TLmFaxClassDetails;
end;

{ supported wave formats }
PLmWaveFormat = ^TLMWaveFormat;
TLMWaveFormat = record
    ChipSet                  : string;
    Speed                    : string;
    SampleSize               : string;
end;

```

```

{ wave details }
TlMWaveDriver = record
    BaudRate                : string;
    WaveHardwareID          : string;
    WaveDevices              : string;
    LowerMid                 : string;
    LowerWaveInPid           : string;
    LowerWaveOutPid          : string;
    WaveOutMixerDest         : string;
    WaveOutMixerSource       : string;
    WaveInMixerDest          : string;
    WaveInMixerSource        : string;
    WaveFormat               : TList;    // LmWaveFormat
end;

{ voice modem properties }
TlMVoiceSettings = record
    VoiceProfile             : string;
    HandsetCloseDelay        : Integer;
    SpeakerPhoneSpecs       : string;
    AbortPlay                : string;
    CallerIDOutSide          : string;
    CallerIDPrivate          : string;
    TerminatePlay            : string;
    TerminateRecord          : string;
    VoiceManufacturerID      : string;
    VoiceProductIDWaveIn     : string;
    VoiceProductIDWaveOut    : string;
    VoiceSwitchFeatures      : string;
    VoiceBaudRate            : Integer;
    VoiceMixerMid            : string;
    VoiceMixerPid            : string;
    VoiceMixerLineID         : string;

    CloseHandset             : TList;    // LmModemCommand;
    EnableCallerID           : TList;    // LmModemCommand;
    EnableDistinctiveRing    : TList;    // LmModemCommand;
    GenerateDigit            : TList;    // LmModemCommand;
    HandsetPlayFormat        : TList;    // LmModemCommand;
    HandsetRecordFormat      : TList;    // LmModemCommand;
    LineSetPlayFormat        : TList;    // LmModemCommand;
    LineSetRecordFormat      : TList;    // LmModemCommand;
    OpenHandset              : TList;    // LmModemCommand;
    SpeakerPhoneDisable      : TList;    // LmModemCommand;
    SpeakerPhoneEnable       : TList;    // LmModemCommand;
    SpeakerPhoneMute         : TList;    // LmModemCommand;
    SpeakerPhoneSetVolumeGain : TList;    // LmModemCommand;

```

```

SpeakerPhoneUnMute      : TList;    // LmModemCommand;
StartPlay               : TList;    // LmModemCommand;
StartRecord             : TList;    // LmModemCommand;
StopPlay               : TList;    // LmModemCommand;
StopRecord             : TList;    // LmModemCommand;
VoiceAnswer            : TList;    // LmModemCommand;
VoiceDialNumberSetup   : TList;    // LmModemCommand;
VoiceToDataAnswer      : TList;    // LmModemCommand;
WaveDriver             : TLMWaveDriver;
end;

{ lots of specialized modem commands }
TLmModemSettings = record
    Prefix                : string;
    Terminator            : string;
    DialPrefix            : string;
    DialSuffix            : string;
    SpeakerVolume_High    : string;
    SpeakerVolume_Low     : string;
    SpeakerVolume_Med     : string;
    SpeakerMode_Dial      : string;
    SpeakerMode_Off       : string;
    SpeakerMode_On        : string;
    SpeakerMode_Setup     : string;
    FlowControl_Hard      : string;
    FlowControl_Off       : string;
    FlowControl_Soft      : string;
    ErrorControl_Forced   : string;
    ErrorControl_Off      : string;
    ErrorControl_On       : string;
    ErrorControl_Cellular : string;
    ErrorControl_Cellular_Forced : string;
    Compression_Off       : string;
    Compression_On        : string;
    Modulation_Bell       : string;
    Modulation_CCITT      : string;
    Modulation_CCITT_V23  : string;
    SpeedNegotiation_On   : string;
    SpeedNegotiation_Off  : string;
    Pulse                 : string;
    Tone                  : string;
    Blind_Off             : string;
    Blind_On              : string;
    CallSetupFailTimer    : string;
    InactivityTimeout     : string;
    CompatibilityFlags    : string;

```



```

        ConfigDelay                : Integer;
    end;

    { modem hardware settings }
    TLmModemHardware = record
        AutoConfigOverride          : string;
        ComPort                     : string;
        InvalidRDP                   : string;
        IoBaseAddress                : Integer;
        InterruptNumber              : Integer;
        PermitShare                   : Boolean;
        RxFIFO                       : string;
        RxTxBufferSize               : Integer;
        TxFIFO                       : string;
        Pcmcia                       : string;
        BusType                      : string;
        PCCARDAttributeMemoryAddress : Integer;
        PCCARDAttributeMemorySize    : Integer;
        PCCARDAttributeMemoryOffset  : Integer;
    end;

    { the whole shebang }
    PLmModem = ^TLmModem;
    TLmModem = record
        Inheritance                  : string;
        AttachedTo                   : string;
        FriendlyName                 : string;
        Manufacturer                 : string;
        Model                       : string;
        ModemID                     : string;
        InactivityFormat             : string;
        Reset                       : string;
        DCB                         : string;
        Properties                   : string;
        ForwardDelay                 : Integer;
        VariableTerminator           : string;
        InfPath                     : string;
        InfSection                   : string;
        ProviderName                 : string;
        DriverDesc                   : string;
        ResponsesKeyName             : string;
        Default                     : string;
        CallSetupFailTimeout         : Integer;
        InactivityTimeout            : Integer;
        SupportsWaitForBongTone      : Boolean;
        SupportsWaitForQuiet         : Boolean;
        SupportsWaitForDialTone      : Boolean;
    end;

```

```

SupportsSpeakerVolumeLow      : Boolean;
SupportsSpeakerVolumeMed      : Boolean;
SupportsSpeakerVolumeHigh     : Boolean;
SupportsSpeakerModeOff        : Boolean;
SupportsSpeakerModeDial       : Boolean;
SupportsSpeakerModeOn         : Boolean;
SupportsSpeakerModeSetup      : Boolean;
SupportsSetDataCompressionNegot : Boolean;
SupportsSetErrorControlProtNegot : Boolean;
SupportsSetForcedErrorControl  : Boolean;
SupportsSetCellular           : Boolean;
SupportsSetHardwareFlowControl : Boolean;
SupportsSetSoftwareFlowControl : Boolean;
SupportsCCITTBellToggle       : Boolean;
SupportsSetSpeedNegotiation    : Boolean;
SupportsSetTonePulse           : Boolean;
SupportsBlindDial              : Boolean;
SupportsSetV21V23             : Boolean;
SupportsModemDiagnostics       : Boolean;
MaxDTERate                     : Integer;
MaxDCERate                     : Integer;
CurrentCountry                 : string;
MaximumPortSpeed               : Integer;
PowerDelay                     : Integer;
ConfigDelay                    : Integer;
BaudRate                       : Integer;
Responses                      : TListResponses;
Answer                         : TList;
Fax                            : TList;
FaxDetails                     : TListFaxDetails;
Voice                          : TListVoiceSettings;
Hangup                         : TList;
Init                           : TList;
Monitor                        : TList;
Settings                       : TListModemSettings;
Hardware                       : TListModemHardware;
BaudRates                      : TListStringList;
Options                        : TListStringList;
end;

```

Not all of the proceeding structures and fields are used in Async Professional, they are included for future enhancements.

Hierarchy

TComponent (VCL)

 TApdLibModem(AdLibMdm)

Properties

 ModemCapFolder

Methods

AddModem	DeleteModem	GetModems
AddModemRecord	DeleteModemRecord	SelectModem
CreateNewDetailFile	GetModem	

Reference Section

AddModem

method

```
function AddModem(  
    const ModemDetailFile : string; Modem : TLmModem) : Integer;
```

↳ Adds a modem definition to modemcap.

AddModem adds the modem specified by Modem to the modem detail file specified by ModemDetailFile. This function will fail if the modem detail file already contains a modem with the same FriendlyName as Modem. The return value will be an ELMXxx error constant describing the result of the call to modemcap.

See also: AddModemRecord

AddModemRecord

method

```
function AddModemRecord(  
    ModemRecord : TLmModemRecord) : Integer;
```

↳ Adds a modem record to the list of available modems.

AddModemRecord adds the modem record specified by ModemRecord to the list of available modems to the modemcap index. This function will fail if the modem cap index already contains a modem with the same name, manufacturer, and modem as ModemRecord. The return value will be an ELMXxx error constant describing the result of the call to modemcap.

See also: AddModem

CreateNewDetailFile

method

```
function CreateNewDetailFile(  
    const ModemDetailFile : string) : Integer;
```

↳ Creates a new modem detail file.

The CreateNewDetailFile method creates a new modem detail file with the appropriate XML headers.

ModemDetailFile is the name of the new modem detail file. If ModemDetailFile already exists, it will be overwritten. The result of this method is ecOK if the file was created successfully.

DeleteModem

method

```
function DeleteModem(  
    const ModemDetailFile : string; Modem : TLmModem) : Integer;
```

↪ Deletes a modem detail record from a modem list.

DeleteModem will search the modem detail file specified by ModemDetailFile for the modem pointed to by Modem. If the modem is found, it will be removed from the list. The return value will be an ELMxxx error constant describing the result of the call to modemcap.

See also: DeleteModemRecord

DeleteModemRecord

method

```
function DeleteModemRecord(  
    ModemRecord : TLmModemRecord) : Integer;
```

↪ Deletes a modem record from a modemcap index.

DeleteModemRecord will search the modemcap index for the modem record pointed to by ModemRecord. If the modem is found it will be removed from the list. The return value will be an ELMxxx error constant describing the result of the call to modemcap.

See also: DeleteModem

GetModem

method

```
function GetModem(const ModemDetailFile,  
    ModemName : string; var Modem : TLmModem) : Integer;
```

↪ Retrieves a specific modem from the modem detail file.

GetModem retrieves a modem definition from modemcap. ModemDetailFile is the name of the modem detail file where the definition resides; ModemName is the name of the modem to retrieve. Modem is the TLmModem structure that contains the details of the modem.

If this method is successful, the return value is ecOK. If the ModemDetailFile is not found, the return value is ecFileNotFound. If a modem matching ModemName is not found in the ModemDetailFile, the return value is ecModemNotFound.

See also: SelectModem

```
function GetModems(  
    const ModemDetailFile : string) : TStringList;
```

↳ Retrieves all modems from a modem detail file.

GetModems retrieves all modem definitions contained in a specific modem detail file. ModemDetailFile is the name of the modem detail file. If successful, the return value is a TStringList containing the details of all modems contained in ModemDetailFile. If GetModems fails, the return value will be nil.

The return value is a TStringList containing an item for each modem contained in the modem detail file. The Strings portion of the item is the “friendly name” of the modem. The Objects portion is a TLmModem structure defining the modem.

ModemCapFolder**property**

```
property ModemCapFolder : string
```

↳ Defines the location of the modemcap modem database.

ModemCapFolder determines where the TapdLibModem component will find the modemcap modem database. Set ModemCapFolder to the name of the folder where modemcap was installed (default installation is in the C:\APRO\MODEMCAP folder).

SelectModem**method**

```
function SelectModem(  
    var ModemFile, ModemManufacturer, ModemName: string;  
    var LmModem : TLmModem) : Boolean;
```

↳ Displays modem selection dialog box.

Call the SelectModem method to display the modem selection dialog and select a modem definition. This method is used by the TAdModem.SelectModem method, and is documented here to allow customization of the selection process.

The ModemFile and ModemName parameters are used to filter the displayed modems, as well as to retrieve the specifications of the selected modem. If ModemFile is not empty, only modems contained in that modem detail file will be displayed. If ModemName is not empty, only modems with that name will be displayed. ModemName is case-sensitive.

If the OK button is selected to close the dialog, ModemFile will contain the name of the modem detail file where the selected modem resides; ModemManufacturer is the manufacturer, and ModemName is the name of the modem; LmModem is the TLmModem structure that defines the modem capabilities and configuration information.

See also: GetModem

TAdModem Component

The TAdModem component combines the features of libmodem-based device selection and configuration and modem control methods. TAdModem integrates the selection of the modem from the modem database and the dialog to show the current status of the modem. The TAdModem component is used to select and configure a modem and to dial or answer the line.

There are many similarities between the TAdModem and TApdTapiDevice components. The TApdTapiDevice uses the Microsoft TAPI library, and most of TAPI operates under the “black box” principle. The TAdModem can be more flexible, especially when custom modem or port configurations must be used.

The TAdModem component requires libmodem and the modemcap database. See “TApdLibModem Component” on page 449, and the appropriate help pages, for details concerning these libraries.

Modem selection

When using the TAdModem component, the appropriate modem definition must be loaded from modemcap. To do this, call the TAdModem.SelectModem method, which will display the dialog box shown in Figure 13.1.

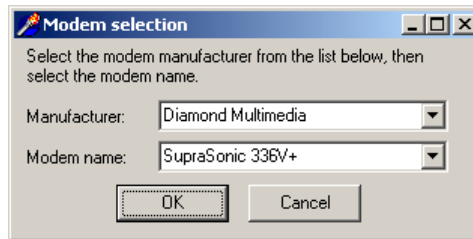


Figure 13.1: TAdModem.SelectModem dialog box.

The “Manufacturer” combo box will be filled with a list of known modem manufacturers (those that are included in modemcap). Select a manufacturer from the list and the “Modem name” combo box will be filled with a list of known modems from the appropriate modem detail file. Select the modem name from the list and click OK.

To select a modem programmatically, assign a TAdModemNameProp object to the SelectedDevice property

Connectionless connections

The TAdModem can establish a connection direct to the port through the ConfigAndOpen method. This does not establish a connection, except to the local modem. When the device has been configured, the OnModemConnect event will be generated and you can communicate with the modem.

Dialing

The TAdModem can also establish connections by dialing. To dial a number through the modem, and have the modem negotiate a connection, call the TAdModem.Dial method. The phone number to dial is passed in the ANumber parameter to the Dial method. This number is dialed without modification. If you need to enter a prefix to gain an outside line, to use a calling card, or other non-dialable use, those digits must be added to ANumber. For example, if you want to dial “555-1212” but you need to dial ‘9’ to access an outside line, ANumber would be “9 555-1212”.

Answering an incoming call

The TAdModem can also establish a connection by answering an incoming call. The TAdModem will answer incoming calls by monitoring the port for the modem’s ring indicator, which is usually a “RING” response. The AnswerOnRing property of the TAdModem component determines how many of these responses to receive before issuing the answer command to the modem. The RingWaitTimeout property determines the number of milliseconds to wait after receiving the last ring before assuming that the caller terminated the connection attempt. In the USA, the ring indicators are sent approximately every six seconds, the default RingWaitTimeout (1200 ms) will wait for about two ring cycles. If another ring indicator is not received, the internal RingCount property is reset, and the TAdModem will wait for the next AnswerOnRing ring indicators. Note that the port associated with the TAdModem must be open when waiting for an incoming call, which will prevent other processes from accessing the serial port. To share the port while passively waiting for calls, use the TApdTapiDevice.

When the modem dials the number, or answers an incoming call, the modem will attempt to establish a data connection with the remote modem. If the modem can establish a mutually supported set of connection parameters (baud, data compression, error correction, etc.) the OnModemConnect event will be generated. If the modem could not establish a connection, the OnModemFail event will be generated. The FailureCode property will contain a code indicating the reason for the failure.

Hierarchy

TComponent (VCL)	
❶ TApdBaseComponent(OOMisc)	8
TAdCustomModem(AdMdm)	
TAdModem(AdMdm)	

Properties

AnswerOnRing	FailureCode	PassthroughMode
BPSRate	LastResponse	RetryWait
CancelCall	MaxAttempts	RingWaitTimeout
ComPort	ModemCapFolder	SelectedDevice
Dialing	ModemState	StatusDisplay
DialTimeout	NegotiationResponses	❶ Version

Methods

AutoAnswer	FailureCodeMsg	SetDevConfig
CancelCall	GetDevConfig	ModemLogToString
ConfigAndOpen	SendCommand	ModemStatusMsg
Dial	SelectDevice	TranslateAddress

Events

OnModemCallerID	OnModemDisconnect	OnModemLog
OnModemConnect	OnModemFail	OnModemStatusMethods

Reference Section

AnswerOnRing

property

```
property AnswerOnRing : Byte
```

Default: 2

- ↪ The number of times the component should allow the incoming call to ring before answering it.

The AnswerOnRing property determines the number of “RING” responses that the TAdModem detects before answering the incoming call. Once the AutoAnswer method is called, internal TapdDataPackets are initialized to detect the “RING” response from the modem when an incoming call is signaled. When AnswerOnRing “RING” responses are detected, the call is answered.

Caller ID information is transmitted between the first and second rings in most countries. The AnswerOnRing property defaults to 2 to avoid problems with Caller ID information corrupting the connection negotiations.

See also: AutoAnswer, RingWaitTimeout

AutoAnswer

method

```
procedure AutoAnswer;
```

- ↪ Prepares the modem to answer a call after a specified number of rings.

After AutoAnswer sets the appropriate variables and triggers, control returns to the program and the modem component watches for incoming calls in the background. If AnswerOnRing “RING” responses are received from the modem, the call is answered. If StatusDisplay is assigned to a TapdAbstractModemStatus component, that status dialog is displayed during an answer attempt.

Once the “RING” response is received, the ModemState changes from msAutoAnswerBackground to msAutoAnswerWait. ModemState remains in msAutoAnswerWait until AnswerOnRing “RING” responses are received, when ModemState enters the msAnswerWait state and the call is answered. When a connection is established, ModemState changes to msConnected and the OnModemConnect event is generated.

Auto answer mode can be cancelled by calling `CancelCall`. The auto answer mode is cancelled regardless of whether the `TAdModem` is waiting in the background for the incoming call or the modem is currently answering the call.

Calling `AutoAnswer` does not turn on the auto answer (AA) light on an external modem. The `AutoAnswer` method does not use the auto answer feature of the modem.

See also: `CancelCall`, `ModemState`, `OnModemCallerID`, `OnModemConnect`, `OnModemFail`, `StatusDisplay`

BPSRate

read-only, run-time property

property `BPSRate` : `DWORD`

↪ The rate of the current call in bits per second.

`BPSRate` is the rate negotiated between the local and remote modems for the current call. If a call is not in progress, `BPSRate` returns zero. `BPSRate` is determined by the extended connection responses provided by the modem, and may take the connection's negotiated data compression into account. This property is available once the `OnModemConnect` event is generated.

See also: `OnModemConnect`

CancelCall

method

procedure `CancelCall`;

↪ Terminates a connection attempt or disconnects the current call.

If a dial or an answer attempt is in progress, calling this method aborts the attempt and returns the modem to its normal state.

If you call `CancelCall` while the `TAdModem` component is in auto answer mode, the modem component is taken out of auto answer mode.

`CancelCall` is the `TAdModem` universal method for terminating the current call. It can be used while waiting for an incoming call, answering an incoming call, dialing a call, or during an established connection. `TAdModem` terminates the current process, assures that the modems have disconnected, and places the modem into an idle state (not waiting for calls).

`CancelCall` returns when the connection, or connection attempt, has terminated. The `OnModemDisconnect` event is generated if a connection was present when `CancelCall` was called; if a connection was not present, no event is generated.

See also: `AutoAnswer`, `Dial`, `OnModemDisconnect`

```
property ComPort : TApdCustomComPort
```

↳ Determines the serial port to which the modem is connected.

ComPort is usually set automatically at design time to the first TApdComPort component the TAdModem finds on the form. If you need to, you can use the Object Inspector to select a different TApdComPort component.

Setting the ComPort property at run time is necessary only when using a dynamically created TApdComPort or when selecting among several TApdComPort components.

Note that some properties of the TApdComPort may be overridden with properties retrieved from modemcap when the TAdModem initializes the modem.

See also: TApdCustomComPort

ConfigAndOpen**method**

```
procedure ConfigAndOpen;
```

↳ Configures the modem and provides access to the modem without a connection.

The ConfigAndOpen method configures the modem according to the configuration settings retrieved from modemcap. Once the modem has been configured, the OnModemConnect event is generated.

ConfigAndOpen is used primarily to configure the modem to a known state where it can be used without a connection. For example, you can use ConfigAndOpen to provide terminal access to a modem for diagnostics or for faxing.

While the modem is being configured, several TApdDataPackets will be initialized to capture the responses from the modem. Once the modem has been configured, the TApdDataPackets will be removed, providing no further notification of the state of the modem.

Note that ConfigAndOpen establishes a connection to the port, not through the modem to another device.

See also: AutoAnswer, CancelCall, Dial, OnModemConnect

```
procedure Dial(ANumber : string);
```

↪ Dials the specified telephone number.

Dial initializes the modem and then dials the number specified by the ANumber parameter. When the connection is established, the OnModemConnected event is generated.

Several TApdDataPackets are initialized to detect modem responses during initialization and to monitor connection progress.

Dial returns immediately. The OnModemConnect event is generated when the appropriate connection responses have been received from the modem. The OnModemFail event is generated if the modem could not be initialized, could not detect dial tone (if that was required by the configuration), if the modems could not negotiate a mutually acceptable set of connection parameters, or if the remote party did not answer the call within DialTimeout seconds. The FailureCode property can be queried to determine the actual reason for the failure.

A dial operation can be cancelled at any time by calling CancelCall.

The ANumber parameter to this event should contain all parameters required for your modem to dial the number. This includes any prefixes required by your phone system to obtain an outside line. For example, if your phone system requires you to dial '9' to get an outside line, the call to Dial would include that digit:

```
ApXModem1.Dial('9 555-1212');
```

See also: AutoAnswer, CancelCall, ConfigAndOpen, DialTimeout, OnModemConnect, OnModemFail, StatusDisplay

DialTimeout**property**

```
property DialTimeout : Integer
```

Default: 60

↪ The number of seconds to wait for a connection after dialing the number.

When a dial attempt begins, the TAdModem component allows DialTimeout seconds for a connection result to be received from the modem. The timeout countdown starts when the Dial method is called. If the connection responses are not received within this time, a timeout occurs, the operation is cancelled and the OnModemFail event is generated.

See also: Dial, FailureCode, OnModemFail

FailureCode

read-only, run-time property

```
property FailureCode : Integer
```

↪ The numerical code indicating the last failure.

FailureCode is the result of the last AutoAnswer, ConfigAndOpen, or Dial method that was called. FailureCode indicates the nature of the failure.

This property is used primarily in the OnModemFail event handler to determine the reason for the failure.

See also: AutoAnswer, ConfigAndOpen, Dial, FailureCodeMsg, OnModemFail

FailureCodeMsg

method

```
function FailureCodeMsg(const FailureCode : Integer) : string;
```

↪ Converts a numerical FailureCode into a string describing the error.

The FailureCodeMsg method converts a FailureCode into a human-readable string describing the error.

See also: FailureCode, OnModemFail

GetDevConfig

method

```
function GetDevConfig : TLmModem;
```

↪ Returns the modem configuration structure.

The modemcap database file contains a list of modems and their configuration. The GetDevConfig method returns the currently active configuration structure for the selected modem.

This method can be used to confirm the modem configuration prior to use, or to make changes to the current configuration.

See also: SetDevConfig

ModemCapFolder

property

```
property ModemCapFolder : string
```

Default: Empty string

↪ The name of the directory where the modemcap modem database has been installed.

The modemcap database file contains a list of modems and their configuration strings. This folder is used when SelectDevice is called to display a list of modems from which the user can choose. ModemCapFolder is also used at design time to display the list of modems when the SelectedDevice property is modified.

See also: SelectDevice, SelectedDevice

ModemLogToString

method

```
function ModemLogToString(  
    const LogCode : TAdModemLogCode) : string;
```

↪ Returns an English string describing an error code.

The ModemLogToString method references the string resource for the log code and returns a text description of the code. This method is used primarily in the OnModemLog event to display a text description of the log event.

See also: OnModemLog

ModemState

read-only, run-time property

```
property ModemState : TAdModemState
```

```
TAdModemState = (msUnknown, msIdle, msInitializing,  
    msAutoAnswerBackground, msAutoAnswerWait, msAnswerWait,  
    msDialWait, msDialCycle, msConnectWait, msConnected,  
    msHangup, msCancel);
```

↪ The current state of the TAdModem component.

Default: msUnknown

ModemState is used internally to track modem responses and controlling the state of the TAdModem component for configuration, dialing, and answering. This property is made public for possible use of status routines.

ModemState values are:

Value	Description
msUnknown	Hasn't been or couldn't be initialized.
msIdle	Idle and ready.
msInitializing	Starting initialize process.
msAutoAnswerBackground	Autoanswer mode -- no rings received.
msAutoAnswerWait	Autoanswer mode -- waiting for Nth ring.
msAnswerWait	Answering call -- waiting for connect.
msDialWait	Dialing call -- waiting for connect.
msConnectWait	Connect in progress -- waiting for optional data.
msConnected	Done with connect process.
msHangup	Starting hangup process.
msCancel	Starting cancel process.

See also: ModemStateMsg, OnModemStatus

NegotiationResponses	read-only, run-time property
-----------------------------	-------------------------------------

property NegotiationResponses : TStringList

↳ Contains the modem's reported negotiated connection parameters.

During a connection attempt, either answering or dialing, the modem may return several lines of text describing the negotiated connection parameters. These responses may indicate the error correction, data compression, or other features that the modem negotiated. These features are informative in nature and are provided in the NegotiationResponses property.

During an AutoAnswer operation, NegotiationResponses will contain all modem responses from the TAdModem sending the answer command to the final connection response. During a Dial operation, NegotiationResponses will contain all modem responses from the TAdModem sending the dial command to the final connection response.

See also: AutoAnswer, Dial

```
property OnModemCallerID : TModemCallerIDEvent  
  
TModemCallerIDEvent = procedure(  
    Modem : TAdCustomModem; ID, IDName : string) of object;
```

↳ Defines an event handler that is generated when Caller ID information is detected.

If the modemcap structure supports Caller ID configurations, the modem will be appropriately initialized to respond to the Caller ID signals provided for incoming calls. Internal TApdDataPackets will be initialized to detect the Caller ID responses from the modem. When Caller ID responses are detected, the OnModemCallerID event will be generated.

Modem is the TAdCustomModem component that generated the event. ID is the identification reported by the Caller ID signal for the number field (usually the caller's phone number). IDName is the identification reported by the Caller ID signal for the name field (usually the caller's subscribed name).

In most countries, the telephone company supplies the Caller ID information between the first and second rings. A typical format is shown in the following:

DATE: MM/DD/YY<CR><LF>

TIME: HH:MM:SS<CR><LF> {24-hour format}

NUMBER: {variable content}<CR><LF>

NAME: {variable content}<CR><LF>

Some telephone companies provide information in a different order, different format, or even different information entirely. The TAdModem will detect the NUMBER and NAME responses. Additional responses can be gathered using your own TApdDataPackets.

```
property OnModemConnect : TModemNotifyEvent
```

```
TModemNotifyEvent = procedure(  
    Modem : TAdCustomModem) of object;
```

↳ Defines an event handler that is generated when a connection is established.

The OnModemConnect event is generated when dialing or answering after the modem returns the connection response. This event is also generated after a call to ConfigAndOpen once the modem has been configured.

Modem is the TAdCustomModem component that generated the event. No other parameters are provided.

The TAdModem will watch for the connection responses as defined by the modemcap entry for the selected modem and by monitoring the DCD signal.

See also: AutoAnswer, ConfigAndOpen, Dial, OnModemDisconnect

```
property OnModemDisconnect : TModemNotifyEvent
```

```
TModemNotifyEvent = procedure(  
    Modem : TAdCustomModem) of object;
```

↳ Defines an event handler that is generated when a connection is terminated.

The OnModemDisconnect event is generated when the TAdModem detects the connection has been terminated. When the TAdModem detects that the connection has been established, a status trigger monitoring for changes in DCD is installed. When DCD is lowered, the connection is considered terminated, and the OnModemDisconnect event is generated. This event is also generated when the CancelCall method successfully terminates the connection.

Modem is the TAdCustomModem that generated the event. No other parameters are provided.

Note that some devices and protocols routinely toggle DCD. For these situations, consult the device and protocol documentation for details on detecting connection termination.

See also: AutoAnswer, ConfigAndOpen, Dial, OnModemConnect

```
property OnModemFail : TModemNotifyEvent
```

```
TModemNotifyEvent = procedure (  
    Modem : TAdCustomModem) of object;
```

↳ Defines an event handler that is generated when a modem or connection failure is detected.

The OnModemFail event is generated if the modem could not be initialized, or if a connection could not be established. When dialing, this event is generated if the modem could not detect dial tone (if the configuration required dial tone), if the modems could not negotiate a mutually acceptable set of connection parameters, or if the remote party did not answer the call within DialTimeout seconds. When answering, this event is generated if the modems could not negotiate a mutually acceptable set of connection parameters, or if the connection attempt timed out.

Modem is the TAdCustomModem that generated the event. No other parameters are provided. The reason for the failure can be obtained from the FailureCode property.

See also: AutoAnswer, ConfigAndOpen, Dial, FailureCode, FailureCodeMsg

OnModemLog**method**

```
property OnModemLog : TModemLogEvent
```

```
TModemLogEvent = procedure(  
    Modem : TAdCustomModem; LogCode : TApdModemLogCode) of object;
```

↳ Defines an event handler that is generated at designated points during a dial or answer attempt.

The primary purpose of this event is to give the application a chance to log auditing information about telephone calls and whether they succeed or fail. This event is intended primarily for high-level logging, not to determine program flow.

Modem is the TAdCustomModem that generated the event. LogCode is the TAdModemLogCode that described the event being logged.

TAdModemLogCode can have one of the following values:

Value	Description
mlNone	None
mlDial	Dialing
mlAutoAnswer	Initiated AutoAnswer
mlAnswer	Answering an incoming call
mlConnect	Connected
mlCancel	Call cancelled
mlBusy	Called number was busy
mlConnectFail	Connection attempt failed

See also: ModemLogToString

OnModemStatus

method

```
property OnModemStatus : TModemStatusEvent
TModemStatusEvent = procedure(
    Modem : TAdCustomModem; ModemState : TAdModemState) of object;
```

↳ Defines an event handler that is generated when the state of the component changes.

The OnModemStatus event is generated periodically when the ModemState property is changed. This event indicates when the state of the modem or connection is changed.

Modem is the TAdCustomModem that generated the event. ModemState is the new state of the component. See the ModemState definition for a list of possible ModemState values and their meanings.

See also: ModemState, ModemStatusMsg

RingCount

run-time, read-only property

```
property RingCount : Byte
Default: 0
```

↳ The number of ring signals detected for the current call.

When the TAdModem is in AutoAnswer mode, the RingCount property indicates the number of ring signals that have been detected. When a ring signal has not been detected, RingCount will be 0. When RingCount equals AnswerOnRing, the call will be answered.

See also: AutoAnswer, RingWaitTimeout

property RingWaitTimeout : DWORD

Default: 1200

- ↪ Determines the number of milliseconds to wait before resetting an AutoAnswer attempt.

Most phone companies generate the ring indicator signal every six seconds. When the TAdModem is in AutoAnswer mode, consecutive ring signals within RingWaitTimeout milliseconds are considered to be from the same call, and will increment the internal ring counter. If RingWaitTimeout milliseconds elapse after the ring signal, the caller is assumed to have aborted the call, and the internal ring counter is reset.

See also: AutoAnswer, RingCount

function SelectDevice : Boolean;

- ↪ Displays the modem selection dialog.

SelectDevice displays the modem selection dialog, which lists the modems defined in the modemcap database. This method will return True if a device is selected, and False if the modem selection dialog box is cancelled.

To select a modem from modemcap, the manufacturer and modem name must be selected. The “Manufacturer” combo box will contain a list of all manufacturers that are included in modemcap. When a manufacturer is selected, the “Modem name” combo box will contain a list of all modems from that manufacturer in modemcap. The OK button will be disabled until this requirement is satisfied. When the “OK” button is clicked, this method returns True and the SelectedDevice property will be updated to reflect the selected device. If the Cancel button is clicked, this method returns False and SelectedDevice is not updated.

If the AutoAnswer, ConfigAndOpen, or Dial methods are called without a valid modem specified in SelectedDevice, the modem selection dialog will be displayed. If a modem is not selected, the ecNoSelectedDevice exception will be raised.

Selecting a new modem configuration through the SelectedDevice property of the SelectDevice method while a connection is established will raise the ecModemBusy exception.

See also: GetDevConfig, SelectedDevice, SetDevConfig

```
property SelectedDevice : TAdModemNameProp

TAdModemNameProp = class(TPersistent)
    published
        property Manufacturer : string;
        property Name : string;
end;
```

↪ The currently selected modem.

SelectedDevice displays the modem manufacturer, model, and name of the modem that has been selected through the SelectDevice method. The properties of the TAdModemNameProp are read-only at design time and are available to determine which modemcap structure will be used.

To change modems configurations at run time, you may either call the SelectDevice method to display the device selection dialog or assign a TAdModemNameProp class to this property.

Selecting a new modem configuration through the SelectedDevice property or SelectDevice method while a connection is established will raise the ecModemBusy exception.

See also: GetDevConfig, SelectDevice, SetDevConfig

```
function SendCommand(const Command : string) : Boolean;
```

↪ Provides a convenient method to send a custom command to the modem.

The SendCommand method can be used to send a modem command to the modem. SendCommand returns when the modem either responds to the command, or times out. If the modem returns a successful response, the return value will be True. If the modem returns a failure response, the return value will be False and the FailureCode property will indicate the type of failure based on the response.

See also: FailureCode, FailureCodeMsg

```
procedure SetDevConfig(const Config : TLmModem);
```

↳ Forces a modem configuration structure.

The modemcap database file contains a list of modems and their configuration. The SetDevConfig method forces a new configuration structure, which will be used in subsequent calls to the AutoAnswer, ConfigAndOpen, and Dial methods.

This method is intended to be used after the GetDevConfig method provides the default configuration structure for the modem. Additional configuration settings, or port options, can be defined for the session. Changes made through SetDevConfig will remain in effect until the SelectedDevice is changed or the application is terminated. This method will not change the modem definition in the modemcap database.

See also: GetDevConfig

```
property StatusDisplay : TAdAbstractModemStatus
```

↳ The status dialog used to provide visual status indications.

During AutoAnswer or Dial operations, the status dialog specified by the StatusDisplay property can provide visual feedback to indicate what the modem is doing.

The provided TAdModemStatus component encompasses several status indicators to illustrate the state of the connection. A custom status display can be used instead of the TAdModemStatus component. See the description of the TAdModemStatus component for details.

See also: OnModemStatus

TAdModemStatus Component

The TAdModemStatus component is a descendant of the TAdAbstractModemStatus class that implements a standard modem status dialog. To use it, create an instance of the component and assign it to the StatusDisplay property of a TAdModem component. The TAdModemStatus component displays common status indicators as well as a detailed history of what the TAdModem is doing. The TAdModemStatus component provides a Cancel button, which will cancel an AutoAnswer or Dial attempt in progress.

Figure 13.2 shows the dialog box that the TAdModemStatus component displays.

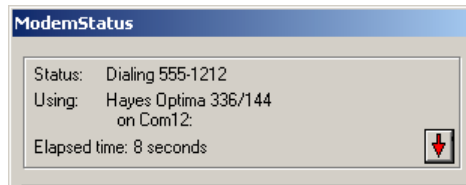


Figure 13.2: TAdModemStatus dialog box.

The Status label displays a string describing the current TAdModem.ModemState property value, and indicates what the TAdModem is doing. The Using label displays the TAdModem.SelectedDevice.Name property value and the serial port that is being used. The Elapsed time label displays the number of seconds that have elapsed since the current operation began.

The Cancel button will cancel the operation by calling the CancelCall method of the TAdModem component.

The down arrow button will display a more detailed status dialog, as shown in Figure 13.3.

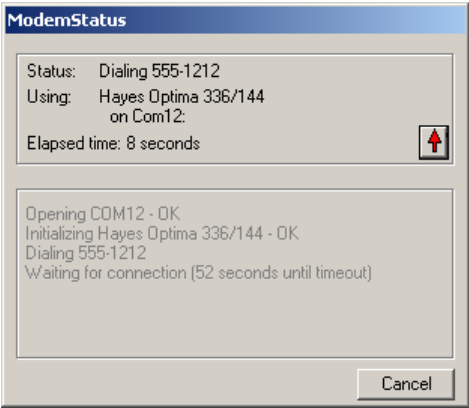


Figure 13.3: TAdModemStatus expanded dialog box.

To return to the compact display, click the up arrow button.

To create a custom status dialog box, the OnModemStatus event of the TAdModem component can be used to update a separate form in your application, or to update a status bar. You can also create a new component descending from the TApdAbstractModemStatus class. The UpdateStatus method of the TApdAbstractModemStatus class is automatically called by the TAdModem component whenever the OnModemStatus event of the TAdModem component is generated. This method must be implemented to update your custom dialog. See the AxMdmDlg.pas unit for details on creating a custom dialog box.

Hierarchy

TComponent (VCL)

- ❶ TApdBaseComponent(OOMisc) 8
 - TApdAbstractModemStatus(AdMdmDlg)
 - TAdModemStatus(AdMdmDlg)

Properties

- Caption
- Started
- StatusDialog
- ❶ Version

Methods

- UpdateDisplay

Reference Section

Caption

property

```
property Caption : string
```

Default: "Modem status"

↪ Determines the caption for the status dialog box.

The Caption property sets the `StatusDialog.Caption` property. If your window manager supports title bars on dialogs, changing this property will change the caption displayed in the title bar. If your window manager does not support title bars on dialog boxes, this property has no visible effect.

See also: `StatusDialog`

Started

run-time, read-only property

```
property Started : Booleanproperty Started : Boolean
```

Default: False

↪ Indicates whether the status dialog box has been created and initialized.

The `TForm` descendant specified by the `StatusDialog` component is created and initialized when the `UpdateDisplay` method is called for the first time. `Started` indicates whether `StatusDialog` has been created. When the `StatusDialog` is no longer needed, the form is destroyed and `Started` is set to False.

`Started` is used by the `TAdModemStatus` dialog, it is available, but not required, for custom status dialogs.

See also: `StatusDialog`, `UpdateDisplay`

```
property StatusDialog : TForm
```

↪ Determines the status form to display.

The TAdModemStatus component defines the TAdModemStatusDialog for displaying modem status. The dialog box that the TAdModemStatus component provides is illustrated and described in the overview section on page 461.

The UpdateDisplay method updates StatusDialog to provide status indications. The dialog properties can be changed at run time by referencing the StatusDialog property. For example, changing the TAdModemStatus.StatusDialog.BorderStyle property will change the dialog's BorderStyle.

To provide a different dialog, create a new component, descend from the TApdAbstractModemStatus class, and define a new TForm descendant for your display.

See also: UpdateDisplay

UpdateDisplay**method**

```
procedure UpdateDisplay(  
    Modem : TAdCustomModem; const Str0, Str1, Str2, Str3 : string);
```

↪ Updates the StatusDialog.

The UpdateDisplay method updates the dialog specified by StatusDialog. When creating a custom display, this method must be overridden to update the controls available on the custom form.

Modem is the TAdCustomModem whose status has changed. Str0, Str1, Str2 and Str3 are strings that can be used to provide additional status indicators for the display.

See also: StatusDialog

Chapter 14: File Transfer Protocols

Many communications applications need to transfer files or other large amounts of data from one machine to another. This could be accomplished by having the sender call `PutBlock` repeatedly and the receiver call `GetBlock` correspondingly. However, the application would have a tremendous amount of detail work still to do. It would need logic to transfer file name and size information, to check for and recover from transmission errors, to handle file I/O, etc., etc.

That's why Async Professional provides standard, tested, reliable, high performance file transfer protocols. The term "protocol" means that both sides of the communication link behave in a clear, well-defined manner following agreed-upon rules. The rules vary among the different protocols and some protocols offer more control and features than others. At a minimum, each protocol handles file I/O and serial port I/O and checks for errors. Some protocols also include error correcting logic, multi-file transfers, and automatic recovery after partial file transfers.

Async Professional offers the most widely used industry standard file transfer protocols, as shown in Table 14.1.

Table 14.1: *Available Async Professional file transfer protocols*

Protocol	Component	Description
Xmodem	TApdProtocol	128 byte blocks with checksum block checking. See "Xmodem" on page 501 for more information.
XmodemCRC	TApdProtocol	128 byte blocks with CRC block checking. See "Xmodem" on page 501 for more information.
Xmodem1K	TApdProtocol	1024 byte blocks with CRC block checking. See "Xmodem" on page 501 for more information.
Xmodem1KG	TApdProtocol	Streaming Xmodem1K. See "Xmodem" on page 501 for more information.
Ymodem	TApdProtocol	1024 byte blocks, batch. See "Ymodem" on page 504 for more information.
YmodemG	TApdProtocol	Streaming Ymodem. See "Ymodem" on page 504 for more information.
Zmodem	TApdProtocol	1024 byte blocks, batch, streaming, restartable. See "Zmodem" on page 507 for more information.

Table 14.1: *Available Async Professional file transfer protocols (continued)*

Protocol	Component	Description
Kermit	TApdProtocol	80 byte blocks, batch, with long blocks and windowing. See “Kermit” on page 513 for more information.
ASCII	TApdProtocol	ASCII stream with inter-character and inter-line delays. See “ASCII” on page 519 for more information.
FTP	TApdFTPClient	An internet file transfer protocol. See “FTP” on page 521 for more information.

Three related classes are also described in this chapter. `TApdAbstractStatus` defines a mechanism by which the protocol can report its status (percent completion, transfer rate, etc.) to the user. `TApdProtocolStatus` derives from `TApdAbstractStatus` to present protocol status in a particular style. `TApdProtocolLog` is a small component that writes to a log file the status of each file transferred by an associated `TApdProtocol` component.

General Issues

The Async Professional protocol engine is implemented in a group of units with names like AwAbsPcl (abstract protocol services), AwXmodem (Xmodem protocol), AwKermit (Kermit protocol), etc. These units are linked directly into your application. The TApdProtocol component, implemented in the AdProtcl unit, is a shell around the protocol engine.

The protocol engine works in the background by using Async Professional timer, data available, and data match triggers. Windows can continue with other tasks while a file transfer is in progress as long as the other tasks yield properly for other events.

The following subsections document issues that arise for all types of file transfers that use the Async Professional protocol engine. Note that the following general issues do not relate to the TApdFTPClient component.

Buffer sizes

When a TApdComPort component is created, you specify the input and output buffer sizes that the Windows communications driver is to use. An interactive communications process such as a terminal window can safely use small buffer sizes, say 4K bytes for input and output. A protocol file transfer requires larger buffers. Why? Consider how a file transfer program is likely to be used under Windows.

Once a file transfer starts, it is likely that the user will work in another window until the file transfer finishes. Because the transfer is running as a background application, it is at the mercy of other Windows tasks. Many Windows applications and built-in Windows operations can hog the CPU to an extent that prevents the background transfer from succeeding.

To different degrees, all file transfer protocols are time-critical. They must respond to incoming events in a timely fashion, usually within a few seconds. If they fail to respond in the required time, the remote protocol software times out and repeats the failed operation. Such timeouts and repetitions at best reduce the protocol transfer rate and at worst can cause the protocol to fail.

In practice, it takes a very ill-behaved program or unusual user (e.g., someone who spends 30 seconds to move a window) to cause most protocols to fail. But this can happen and your application should do whatever it can to minimize the chances.

One of the things your program can do is use a large input buffer. The Windows communications driver continues to receive data and store it in the input buffer even if the associated application program isn't getting any time to run. With a 30K byte input buffer and a data rate of 1600 characters per second, the buffer can hold 19 seconds worth of

incoming data before overflowing. When the application eventually regains control it processes all received data before relinquishing control. The input buffer is then ready to hold another 19 seconds worth of data.

A large output buffer is also valuable when transmitting files. Streaming protocols such as YmodemG and Zmodem, and even Kermit to a lesser degree, typically transmit until they fill the output buffer, then relinquish control. They don't regain control until the buffer drains enough to hold another data block and Windows can process the associated status trigger message.

If the status trigger message is delayed because another Windows application didn't yield, the Windows communications driver continues to transmit the data remaining in the output buffer. Using the same numbers as the input buffer example, the driver can transmit independently for up to 19 seconds before running out of data.

When transmitting files under protocol control, the output buffer size must be at least 2078 bytes (i.e., 2K plus 30 bytes). This size is required because the protocols send protocol data to the port by copying the entire block into the output buffer. If the block size is 1024 bytes and every character is escaped (preceded by a special character that prevents the link from misinterpreting the data as a control sequence), the output buffer must hold 2048 bytes. The extra 30 bytes is a safety margin provided so that the protocol doesn't need to check for output buffer space for block check characters and a subsequent header, if there is one. The protocols use the direct copy approach because it's much faster than calling PutChar for each character in the block.

In summary, your application should use input and output buffers that are as small as possible. However, the smallest possible buffer might actually be quite large, even as large as 32K, for applications that are designed to run in the background.

Protocol events

The protocol component generates several kinds of events. General descriptions of these events follow:

OnProtocolAccept

```
procedure(CP : TObject; var Accept : Boolean;  
          var FName : string) of object;
```

Generated as soon as the protocol window knows the name of an incoming file. This provides an opportunity to accept or reject the file, or to change its name. See the OnProtocolAccept event on page 544 for details about how to handle this event. Also see "AcceptFile processing" on page 495.

OnProtocolError

```
procedure(CP : TObject; ErrorCode : SmallInt) of object;
```

Generated when an unrecoverable error occurs. Recoverable errors do not generate this message because such errors are a routine part of transferring files and the failed operation is retried automatically. See OnProtocolError (page 545) for details. Also see “Error handling” on page 488.

OnProtocolFinish

```
procedure(CP : TObject; ErrorCode : SmallInt) of object;
```

Generated after all files have been transferred or after the protocol terminates due to an unrecoverable error. This event also sends the final result code of the protocol.

OnProtocolLog

```
procedure(CP : TObject; Log : Word) of object;
```

Generated at the start and end of transferring each file. This provides an opportunity to log the status of each file transfer. See OnProtocolLog (page 546) for details. Also see “Protocol logging” on page 493.

OnProtocolNextFile

```
procedure(CP :TObject; var FName : string) of object;
```

Generated whenever it is time to transmit another file. By default this message is handled by the protocol component, which returns the name of the next file that matches the FileMask property. Programs can intercept this event to provide other ways to choose the next file. See OnProtocolNextFile (page 547) for details. Also see “NextFile processing” on page 494.

OnProtocolStatus

```
procedure(CP : TObject; Options : Word) of object;
```

Generated at regular intervals so that programs can display the progress of the protocol. See OnProtocolStatus (page 548) for details. Also see “Protocol status” on page 489.

Aborting a protocol

There will certainly be times when a protocol in progress must be canceled (e.g., when something goes wrong at the remote computer or the user simply decides not to continue the transfer). Async Professional protocols provide for this situation with the CancelProtocol method.

To cancel any protocol simply call the CancelProtocol method of the TApdProtocol component. This method sends an appropriate cancel sequence to the remote computer and terminates. The protocol component remains intact, ready to handle additional protocol transfers.

When protocol transfers take place over a modem link it is a good idea to monitor the DCD (data carrier detect) line and abort the protocol if carrier is lost. The DCD line goes high when modems first connect and remains high until one of the modems hangs up. Occasionally, line noise or other disturbances in the telephone network break the connection between the modems, causing DCD to go low.

Some protocols quickly detect that the remote isn't acknowledging after the connection is broken. These protocols soon abort with an error code of `ecTimeout` or `ecTooManyErrors`. By contrast, streaming protocols can take a very long time to notice that the connection is broken because they don't require acknowledgments.

The protocol component provides an option to handle dropped carrier automatically. Set the `AbortNoCarrier` property to `True` before calling `StartTransmit` or `StartReceive` and the protocol engine automatically aborts if the DCD line is not high at any point during the protocol. The protocol cancels itself immediately and generates an `OnProtocolFinish` event with an error code of `ecAbortNoCarrier`.

Using the `AbortNoCarrier` property is better than checking DCD and calling `CancelProtocol` in your own code. When you do this, the protocol engine sends a cancel sequence to the remote computer. If hardware flow control is enabled and the modem has lowered the DSR or CTS signals as well as DCD, the protocol waits several seconds before deciding it can't send the cancel command, leading to an unnecessary delay for the application. The `AbortNoCarrier` option prevents the protocol engine from sending the cancel sequence, so the protocol stops immediately.

Error handling

All protocol transfers are subject to errors, including parity errors, files not found, and other file I/O errors. Whenever possible the protocol window handles errors internally by retrying an operation or requesting the remote computer to retry. At some point, however, it determines that the situation is unrecoverable and generates an `OnProtocolError` event. An application should include a handler for this event. Following is a simple example:

```
procedure Form1.ApdpProtocol1ProtocolError(  
    CP : TObject; ErrorCode : SmallInt);  
begin  
    ShowMessage('Fatal protocol error: ' + ErrorMsg(ErrorCode));  
end;
```

This event handler's sole task is to display a message about the error. `ErrorMsg` is a function from the `AdExcept` unit that returns an English string for any `Async Professional` error code.

See "Error Handling and Exception Classes" on page 900 for additional information about errors.

Protocol status

A protocol transfer can last a few seconds or several hours depending on the size and speed of the transfer. Because the protocol component handles the details of the transfer from start to finish, your application's code is not executing during this entire time. You and your users certainly want to know what's happening as the transfer progresses, so Async Professional provides a hook for your application to regularly regain control during this time.

During a protocol transfer the protocol window frequently generates an `OnProtocolStatus` event. This gives your code the opportunity to monitor and display the progress of the protocol. Following are code fragments that illustrate how:

```
TForm1 = class(TForm)
...
FN: TLabel;
BT: TLabel;
BR: TLabel;
...
end;

procedure TForm1.ApProtocol1ProtocolStatus(
  CP : TObject; Options : Word);
begin
  case Options of
    apFirstCall :
      ...do setup stuff
    apLastCall :
      ...do cleanup stuff
  else
    {show status}
    FN.Caption := ApProtocol.FileName;
    BT.Caption := IntToStr(ApProtocol.BytesTransferred);
    BR.Caption := IntToStr(ApProtocol.BytesRemaining);
  end;
end;
```

The method named `ApProtocol1ProtocolStatus` handles the `OnProtocolStatus` event by updating a form at each call. The `Options` parameter passed to this routine can take on two special values:

```
apFirstCall = 1;
apLastCall  = 2;
```

`Options` is set to `apFirstCall` the first time the protocol generates the event after being started by `StartTransmit` or `StartReceive`. `Options` is set to `apLastCall` the last time it generates the event, when the protocol is finished. `Options` equals zero for all other times.

The rest of the information about protocol progress is obtained by reading the values of various `TApdProtocol` properties, including:

`BlockCheckMethod` - the type of block check calculation used by the protocol. See the reference section entry for this property (page 530) for a complete description of all block check types.

`BlockErrors` - the number of errors for the current block. This is the number of times the protocol has unsuccessfully tried to transmit or receive the current block. It is reset to zero when the block is finally accepted.

`BlockLength` - the current transfer block length. Although this value is usually static, some protocols modify the length of the block on the fly. Zmodem in particular reduces the block length after several block errors in a row and raises it again after several good blocks.

`BlockNumber` - the number of blocks transmitted so far. This is obtained by dividing the number of bytes transferred by the current block length, so it will change if the block length changes.

`BytesRemaining` - the size of the file minus `BytesTransferred`. When the file size isn't known, `BytesRemaining` returns zero.

`BytesTransferred` - the number of bytes transmitted or received so far. When transmitting, this number is sometimes only an estimate. The uncertainty comes from the fact that the protocol window doesn't know when a particular byte has actually been transferred. `BytesTransferred` is the number of bytes the protocol window has transferred to the output buffer of the communications driver, minus the number of bytes that the driver reports are currently in the buffer. Unfortunately, this calculation is still imperfect because it's impossible to know how much of the output buffer holds actual file data and how much holds overhead characters needed by the protocol. Each protocol has a few simple rules it uses to estimate this proportion, which in practice yield good estimates.

`ElapsedTicks` - the number of ticks elapsed since the protocol started. In order to provide accurate CPS values, the protocol engine doesn't start the timer until it receives the first block from the remote computer.

`FileDate` - the date and time of the file being transmitted or received. If the protocol does not support this feature, `FileDate` returns zero.

`FileLength` - the size of the file being transmitted or received. For transmitted files the file size is always known. For received files the file size is known only if the protocol supports this feature and the receiver has received this information. If the file size is not known, `FileLength` returns zero.

`FileName` - the fully qualified name of the file that is being received or transmitted. When receiving with a protocol that does not transfer the file name, `FileName` simply returns the name previously assigned to it.

InitialPosition - used only for resumed file transfers using the Zmodem protocol. To display an accurate transfer rate (CPS, or character per second, rate), status routines for these protocols must subtract InitialPosition from BytesTransferred to obtain the actual number of bytes transferred during this session. If this is not a resumed file transfer, InitialPosition returns zero.

ProtocolError - the code of the last error encountered by the protocol. This equals zero except for the first status call after an error is encountered. See “Error Handling and Exception Classes” on page 900 for additional information.

ProtocolStatus - a code that indicates the current state of the protocol. Table 14.2 shows all of the possible values. The usual status value is psOK, which means that the protocol is operating normally. Other status values indicate recoverable error conditions, protocol resume conditions, protocol start-up states, and internal protocol states. Fatal protocol errors are not represented by protocol states; they are first reported via the OnProtocolError event. However, it is possible that a final status message might be sent after a fatal error occurs.

Table 14.2: Possible values for the ProtocolStatus property of TApdProtocol

Status Code	Value	Explanation
psOK	0	Protocol is OK.
psProtocolHandshake	1	Protocol handshaking in progress.
psInvalidDate	2	Bad date/time stamp received and ignored.
psFileRejected	3	Incoming file was rejected.
psFileRenamed	4	Incoming file was renamed.
psSkipFile	5	Incoming file was skipped.
psFileDoesntExist	6	Incoming file doesn't exist locally and is skipped.
psCantWriteFile	7	Incoming file skipped due to Zmodem options.
psTimeout	8	Timed out waiting for something.
psBlockCheckError	9	Bad checksum or CRC.
psLongPacket	10	Block too long.
psDuplicateBlock	11	Duplicate block received and ignored.
psProtocolError	12	Error in protocol.
psCancelRequested	13	Cancel requested.
psEndFile	14	At end of file.

Table 14.2: *Possible values for the ProtocolStatus property of TApdProtocol (continued)*

Status Code	Value	Explanation
psSequenceError	15	Block was out of sequence.
psAbortNoCarrier	16	Aborting on carrier loss.
psGotCrcE	17	Got Zmodem CrcE packet.
psGotCrcG	18	Got Zmodem CrcG packet.
psGotCrcW	19	Got Zmodem CrcW packet.
psGotCrcQ	20	Got Zmodem CrcQ packet.

ProtocolType - the protocol type, which is one of ptXmodem, ptXmodemCRC, ptXmodem1K, ptXmodem1KG, ptYmodem, ptYmodemG, ptZmodem, ptKermit, or ptAscii.

TotalErrors - the number of errors encountered since the current file was started. It is reset only when a new file is started.

Various properties that describe the option settings for the protocol may also be used within the status routine. These include HonorDirectory, IncludeDirectory, RTSLowForWrite, AbortNoCarrier, and other options that are specific to particular protocols.

The StatusInterval property, which defaults to 18, is the maximum number of ticks between OnProtocolStatus events. The protocol generates an OnProtocolStatus event after every significant event (received a file name, received a complete block, etc.) or after at most StatusInterval ticks.

Async Professional includes a mechanism for providing an automatic protocol status display without programming, through the TApdProtocol's StatusDisplay property:

```
property StatusDisplay : TApdAbstractStatus
```

The TApdAbstractStatus class is described in more detail beginning on page TApdAbstractStatus Class. For each OnProtocolStatus event the protocol checks whether StatusDisplay is assigned. If it is, the protocol calls the UpdateDisplay method of StatusDisplay to update the display. It then calls the OnProtocolStatus event if one was implemented.

When a protocol component is created, either dynamically or when dropped on a form, it searches the form for a TApdAbstractStatus instance and updates the StatusDisplay property with the first one it finds. It also fills in StatusDisplay when a TApdAbstractStatus component is added to the form. StatusDisplay can also be modified at design time or run time to point to a TApdAbstractDisplay component other than the one assigned automatically.

Async Professional also provides a non-abstract implementation of a `TApdAbstractStatus` class called the `TApdProtocolStatus` component. If you drop one of these on your form it will automatically display full status information during all file transfers. See page 582 for more information.

Protocol logging

File transfer is often an automated process. For example, an application might send all of the day's transaction files to a remote computer during the night. In this case the application would also keep a record of the files that were successfully transmitted and those that weren't.

The Async Professional protocol logging feature is ideal for this kind of application. It provides the opportunity to log information about each received or transmitted file and whether the transfer succeeded.

To support logging, the protocol component generates an `OnProtocolLog` event at the start and end of each file transfer. The event passes a parameter that identifies the current logging action. Here is an example that handles this event:

```
procedure TForm1.ApdProtocol1ProtocolLog(
  CP : TObject; Log : Word);
begin
  case Log of
    lfReceiveStart,
    lfTransmitStart :
      CurrentFile.Caption := ApdProtocol1.FileName;

    lfReceiveOk,
    lfTransmitOk :
      GoodList.Items.Add(ApdProtocol1.FileName);

    lfReceiveFail,
    lfTransmitFail :
      BadList.Items.Add(ApdProtocol1.FileName);

    lfReceiveSkip
    lfTransmitSkip :
      SkipList.Items.Add(ApdProtocol1.FileName);
  end;
end;
```

The example shows every possible logging value. The meaning of the various logging conditions should be clear except for perhaps `lfReceiveSkip` and `lfTransmitSkip`. `lfReceiveSkip` is generated by any of the protocols when an incoming file is rejected by the

handler for the `OnProtocolAccept` event. `IfTransmitSkip` is generated only in a Zmodem transfer when the remote receiver indicates that it does not want to receive the file being transmitted by the logging application.

This example uses a `TLabel` control named `CurrentFile` to display the name of the file currently being transmitted. As files are transmitted or received it updates three `TListBox` components: `GoodList` for all successful transfers, `BadList` for all failed transfers, and `SkipList` for all skipped files.

The logging routine isn't limited to just writing status information. It also can take care of file-related start-up and cleanup activities. One example of this is to delete partially received files. You would probably want to do this for all protocols except Zmodem, which can resume failed transfers from the point of the error without having to retransmit the entire file.

Async Professional includes a mechanism for providing automatic protocol logging without programming, through the `ProtocolLog` property of `TApdProtocol`:

```
property ProtocolLog : TAPdProtocolLog
```

The `TApdProtocolLog` component is described in more detail beginning on page 583. `TApdProtocolLog` Component. For each `OnProtocolLog` event the protocol checks whether `ProtocolLog` is assigned. If it is, the protocol calls `UpdateLog` to write information to the log file. It then calls the `OnProtocolLog` event if one was implemented.

When a protocol component is created, either dynamically or when dropped on a form, it searches the form for a `TApdProtocolLog` instance and updates the `ProtocolLog` property with the first one it finds. It also fills in `ProtocolLog` when a `TApdProtocolLog` component is added to the form. `ProtocolLog` can also be modified at design time or run time to point to a `TApdProtocolLog` component other than the one assigned automatically.

NextFile processing

Several of the protocols provided by Async Professional can transmit and receive batches of files. When the protocol is ready to transmit a new file it generates an `OnProtocolNextFile` event. The event handler responds by returning the name of the next file to transmit, or an empty string to terminate the batch.

In most cases, you don't need to worry about handling this event because the protocol component does so itself. The protocol component determines the next file to send with DOS filemask processing, using the mask assigned to the `FileMask` property.

For non-batch protocols like Xmodem the file mask should not contain wildcards. Such protocols are capable of transmitting only a single file at a time, and if the mask matches more than one file only the first matching file is transmitted.

When the filemask technique is not adequate, you can write an event handler that implements whatever behavior you need.

Here is an OnProtocolNextFile event handler that provides custom NextFile processing.

```
const
  FileIndex : Word = 0;
  File1 = 'C:\AUTOEXEC.BAT';
  File2 = 'C:\CONFIG.SYS';
  File3 = 'C:\ASYNCPRO\ADPORT.PAS';
...
procedure TForm1.ApProtocolNextFile(
  CP : TObject; var FName : string);
begin
  Inc(FileIndex);
  case FileIndex of
    1 : FName := File1;
    2 : FName := File2;
    3 : FName := File3;
    else FName := '';
  end;
end;
```

This example sends only the three files named by the constants File1, File2, and File3. The event handler returns each string in sequence. After the three files are transmitted, the handler returns an empty string to tell the protocol that no more files are available to transmit.

Async Professional does not provide built-in support for transmitting an arbitrary list of files. That's because VCL provides the TStringList class, which can easily be combined with an OnProtocolNextFile event handler to send any list of files. The EXFLIST example program shows how.

AcceptFile processing

When receiving files, there may be times when you don't want the incoming file. Consider, for example, an open BBS where a first time caller is attempting to upload a 10M byte file at 2400 baud. Since this would tie up the BBS for more than 11 hours you probably would want to refuse it immediately. If the caller is using a protocol that transmits the file size in advance, you can detect that it's bigger than you want and refuse the upload.

As another example, suppose that a BBS has a well-publicized rule that it accepts only LZH uploads and it detects that an incoming file has a ZIP extension. If a caller is using a protocol that transmits the file name in advance, you can refuse an upload immediately.

The OnProtocolAccept event can be used to build such behavior into your application.

Note that Zmodem, alone among the Async Professional protocols, has built-in functionality for certain kinds of accept file functions. For example, it can reject an upload that would overwrite an existing file, or accept it only if the upload's time stamp is newer than the existing file. These options are described fully in “Zmodem” on page 507. The `AcceptFile` function is more general than this, and applies to all of the Async Professional protocols.

Once the protocol knows the name of an incoming file, but before it starts receiving data, it generates an `OnProtocolAccept` event. A form can respond to the event by setting the `Accept` parameter to `True` to accept the file, or `False` to refuse it. By default, all files are accepted.

For all protocols except Zmodem, the first rejected file terminates the entire batch transfer. Zmodem has provisions for skipping files, and the transmitter picks up again with the next file after the rejected one.

The `OnProtocolAccept` event also provides an opportunity to rename an incoming file if its current name isn't acceptable. For example, if a file name conflicts with an existing file, you can accept the file but change its name.

Note that all protocols have built-in options for handling incoming file name collisions. See the `WriteFailAction` property on page `WriteFailAction` property for a complete description. You don't need to write an `OnProtocolAccept` event handler if these constants provide the needed behavior.

Here is an event handler that rejects all files with the ARC extension:

```
procedure TForm1.ApdpProtocol1ProtocolAccept(  
  CP : TObject; var Accept : Boolean; var FName : string);  
begin  
  Accept := AnsiCompareText(ExtractFileExt(FName), '.ARC') <> 0;  
end;
```

This example examines the extension of the incoming file and sets `Accept` to `False` if it matches “.ARC”. Note that the function can check various properties to obtain additional information about the file. In particular, the file size is available by reading the `FileLength` property for protocols that transmit the size.

Internal logic

The protocol component has so far been described as a black box—you initialize it and call `StartTransmit` or `StartReceive` to perform the protocol “magic.” Now it is time to look inside the box, at how the protocol engine works. With this additional information, you will be able to use the protocols more effectively and take better advantage of the events generated by the protocol window.

Receiving files

When receiving files, the protocol window employs the following logic shown in Figure 14.1. This diagram generally applies to all protocols.

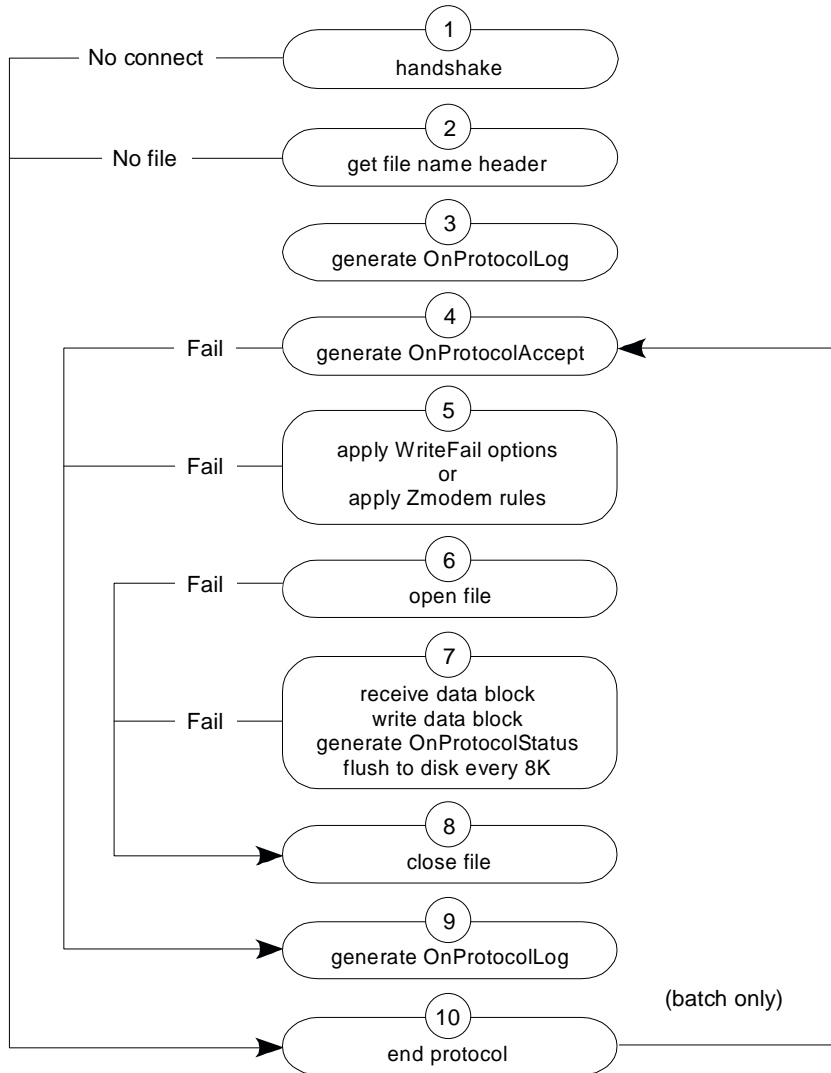


Figure 14.1: Protocol window logic when receiving files.

The protocol first attempts to “handshake” with the remote machine. A handshake consists of a valid response to an initial character sequence sent by the transmitter. If the handshake is unsuccessful after a specified number of retries, the protocol ends.

If the handshake is successful, the transmitter is asked for the name of the next file to transmit. It responds by sending a block containing the name.

The protocol generates the `OnProtocolLog` event to give the application an opportunity to record the file name or take any other special action needed at the start of the transfer. Note that the logging routine receives the file name before the `OnProtocolAccept` event handler has had a chance to modify it.

Next the protocol generates the `OnProtocolAccept` event. If the message handler sets `Accept` to `False`, the file is skipped and control is transferred to step 9. Otherwise, the built-in `WriteFail` options or `Zmodem`’s built-in file management rules are applied. If the protocol fails at this point, control is transferred to step 9.

The received file is created using the name from the file name header, perhaps as modified by the `OnProtocolAccept` event handler. Step 6 also allocates work buffers and initializes several internal variables used to manage an 8K byte receive buffer. If the open fails, control is transferred to step 8, which disposes of buffers and closes the file.

The actual transfer of data comes next in step 7. The internal operations of this step vary tremendously among the protocols, so it is condensed in this diagram. The `OnProtocolStatus` event is generated at least once for each block received. If unrecoverable errors occur for any reason (user abort, broken connection, disk full, etc.), control is transferred to step 8.

After the file transfer is complete, the file is closed in step 8. Then the `OnProtocolLog` event is generated with information regarding whether the file was received OK, rejected, or failed.

In a batch protocol, control returns to the top of the loop to get another file header. If one is received, the whole process is repeated. Otherwise, the protocol is ended by coordinating with the transmitter and cleaning up.

Transmitting files

When transmitting files, the protocol window employs the following logic as shown in Figure 14.2. This diagram generally applies to all protocols.

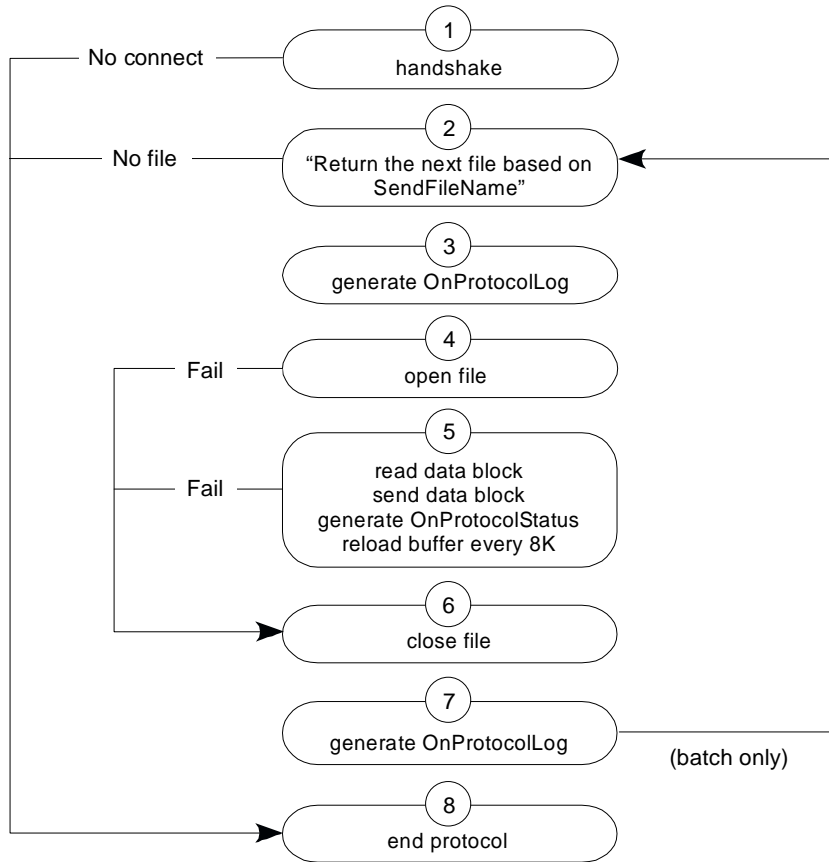


Figure 14.2: Protocol window logic when transmitting files.

The protocol first attempts to *handshake* with the remote machine. A handshake consists of sending an initial character sequence and waiting for a valid response. If the handshake is unsuccessful after a specified number of retries, the protocol ends.

If the handshake is successful, the protocol generates an `OnProtocolNextFile` event. The default handler of the component returns the next file based on a file mask, or a custom event handler can return the next file using its own logic.

The protocol generates the `OnProtocolLog` event to give the application an opportunity to record the outgoing file name or take any other special action needed at the start of the transfer.

The outgoing file is opened in step 4. This step also allocates work buffers and initializes variables used to manage an 8K byte send buffer. If any of these steps fail, control is transferred to step 6 to clean up.

The actual transfer of data comes next in step 5. The file is read in 8K byte blocks and sent using the block size native to the protocol. The `OnProtocolStatus` event is generated at least once for every block sent. If unrecoverable errors occur, control is immediately transferred to step 6.

After the file transfer is complete, the file is closed and buffers are disposed in step 6. Then the `OnProtocolLog` event is generated with information regarding whether the file was transferred OK, rejected, or failed.

In a batch protocol, control returns to the top of the loop to get another file to send. If one is available, the whole process is repeated. Otherwise, the protocol is ended by coordinating with the receiver and cleaning up.

Xmodem

Xmodem is the oldest protocol supported by Async Professional. It was developed and first implemented by Ward Christensen in 1977 and placed in the public domain. Since then, it has become an extremely popular protocol and continues in use today (although at a diminished frequency).

Xmodem is also the simplest, and perhaps the slowest, protocol supported by Async Professional. Xmodem uses blocks of only 128 bytes and requires an acknowledgment of each block. It uses only a simple checksum for data integrity.

What follows is a simplified description of the Xmodem protocol, although it describes far more than is required to actually use the protocol in Async Professional. For additional details on the internals of the Xmodem protocol, see the YMODEM.DOC file in the PROTD0C.LZH archive. Figure 14.3 shows the format for XModem blocks.

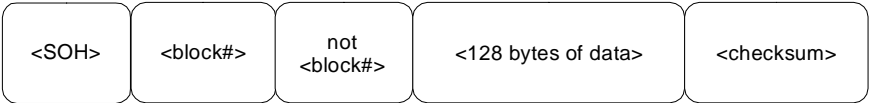


Figure 14.3: The format for XModem blocks.

The <SOH> character marks the start of the block. Next comes a one byte block number followed by a ones complement of the block number. The block number starts at one and goes up to 255 where it rolls over to zero and starts the cycle again. Following the block numbers are 128 bytes of data and a one-byte checksum. The checksum is calculated by adding together all the data bytes and ignoring any carries that result. Table 14.3 describes a typical Xmodem protocol transfer.

Table 14.3: Description of a typical XModem protocol transfer

Transmitter		Receiver
	<---	<NAK>
<SOH><1><254><128 data bytes><chk>	--->	
	<---	<ACK>
<SOH><2><253><128 data bytes><chk>	--->	
	<---	<ACK>
<EOT>	--->	
	<---	<ACK>

The receiver always starts the protocol by issuing a <NAK>, also called the handshake character. It waits 10 seconds for the transmitter to send a block of data. If it doesn't get a block within 10 seconds, it sends another <NAK>. This continues for 10 retries, after which the receiver gives up.

If the receiver does get a block, it compares the checksum it calculates to the received checksum. If the checksums differ, the receiver sends a <NAK> and the transmitter resends the block. If the checksums match, the receiver accepts the block by sending an <ACK>. This continues until the complete file is transmitted. The transmitter signifies this by sending an <EOT>, which the receiver acknowledges with an <ACK>.

Either side can cancel the protocol at any time by sending three <CAN> characters (^X). However, during an Xmodem receive the receiver cannot tell whether the <CAN> characters are real data or a cancel request. The sequence is recognized as a cancel request only when it comes between blocks. Hence, more than three <CAN> characters are sometimes required to cancel the receiver. Sufficient characters are required to complete the current block, then three more <CAN> characters to cancel the protocol.

From this description several things become clear. First, this protocol does not transfer any information about the file being transmitted. Hence, the receiver must assign a name to the incoming file.

The receiver also doesn't know the exact size of the file, even after it is completely received. The received file size is always a multiple of the block size. This Xmodem implementation fills the last partial block of a transfer with characters of value BlockFillChar, whose default is ^Z. BlockFillChar is a typed constant defined in AwTPcl.pas. To change the default BlockFillChar, simply assign the new character to that typed constant.

Xmodem often exhibits a start-up delay. The transmitter always waits for a <NAK> from the receiver as its start signal. If the receiving program was started first, the transmitter probably missed the first <NAK> and must wait for the receiver to time out and send another <NAK>.

Xmodem offers no *escaping* of binary control characters. Escaping means that characters can be transformed before being transmitted to prevent certain binary data characters, such as <XON>, from being interpreted as data link control characters. As a result, you can't use software flow control in an Xmodem transfer (since the flow control software would misinterpret <XON> or <XOFF> characters in the data stream as flow control requests) and Xmodem itself can't tell the difference between <CAN> characters used for protocol cancellation and for data.

The only merit of the basic Xmodem protocol is that it is so widespread that it's probably supported by any microcomputer communications program you can find, thus providing a lowest common denominator between systems.

Xmodem extensions

Xmodem has been tweaked and improved through the years. Some of these variations have become standards of their own and are supported by Async Professional. These Xmodem extensions are treated as separate protocols enabled by assigning a different value to the `ProtocolType` property.

The first of these improvements is called Xmodem CRC, which substitutes a 16 bit CRC (cyclic redundancy check) for the original checksum. This offers a much higher level of data integrity. When given the opportunity, you should always choose Xmodem CRC over plain Xmodem.

The receiver indicates that it wants to use Xmodem CRC by sending the character 'C' instead of <NAK> to start the protocol. If the transmitter doesn't respond to the 'C' within three attempts, the receiver assumes the transmitter isn't capable of using Xmodem CRC. The receiver automatically drops back to using checksums by sending a <NAK>.

Another popular extension is called Xmodem 1K. This derivative builds on Xmodem CRC by using 1024 byte blocks instead of 128 byte blocks. When Xmodem 1K is active, each block starts with an <STX> character instead of an <SOH>. Xmodem 1K also uses a 16 bit CRC as the block check.

A larger block size can greatly speed up the protocol because it reduces the number of times the transmitter must wait for an acknowledgment. However, it can actually reduce throughput over noisy lines because more data must be retransmitted when errors are encountered. The implementation of Xmodem 1K in Async Professional drops back to 128 byte blocks whenever it receives more than 5 <NAK> characters in a row. Once it drops back to 128 byte blocks, it never tries to step back up to 1024 byte blocks.

The final Xmodem extension supported by Async Professional is Xmodem 1KG. This *streaming* protocol is requested when the receiver sends 'G' as the initial handshake character instead of <NAK>. Streaming in this context means that the transmitter continuously transmits blocks without waiting for acknowledgements. In fact, all blocks are assumed to be correct and the receiver never sends acknowledgements. If the receiver does encounter a bad block, it aborts the entire transfer by sending a <NAK>.

You shouldn't even consider using this streaming protocol unless you are using error correcting modems with their error control features turned on. In fact, you might want to have your application refuse to use Xmodem 1KG if error correcting modems aren't detected. The `OnGotErrCorrection` event of the `TApdModem` component (see page 461) can be used to detect an error correcting connection. The advantage of a streaming protocol like Xmodem 1KG is its very high throughput. There is no acknowledgement delay, so the protocol is extremely efficient. Zmodem has this same property, but can also retry and recover from errors.

Ymodem

Ymodem is a derivative of Xmodem that is different enough to be called a unique protocol. The details and history of Ymodem are fully explained in the file YMODEM.DOC in the PROTD0C.LZH archive. What follows is a simplified explanation of Ymodem that provides more than enough information to use it with Async Professional.

Ymodem is essentially Xmodem 1K with batch facilities added, which means that a single protocol session can transfer as many files as you care to transmit. Another important enhancement is that the transmitter can provide the receiver with the incoming file name, size, and timestamp.

Ymodem achieves this by adding block zero to the Xmodem 1K protocol. Block zero is transferred first and contains file information in the format shown in Figure 14.4.

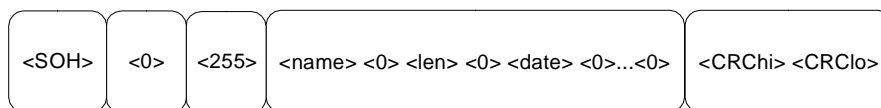


Figure 14.4: File information and format for a YModem block zero transfer.

The <name> field is the only required field. It supplies the name of the file in lower case letters. Path information can be included but the protocol requires that all '\ ' characters be converted to '/ '.

The <len> field specifies the file length as an ASCII string. This field allows the receiver to truncate the received file to discard the filler characters at the end of the last block.

The <date> field is the date and time stamp of the file. It is transmitted as the number of seconds since January 1, 1970 GMT, expressed in ASCII octal digits (a Unix convention).

Async Professional takes care of properly formatting this block. You don't need to do anything but specify the name of the file to transmit. When receiving Ymodem files, Async Professional uses the information in this block, if present, to adjust the size of the received file and its modification date.

Table 14.4 describes a typical YModem protocol transfer.

Table 14.4: *Description of a typical YModem protocol transfer*

Transmitter		Receiver
	<---	'C'
<SOH><0><255><file info><crc>	--->	
	<---	<ACK>
	<---	'C'
<STX><1><254><1024 data bytes><crc>	--->	
	<---	<ACK>
<STX><2><253><1024 data bytes><crc>	--->	
	<---	<ACK>
<EOT>	--->	
	<---	'C'
<SOH><0><255><file info><crc>	--->	
	<---	<ACK>
	<---	'C'
<STX><1><254><1024 data bytes><crc>	--->	
	<---	<ACK>
<EOT>	--->	
<SOH><0><255><128 zeros><crc>	--->	
	<---	<ACK>

As with the Xmodem protocols, the Ymodem protocol starts when the receiver sends a handshake character ('C') to the transmitter. The transmitter responds with a properly formatted block zero. The receiver acknowledges this with an <ACK> and then starts a normal Xmodem CRC protocol by issuing another 'C' handshake character.

Ymodem extensions

The Ymodem specification permits Ymodem to use a combination of 128 and 1024 byte blocks. Most Ymodem protocols start with 1024 byte blocks and drop back to 128 byte blocks only if repeated errors are detected. Once the block size is reduced to 128 bytes, it is never stepped back up to 1024.

Like Xmodem, Ymodem also offers a streaming extension called Ymodem G. This is similar in performance (and drawbacks) to Xmodem 1KG, but like Ymodem itself offers the advantages of batch transfers and file information.

Zmodem

Of all the protocols supported by Async Professional, Zmodem offers the best overall mix of speed, features, and tolerance for errors. The Zmodem protocol has many options and clearly was meant to have lots of room for growth. The Async Professional implementation of Zmodem does not cover the entire protocol specification but it does implement the features most likely to be required by your application. It should generally be your protocol of choice.

Zmodem was developed for the public domain by Chuck Forsberg under contract to Telenet. The original purpose was to provide a durable protocol with strong error recovery features and good performance over a variety of network types (switched, satellite, etc.). It has generally achieved these design goals.

What follows is a simplified explanation of Zmodem that provides more than enough information to use it with Async Professional. Refer to the ZMODEM.DOC file of PROTDOL.ZH for further details.

Zmodem borrows some concepts from Xmodem, Ymodem, and Kermit but is really a completely new protocol. Instead of adopting the simple block structure of Xmodem and Ymodem, Zmodem employs headers, data subpackets, and frames. A header contains a header identifier, a type byte, four information bytes, and some block check bytes. A data subpacket contains up to 1024 data bytes, a data subpacket type identifier, and some block check bytes. A frame consists of one header and zero or more data subpackets.

Due to the complexity and variety of the Zmodem header and data subpacket formats, they are not all detailed here. Instead, Table 14.5 provides a high level look at a sample Zmodem file transfer.

Table 14.5: *Description of a typical ZModem protocol transfer*

Sender	Receiver	Explanation
'rz'<cr>	--->	Start marker for automated transfers.
ZrQinit	--->	Request for receiver's information.
	<--- ZrInit	Receiver answers with its options.
ZFile	--->	Transmitter sends file information.
	<--- ZrPos	Receiver sets the starting filepos.
ZData	--->	Transmitter says file data to follow.
data subpacket	--->	Transmitter sends a data subpacket.
...		Continues until all data is sent.

Table 14.5: *Description of a typical ZModem protocol transfer (continued)*

Sender	Receiver	Explanation
ZEOF	--->	Transmitter indicates end-of-file.
	<--- ZRINIT	Receiver says ready for next file.
ZFIN	--->	Transmitter indicates no more files.
	<--- ZFIN	Receiver acknowledges.
'OO'	--->	Transmitter signs off.

The ZXxx tags are the header types that the two computers trade back and forth as they decide what is to be done. You don't need to understand them unless you find yourself trying to decipher a Zmodem transfer using the Async Professional tracing facility. To do so, you'll likely need to consult the ZMODEM.DOC file and/or the Async Professional source code.

In most cases all data in the file is sent in one ZData frame (the ZData header followed by as many data subpackets as required). The receiver doesn't have to acknowledge any of the blocks unless the transmitter specifically asks for it. The Zmodem protocol as implemented by Async Professional never asks for an acknowledgement; however, it respects such requests from the transmitter.

Typically, once a file transfer is underway, the receiver interrupts the transmitter only if it receives a bad block as determined by comparing block check values. An error is reported by sending a ZrPos header, telling the transmitter where in the file to start retransmitting.

The protocol can be canceled at any time if either side sends five <CAN> characters (^X).

Control character escaping

Zmodem *escapes* certain control characters. Escaping means that characters are transformed before being transmitted to prevent certain binary data characters, such as <XON> and <CAN>, from being interpreted as data link control characters.

Escaping isn't something you need to enable or disable because it's always on. It is mentioned here because escaping is what permits you to use software flow control with Zmodem. That isn't possible with the Xmodem/Ymodem family.

Zmodem always escapes the following characters:

<DLE>	Data link escape character (10h, ^P)
<XON>	XOn character (11h, ^Q)
<XOFF>	XOff character (13h, ^S)
<CAN>	Zmodem escape character (18h, ^X)
<DLE*>	Data link escape character with high bit set (90h)
<XON*>	XOn character with high bit set (91h)
<XOFF*>	XOff character with high bit set (93h)

Zmodem escapes all control characters when requested to by the remote protocol.

Protocol options

While the Zmodem specification describes all sorts of features, not all Zmodem implementations are expected to support all of the features. One of the first things that happens in a Zmodem protocol is that the receiver tells the transmitter what features it supports. The transmitter might modify its standard behavior to accommodate the receiver's support (or lack of support) for a particular option.

Since this process is handled automatically, you generally don't need to worry about it. For your information, the protocol options that Async Professional Zmodem provides and doesn't provide are listed here.

Async Professional supports the following Zmodem protocol options:

- True full duplex for data and control channels.
- Receiving data during disk I/O.
- Sending a break signal.
- Using 32 bit CRCs.
- Escaping all control characters.

Async Professional does not support the following protocol options:

- Encryption.
- LZ data compression.
- Escaping the 8th bit.
- End-of-line conversion for Unix newline characters.
- Sparse files.

Transfer resume

The Zmodem specification describes an option called *recover/resume*. This option is requested by the transmitter when it wants to resume a previously interrupted file transfer. When the receiver sees the request for this option, it compares the incoming file name with the files in its destination directory. If the incoming file already exists and is smaller than the one being transmitted, the receiver assumes that the transmitter wants to transfer only the remaining portion of the file.

When this condition exists, the receiver opens the existing file and moves the file pointer to the end of the file. It then tells the transmitter to move its file pointer to the same point in its copy of the file. The transmitter starts sending data from that point, which resumes the transfer from where it was interrupted.

This option can also be used to append new data to a remote copy of a file.

In either case, you use this option as follows:

```
ApdProtocol.FileMask := 'BIGFILE';
ApdProtocol.ZmodemRecover := True;
ApdProtocol.StartTransmit;
```

File management options

Zmodem has a variety of file management options built into it. These are simple rules that tell Zmodem whether or not to accept a file. Table 14.6 shows the possible options.

Table 14.6: *ZModem file management options*

Option Code	Explanation
zfoWriteNewerLonger	Transfer if new, newer, or longer.
zfoWriteCrc	Not supported, interpreted same as zfWriteNewer.
zfoWriteAppend	Transfer if new, append if existing.
zfoWriteClobber	Transfer always.
zfoWriteNewer	Transfer if new or newer.
zfoWriteDifferent	Transfer if new or different dates or sizes.
zfoWriteProtect	Transfer only if new.

The `zfoWriteCrc` option, which requests that a file be transferred only if its CRC is different from the remote copy's, is not supported. When this option is requested, it is treated the same as the `zfoWriteNewer` option.

The file management options are always requested by the transmitter. To use them, assign a value to the `ZmodemFileOption` property before calling `StartTransmit`. The default behavior is `zfoWriteNewer`. For example, to transmit all files regardless of whether such files already exist on the remote machine, make the following assignment:

```
ApdProtocol.ZmodemFileOption := zfoWriteClobber;
```

Even though the transmitter sets the file management options, Async Professional allows the receiver to change them. For example, suppose the transmitter has requested `zfoWriteClobber` but you want to accept only newer files. In this case you would set the `ZmodemOptionOverride` property to `True` before calling `StartReceive`:

```
ApdProtocol.ZmodemOptionOverride := True;  
ApdProtocol.ZmodemFileOption := zfoWriteNewer;
```

Setting this property to `True` tells Zmodem to ignore the file management options requested by the transmitter and to use `zfoWriteNewer` instead.

Another file management property called `ZmodemSkipNoFile` is available. Set this property to `True` to force the receiver to skip any incoming file that doesn't already exist in the destination directory.

Whatever file management rules are in effect, the receiver applies them and either accepts each file or rejects it. If the file is accepted, the file transfer proceeds normally. If the file is rejected, the receiver sends a `ZSkip` frame to the transmitter, which stops sending the current file and moves on to the next one in its list.

Don't forget that you can implement your own file management rules with an `OnProtocolAccept` event handler.

Automatic block size control

The Zmodem protocol decreases or increases the number of bytes transmitted per block in response to retransmission requests, usually due to poor line conditions or random line errors. The rationale is that small blocks can transmit more frequently without errors, since there's less time for a small block to be hit by line noise. And, even if a small block is corrupted by noise, it is faster to retransmit than a large block.

The protocol employs the following logic to control the block size. If the transmitter receives an unsolicited request from the receiver to resend data, it reduces the block size from 1024 to 512. If the transmitter receives another request to resend, it reduces the block size from 512 to 256. It never reduces the block size below 256 bytes. Conversely, the transmitter raises the block size immediately back to 1024 bytes when it sends four blocks in a row without receiving any requests to resend.

Similar logic is employed with 8K Zmodem, which uses 8192 byte blocks by default. The block size is halved for each retransmission request received, down to a minimum of 256 bytes. The block size is increased to 8K bytes in a single step after four blocks are transmitted without any requests to resend.

Block size control is automatic and cannot be disabled. While this behavior is not documented in the public domain Zmodem specification, it is the process followed by the popular DSZ program and is acceptable to any common Zmodem implementation.

Large block support

Async Professional Zmodem also includes support for 8K byte blocks. This behavior is outside the public domain specification and was added largely for programmers who need to transfer files to or from several popular BBS and FIDONet mailer programs. Since large blocks are not supported by all common Zmodem implementations, their use is not automatic—you must specifically enable them before starting a file transfer by setting the Zmodem8K property to True.

The output buffer size of the port object used by the protocol window must also be large enough to support the 8K option. The output buffer must be capable of holding an entire 8K block with escaping, which in the worst case can double the size of the block. Hence, the output buffer must be at least $2 \times 8192 + 30$, or 16414, bytes. (See “Buffer sizes” on page 485 for more information.)

Kermit

The Kermit protocol was developed to allow file transfers in environments that other protocols can't handle. Such environments include links that only pass 7 data bits, links that can't handle control characters, computer systems that can't handle large blocks, and diverse other links such as those between a PC and a mainframe.

Kermit is a public domain protocol that was developed at Columbia University. (The name refers to Kermit the Frog, from The Muppet Show.) What follows is a simplified explanation of Kermit that provides more than enough information to use it with Async Professional. For additional details, get the Kermit protocol specification from Columbia University, Kermit Distribution, Department OP, 612 West 115th Street, New York, NY 10025.

Character quoting

Character quoting means pretty much the same thing that escaping means in Zmodem. The character is replaced by a quote character and a modified form of the original character. The quote character tells the receiver how to convert the modified character back to its original value. Quoting ensures that certain binary characters are never put into the data stream where they could be misinterpreted by a modem or another part of the serial link.

Although Zmodem transforms only a few critical characters such as <XON> and <XOFF>, Kermit quotes nearly all characters. This is one of the features that permits Kermit to run in nearly any environment. When quoting is finished, a Kermit data packet consists almost entirely of printable ASCII characters. The only exceptions are an <SOH> character at the start of each packet and a <CR> at the end.

Kermit quotes control characters by replacing them with a quote character and a modified version of the control character. For example, ^A becomes '#A' where '#' is the quote character. The process of converting ^A to 'A' is called "Ctl" and it works like this:

```
Ctl(x) = x xor 40h;
```

This operation is its own inverse, that is, $Ctl(Ctl(x)) = x$.

Kermit also quotes characters with their eighth bit set, which allows it to transmit 8 bit data over 7 bit data links. The quote character in this case is '&' and the quoted data character is obtained simply by stripping the high bit. For example, the quoted version of character \$C1 ('A' with its high bit set) is '&A'.

Binary numbers in Kermit packet headers and in repeated character strings are also transformed to assure that they are printable characters. This is achieved by adding 32 to each number before it is transmitted and subtracting 32 after it is received. In Kermit parlance, these operations are known as "ToChar" and "UnChar."

Kermit has a simple built-in data compression mechanism called run length encoding. When it sees a long string of repeated characters, it compresses the string into a quote character, a length byte, and the repeated character. Obviously, there must be at least 4 repeated characters before there is any compression. The quote character for run length encoding is '~'. Hence, the string "AAAAA" becomes "~%A", where '%' is equivalent to a binary 5 after the "ToChar" operation.

Kermit packets

Figure 14.5 shows the general format of a Kermit packet.

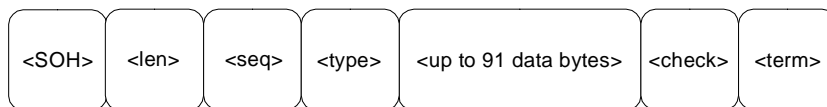


Figure 14.5: The format for a typical Kermit packet.

The <SOH> character, also called the mark field, indicates the start of a Kermit packet.

The length byte specifies the number of bytes that follow. Since it must be transmitted as a printable 7 bit character the binary maximum value is 94, which means that the maximum length of a normal Kermit packet is 96 bytes including the <SOH> and the <length> field.

The <seq> byte is a packet sequence number in the range of 0 to 63. After 63 it cycles back to 0.

The <type> byte describes the various Kermit packet types, which are analogous to the Zmodem frame types.

The data field contains up to 91 bytes including all quote characters. The number of actual data bytes could be considerably less, particularly if binary data is being transmitted.

The standard Kermit <check> field is a single-byte checksum. Kermit offers two optional block check methods called two-byte checksum and three-byte CRC. See the BlockCheckMethod property (page 530) for more information.

The <term> character is the packet terminator which equals carriage return (ASCII 13) by default. You will probably never need to change the terminator.

Table 14.7 describes a typical Kermit protocol transfer.

Table 14.7: *A description of a typical Kermit protocol transfer*

Transmitter		Receiver	Explanation
KSendInit	--->		Transmitter sends its options.
	<---	KAck	Receiver answers with its options.
KFile	--->		Transmitter sends filename.
	<---	KAck	Receiver acknowledges filename.
KData	--->		Transmitter sends data packet.
	<---	KAck	Receiver acknowledges data packet.
KData	--->		Transmitter sends data packet.
	<---	KAck	Receiver acknowledges data packet.
...			Continues until all data sent.
KEndoffile	--->		Transmitter says end of file.
	<---	KAck	Receiver acknowledges and closes file.
KBreak	--->		Transmitter says end of protocol.
	<---	KAck	Receiver acknowledges end of protocol.

The KXxx tags are the packet types that the two computers exchange as they decide what is to be done. You don't need to understand them unless you find yourself trying to decipher a Kermit transfer using the Async Professional tracing facility. To do so, you'll likely need to consult the Kermit specification file and/or the Async Professional source code.

Kermit options

Like Zmodem, Kermit offers a variety of options. An implementation of Kermit is not required to support all options. Hence, one of the first things that happens in a Kermit protocol is that the two sides exchange their desired options and use the lowest common denominator of the two sets.

Table 14.8 shows the Kermit options that Async Professional supports and the default values each uses. The entries in the first column are TApdProtocol property names that can be used to adjust each option.

Table 14.8: *Kermit property options and default values*

Property	Default	Explanation
KermitMaxLen	80 bytes	Maximum length of the data field.
KermitTimeoutSecs	5 seconds	Maximum timeout between characters.
KermitPadCount	0 bytes	No pad characters before packets.
KermitPadCharacter	' '	Space character used for padding.
KermitTerminator	<CR>	Packet terminator is a carriage return.
KermitCtlPrefix	'#'	Control character prefix is '#'.
KermitHighbitPrefix	'Y'	Honor 8-bit quoting but don't require it.
BlockCheckMethod	'1'	Use a 1 byte checksum.
KermitRepeatPrefix	'~'	Repeat prefix is '~'.
KermitMaxWindows	0	No sliding windows.

KermitMaxLen is the maximum number of bytes you want Kermit to include in one packet. The normal maximum value is 94; the default value is 80 as suggested by the Kermit Protocol Manual. If KermitMaxLen exceeds 94, the Kermit “long packets” feature is enabled. The absolute maximum value is 1024.

KermitTimeoutSecs is the amount of time, in seconds, that a Kermit transmitter will wait for an acknowledgement or a Kermit receiver will wait for the next byte to be received. If more than KermitTimeoutSecs seconds elapse without receiving anything, Kermit assumes an error occurred and resends.

KermitPadCount and KermitPadCharacter describe padding that can be added at the front of all Kermit packets. The only reason for padding is if the remote machine needs a delay between sending a packet and receiving a response. In this case, you can specify enough padding characters to generate the required delay. Generally, though, padding is unnecessary. The Kermit protocol as implemented by Async Professional automatically honors a remote’s request for padding.

KermitTerminator is the character that follows the check field in a packet. Although all Kermit packets have a terminator, it is used only by systems that need an end-of-line character before they can start processing input.

`KermitCtlPrefix` is the control character prefix that Kermit uses when performing “Ctl” quoting to transform control characters into printable ASCII characters. Generally you won’t need to change this prefix.

`KermitHighbitPrefix` specifies how Kermit transforms high-bit characters into characters without the high-bit set. Generally you won’t need to change this setting. See the property description for more information.

`BlockCheckMethod` specifies the type of block checking Kermit should perform. ‘1’ corresponds to the `bcmChecksum` value of the `TBlockCheckMethod` type, and it means that Kermit should use a single-byte checksum. All Kermit implementations are guaranteed to support this form of block checking. `bcmChecksum2` means that Kermit should use a two-byte checksum, which offers only slightly more protection than the single-byte checksum. `bcmCrcK` means that Kermit should use a three-byte CRC. This is the preferred block check method because it offers the highest level of error detection. Unfortunately, not all Kermit implementations support the non-default block check methods. If the remote computer doesn’t support the block check method you request, both sides drop back to the single-byte checksum.

`KermitRepeatPrefix` is the repeated-character prefix that Kermit uses when compressing long strings of repeated characters. Generally you won’t need to change this prefix.

`KermitMaxWindows` is the number of sliding windows requested. Setting this to a value between 1 and 27 (the maximum allowed) enables sliding windows support.

The two sides of a Kermit protocol automatically negotiate which options to use, so no intervention is required by your program. If you wish to change the default options, use these properties.

Async Professional does not provide Kermit server functions and does not support file attribute packets.

Long packets

Async Professional includes support for long packets, which is an extension to standard Kermit that permits data packets of up to 1024 bytes. Long packets can substantially improve protocol throughput on clean connections that have small turnaround delays. Long packet support is turned off by default and must be enabled by setting `KermitMaxLen` to a value between 95 and 1024. Most other Kermit implementations also disable this option by default.

Although the specification allows for packets up to 9024 bytes, Async Professional limits long packets to 1024 bytes. Packets longer than 1024 bytes do not appreciably increase throughput, but they dramatically increase retransmission time when a line error occurs.

The specification also recommends the use of the higher-order checksums with long packets, but does not require it. Async Professional defaults to 2 byte checksums when long packet support is enabled, but drops back to 1 byte checksums if requested to do so by the remote Kermit.

Sliding Windows Control

Async Professional includes supports for the Kermit extension known as Sliding Windows Control, also called “SuperKermit.” Sliding Windows Control (SWC) provides a “send-ahead” facility that dramatically improves throughput when turnaround delays tend to be large, as when using satellite links.

Send-ahead means that the transmitter sends many blocks without waiting for an acknowledgement for each block. The transmitter collects acknowledgements when they eventually arrive and marks the previously transmitted blocks as acknowledged. This reduces turnaround delay (the time it takes the receiver to send an acknowledgement) to zero.

SWC operates by keeping a circular table of transmitted packets. The maximum number of packets in this table is called the window size, which is a number between 0 (no sliding window support) and 31. If the transmitter and receiver specify different window sizes, the smaller of the two is used. Async Professional's Kermit actually limits the maximum number of windows to 27 to avoid encountering a bug in the popular program MSKERMIT.

Sliding window support is off by default. It is enabled by setting the KermitMaxWindows property to a non-zero value.

On the sender's side, each transmitted packet is added to the table. When an acknowledgement is eventually received for a packet, its entry in the table is freed. If the table fills, the transmitter does not send more packets until it receives acknowledgements for one or more existing packets.

On the receiver's side, each received packet is added to the table and remains there until the table is full. Then the oldest packet is written to disk. When errors are detected, the receiver sends a <NAK> for each missed packet, starting past the last known good packet and continuing up to the most recently received packet.

It is possible to enable long packets and SWC simultaneously, but memory consumption rises dramatically from the single 80 byte buffer normally used by Kermit. In the worst case you could have 27 windows of 1024 bytes each, adding up to 27648 bytes.

ASCII

The term *ASCII protocol* is a bit of a misnomer, because in an ASCII transfer neither side of the link is following well-documented rules. An ASCII protocol is really just a convenient way of transmitting a text file.

A typical use for the ASCII protocol is when you need to transfer a text file to a minicomputer that doesn't have any protocols available. One way of accomplishing this is to run an Async Professional program that supports a terminal window and the ASCII protocol, such as the TCom demo program. You connect to the minicomputer, navigate to the minicomputer's editor, and open up a new text file. Then you start an ASCII protocol transmit of the file you need to transfer. The minicomputer's editor sees this as keystroke input to the editor. You finish the transfer by saving the editor's file.

The ASCII protocol provides options for tailoring such transfers to the remote machine's speed, which might necessitate delays between transmitted characters and lines. For example, when transmitting a file into a remote computer's editor, you might need to use delays to avoid overflowing the editor's keystroke buffer.

It is difficult for the receiver to know when an ASCII transfer is over because there is no agreed-upon method for indicating termination. The ASCII protocol terminates on any of three conditions: when it receives a ^Z character, when it times out waiting for more data, or when the user aborts the protocol and the application calls CancelProtocol. When any of these conditions is detected, the file is saved and the protocol ends.

End-of-line translations

Computer systems sometimes use different character sequences to terminate each line of a text file. Most PC software stores both a carriage return <CR> and a line feed <LF> at the end of each line. Other systems store only <LF> or only <CR> at the end of each line.

The ASCII protocol provides a number of options for translating from one end-of-line sequence to another, both when transmitting and when receiving. When performing these translations, the <CR> and <LF> characters are treated separately, based on the values assigned to the AsciiCRTranslation and AsciiLFTranslation properties. Table 14.9 shows the enumerated values used to control the behavior.

Table 14.9: *ASCII protocol enumerated property values*

Value	Explanation
aetNone	The character is not to be modified (the default).
aetStrip	The character is to be stripped from the data stream.
aetAddCRBefore	A <CR> is to be inserted before each <LF>. This can be specified only for the AsciiLFTranslation property.
aetAddLFAfter	An <LF> is to be added after each <CR>. This can be specified only for the AsciiCRTranslation property.

FTP

Async Professional provides two File Transfer Protocol (FTP) components that make it easy to implement FTP client support in your application.

The TApdFtpClient component takes care of the FTP protocol details and presents a friendly interface for transferring files to and from an FTP server. In addition, the component also provides a set of standard functions for manipulating files and directories at the FTP server.

The TApdFtpLog component automates the process of logging an FTP client-server dialog for auditing FTP activities.

Overview of FTP

File Transfer Protocol (FTP) is used to transfer files from one location on the Internet to another. An FTP client establishes a connection (called the control connection) to an FTP server at a well-known port number (usually 21). The control connection is used for a command-response dialog between the client and server. The client issues an FTP command which consists of an ASCII string containing a mnemonic command followed by any parameters required for the command. The server then responds with a reply consisting of an ASCII string containing a three digit reply code followed by some text. During a file transfer, a separate data connection is established to transfer the file data. When the transfer is complete, the data connection is closed.

When an FTP client establishes a control connection with an FTP server, the server will respond with a reply code that indicates that the user login procedure can commence. The login procedure consists of sending commands containing the user's ID name, password, and (possibly) account information as parameters. An FTP server requires authenticated login information before giving a user access to files and directories, however many servers allow a user to login "anonymously" for restricted access. Users log in anonymously using "ANONYMOUS" for their user ID name and their e-mail address for their password.

FTP error codes

If an FTP server rejects a command from an FTP client for one reason or another, the server will reply with an error code. Table 14.10 is a list of common FTP errors.

Table 14.10: *Common FTP errors*

Reply Code	Explanation
421	Service not available, closing control connection.
425	Can't open data connection.
426	Connection closed, transfer aborted.
451	Requested action aborted, local error in processing.
452	Requested action not taken.
500	Syntax error, command unrecognized.
501	Syntax error in parameters or arguments.
502	Command not implemented.
503	Bad sequence of commands.
504	Command not implemented for that parameter.
505	No such file or directory.
506	Usage error.
522	Transfer error bytes written.
530	Error in user login.
550	Requested action not taken due to error.
551	Requested action aborted.
553	Requested action not taken due to system error.

TApdProtocol Component

TApdProtocol implements all of the Async Professional file transfer capabilities in one comprehensive component. General issues associated with using this component are discussed in the first part of this chapter.

Note that certain properties that are described in the following reference section are specific to a particular protocol type. If a particular property is not supported by the current value of the ProtocolType property (e.g., the AsciiCharDelay property is not relevant to the Zmodem protocol), assigning a value to that property stores the new value in a field of the component, but has no effect until the ProtocolType is changed to the corresponding protocol. Protocol-specific properties have names that begin with the name of the protocol itself (e.g., ZmodemOptionOverride, ZmodemSkipNoFile, ZmodemFileOption, ZmodemRecover, and Zmodem8K).

Example

This example shows how to construct and use a protocol component. It includes a terminal window so you can navigate around an on-line service while you test the program, and a TApdProtocolStatus component so you can see the progress of the transfer.

Create a new project, add the following components, and set the property values as indicated in Table 14.11.

Table 14.11: *Example components and property values*

Component	Property	Value
TApdComPort	ComNumber	<Set as needed for your PC>
TApdEmulator		
TApdTerminal		
TApdProtocol	FileMask	EXPROT*. *
TApdProtocolStatus		
TButton	Name	Upload
TButton	Name	Download

Double-click on the Upload button's OnClick event handler within the Object Inspector and modify the generated method to match this:

```
procedure TForm1.UploadClick(Sender : TObject);
begin
    ApdProtocol1.StartTransmit;
end;
```

This method starts a Zmodem background protocol transmit session for all of the files matching the mask "EXPROT*.*". (Zmodem is the default protocol type for TApdProtocol instances.)

Double-click on the Download button's OnClick event handler within the Object Inspector and modify the generated method to match this:

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

This method starts a Zmodem background session to receive whatever files the transmitter sends.

The form includes a TApdProtocolStatus component, which is automatically displayed by the protocol and periodically updated to show the progress of the file transfer.

This example is in the EXPROT project in the \ASYNCPRO\EXAMPLES directory.

Hierarchy

TComponent (VCL)

- TApdBaseComponent (OOMisc) 8
 - TApdCustomProtocol (AdProtcl)
 - TApdProtocol (AdProtcl)

Properties

AbortNoCarrier	FileName	ProtocolError
ActualBPS	FinishWait	ProtocolLog
AsciiCharDelay	HandshakeRetry	ProtocolStatus
AsciiCRTranslation	HandshakeWait	ProtocolType
AsciiEOFTimeout	HonorDirectory	RTSLowForWrite
AsciiEOLChar	IncludeDirectory	StatusDisplay
AsciiLFTranslation	InitialPosition	StatusInterval
AsciiLineDelay	InProgress	TotalErrors
AsciiSuppressCtrlZ	KermitCtlPrefix	TransmitTimeout
Batch	KermitHighbitPrefix	TurnDelay
BlockCheckMethod	KermitLongBlocks	UppcaseFileNames
BlockErrors	KermitMaxLen	❶ Version
BlockLength	KermitMaxWindows	WriteFailAction
BlockNumber	KermitPadCharacter	XYmodemBlockWait
BytesRemaining	KermitPadCount	Zmodem8K
BytesTransferred	KermitRepeatPrefix	ZmodemFileOption
ComPort	KermitSWCTurnDelay	ZmodemFinishRetry
DestinationDirectory	KermitTerminator	ZmodemOptionOverride
ElapsedTicks	KermitTimeoutSecs	ZmodemRecover
FileDate	KermitWindowsTotal	ZmodemSkipNoFile
FileLength	KermitWindowsUsed	
FileMask	Overhead	

Methods

CancelProtocol	StartReceive	StatusMsg
EstimateTransferSecs	StartTransmit	

Events

OnProtocolAccept	OnProtocolFinish	OnProtocolNextFile
OnProtocolError	OnProtocolLog	OnProtocolStatus

Reference Section

AbortNoCarrier

property

```
property AbortNoCarrier : Boolean
```

Default: False

- ↳ Determines whether the protocol is canceled automatically when the DCD modem signal drops.

Using the AbortNoCarrier property is better than checking DCD and calling CancelProtocol in your own code. When you do this, the protocol engine sends a cancel sequence to the remote computer. If hardware flow control is enabled and the modem has lowered the DSR or CTS signals as well as DCD, the protocol waits several seconds before deciding it can't send the cancel command, leading to an unnecessary delay for the application. The AbortNoCarrier property prevents the protocol engine from sending the cancel sequence, so the protocol stops immediately.

Note that when transferring through Winsock the DCD signal is not present (Winsock does not contain the concept of a DCD line). The TAPdWinsockPort artificially raises and lowers the DCD property according to the connection state. The DCD property is therefore valid for testing connection states, and the AbortNoCarrier functionality will work as expected.

See also: CancelProtocol

ActualBPS

run-time property

```
property ActualBPS : LongInt
```

- ↳ Determines the data transfer rate used by EstimateTransferSecs.

This property can be used to set a bit per second (bps) rate that differs from the associated comport component's baud rate. The bps rate is used only by EstimateTransferSecs to compute transfer times. The actual bps differs from the port baud rate only in cases like the following: Two machines are communicating via MNP or V.32 modems at 9600 bps; both modems are using built-in data compression facilities to increase the effective bps rate (perhaps to 11000 bps); the machines are communicating with their modems at a rate of 19200 baud to ensure that the modems don't waste time waiting for data to send; and the machines are using hardware flow control to pace the flow of data between the modems and the machines.

Without setting ActualBPS, the protocol would base transfer rate calculations on a bps rate of 19200, the port baud rate. You should set ActualBPS to 9600, the actual connection speed. Even this is somewhat inaccurate because it doesn't take into account the improvements due to data compression, which are difficult to predict.

See also: EstimateTransferSecs

AsciiCharDelay

property

property AsciiCharDelay : Word

Default: 0

↪ Determines the number of milliseconds to delay between characters during an ASCII file transfer.

The default delay of zero should be retained whenever possible to maximize performance. However, if ASCII data is being fed directly into an application such as a text editor, it might be necessary to insert delays to allow the application time to process the data.

The following example sets the inter-character delay to 2 milliseconds and the inter-line delay to 50 milliseconds:

```
ApdProtocol1.AsciiCharDelay := 2;  
ApdProtocol1.AsciiLineDelay := 50;
```

See also: AsciiLineDelay

AsciiCRTranslation

property

property AsciiCRTranslation : TAsciiEOLTranslation

```
TAsciiEOLTranslation = (  
    aetNone, aetStrip, aetAddCRBefore, aetAddLFAfter);
```

Default: aetNone

↪ Determines the end-of-line translation mode for carriage returns.

Acceptable values to assign to this property are as follows:

Value	Description
aetNone	Do not modify the character.
aetStrip	Strip the character from the data stream.
aetAddLFAfter	Add an <LF> after each <CR>.

aetAddCRBefore does not apply to AsciiCRTranslation, so it is treated as aetNone.

The following example causes all <LF> characters to be stripped while <CR> characters are transmitted:

```
ApdProtocol1.ProtocolType := ptAscii;  
ApdProtocol1.AsciiCRTranslation := aetNone;  
ApdProtocol1.AsciiLFTranslation := aetStrip;  
ApdProtocol1.StartTransmit;
```

See also: AsciiEOLChar, AsciiLFTranslation

AsciiEOFTIMEOUT

property

```
property AsciiEOFTIMEOUT : Word
```

Default: 364

↳ Determines the number of ticks before an ASCII transfer is automatically terminated.

Because most text files are terminated by a ^Z character (ASCII 26), the ASCII protocol closes the file and ends the protocol when it finds a ^Z. If the received file isn't terminated by a ^Z, the ASCII protocol determines the file was completely received after a specified number of ticks elapse without receiving any new data. The default of 20 seconds can be changed by assigning a new tick value to this property.

AsciiEOLChar

property

```
property AsciiEOLChar : Char
```

Default: ^M (ASCII 13)

↳ Determines the character that triggers an inter-line delay.

After an ASCII file transmit sends the character specified by this property, it pauses for the number of milliseconds specified by the AsciiLineDelay property.

Note that this character is not involved in on-the-fly translation of end-of-line characters read from or written to an ASCII file; that translation is controlled by the AsciiCRTranslation and AsciiLFTranslation properties.

The default end-of-line character is <CR> or ^M. If you are transmitting Unix files, which use <LF> or ^J for the end-of-line marker, you should set AsciiEOLChar to ^J.

See also: AsciiLineDelay

AsciiLFTranslation

property

```
property AsciiLFTranslation : TAsciiEOLTranslation
TAsciiEOLTranslation = (
    aetNone, aetStrip, aetAddCRBefore, aetAddLFAfter);
```

Default: aetNone

↪ Determines the end-of-line translation mode for line feeds.

Acceptable values to assign to this property are as follows:

Value	Description
aetNone	Do not modify the character.
aetStrip	Strip the character from the data stream.
aetAddCRBefore	Insert a <CR> before each <LF>.

aetAddLFAfter does not apply to AsciiLFTranslation, so it is treated as aetNone.

See also: AsciiCRTranslation, AsciiEOLChar

AsciiLineDelay

property

```
property AsciiLineDelay : Word
```

Default: 0

↪ Determines the number of milliseconds to delay between lines during an ASCII file transfer.

The default delay of zero should be retained whenever possible to maximize performance. However, if ASCII data is being fed directly into an application such as a text editor, it might be necessary to insert delays to allow the application time to process the data.

See also: AsciiCharDelay

AsciiSuppressCtrlZ

property

```
property AsciiSuppressCtrlZ : Boolean
```

Default: False

↪ Determines whether an ASCII protocol stops transmitting when it encounters the first ^Z in the file.

If this property is False, the ASCII protocol transmits all characters in the file, including ^Z characters. If it is True, it stops before transmitting the first ^Z that it encounters. Generally you should leave this property set to False because the receiver might use ^Z as an end-of-protocol indicator, as Async Professional does.

```
property Batch : Boolean
```

↳ Returns True if the current protocol supports batch transfers.

Batch transfers are those that allow sending more than one file in the same protocol session. Batch transfers in Async Professional include: Kermit, Ymodem, and Zmodem.

This property is most useful within an OnProtocolStatus event handler. See “Protocol status” on page 489 for more information.

BlockCheckMethod

property

```
property BlockCheckMethod : TBlockCheckMethod
```

```
TBlockCheckMethod = (  
    bcmNone, bcmChecksum, bcmChecksum2, bcmCrc16, bcmCrc32, bcmCrcK);
```

↳ Determines the error checking method used by the protocol.

The default error checking method depends on the protocol. See the section describing each protocol at the beginning of this chapter for additional information.

The following values can be assigned to the property:

Value	Description
bcmNone	No error checking.
bcmChecksum	Single byte checksum.
bcmChecksum2	Two byte checksum used by Kermit.
bcmCrc16	16-bit CRC.
bcmCrc32	32-bit CRC used by Zmodem.
bcmCrcK	Three byte CRC used by Kermit.

The Xmodem1K, Xmodem1KG, Ymodem, YmodemG, and ASCII protocols provide either no error checking or a single error checking mode, so they ignore assignments to BlockCheckMethod.

Assigning `bcmCrc16` to `BlockCheckMethod` converts an Xmodem protocol into an XmodemCrc protocol. Conversely, assigning `bcmChecksum` to `BlockCheckMethod` converts an XmodemCrc protocol to an Xmodem protocol.

The Zmodem protocol accepts only the `bcmCrc16` and `bcmCrc32` types. The Kermit protocol accepts only the `bcmChecksum`, `bcmChecksum2`, and `bcmCrcK` types.

No error is generated if an unaccepted type is assigned, but the assignment is ignored. You should be sure to set the desired `ProtocolType` before setting a non-default `BlockCheckMethod`.

See also: `ProtocolType`

BlockErrors

read-only, run-time property

```
property BlockErrors : Word
```

↳ The number of errors that have occurred while transferring the current block.

This property is most useful within an `OnProtocolStatus` event handler. See “Protocol status” on page 489 for more information.

See also: `TotalErrors`

BlockLength

read-only, run-time property

```
property BlockLength : Word
```

↳ The number of bytes currently being transferred per block.

For some protocols this value remains fixed (e.g., Xmodem always uses 128 byte blocks); for others it can vary during the transfer process (e.g., Zmodem can vary between 8192 bytes and 256 bytes depending on options and line conditions).

This property is most useful within an `OnProtocolStatus` event handler. See “Protocol status” on page 489 for more information.

BlockNumber

read-only, run-time property

```
property BlockNumber : Word
```

↳ The number of blocks transferred so far.

This is obtained by dividing the number of bytes transferred by the current block length, so it will change if the block length changes.

This property is most useful within an `OnProtocolStatus` event handler. See “Protocol status” on page 489 for more information.

BytesRemaining

read-only, run-time property

```
property BytesRemaining : LongInt
```

↳ The number of bytes still to be transferred in the current file.

This is computed as the `FileLength` minus the value of `BytesTransferred`. When the file size isn't known, `BytesRemaining` returns zero.

This property is most useful within an `OnProtocolStatus` event handler. See “Protocol status” on page 489 for more information.

See also: `BytesTransferred`, `FileLength`

BytesTransferred

read-only, run-time property

```
property BytesTransferred : LongInt
```

↳ The number of bytes transferred so far in the current file.

When transmitting, this number is sometimes only an estimate. The uncertainty comes from the fact that the protocol window doesn't know when a particular byte has actually been transferred. `BytesTransferred` is the number of bytes the protocol window has transferred to the output buffer of the communications driver, minus the number of bytes that the driver reports are currently in the buffer.

Unfortunately, this calculation is still imperfect because it's impossible to know how much of the output buffer holds actual file data and how much holds overhead characters needed by the protocol. Each protocol has a few simple rules it uses to estimate this proportion, which in practice yield good estimates.

This property is most useful within an `OnProtocolStatus` event handler. See “Protocol status” on page 489 for more information.

See also: `BytesRemaining`

CancelProtocol

method

```
procedure CancelProtocol;
```

↪ Cancels the protocol currently in progress.

CancelProtocol cancels the protocol regardless of its current state. If appropriate, a cancel string is sent to the remote computer. The protocol generates an OnProtocolFinish event with the error code `ecCancelRequested`, then cleans up and terminates.

The following example shows how to cancel a protocol from within a protocol status dialog box:

```
procedure TStandardDisplay.CancelClick(Sender: TObject);  
begin  
    ApdProtocol1.CancelProtocol;  
end;
```

See also: `InProgress`, `OnProtocolError`

ComPort

property

```
property ComPort : TApdCustomComPort
```

↪ Determines the serial port used by the protocol.

A properly initialized comport component must be assigned to this property before using the protocol.

The comport should almost always be set to use 8 data bits, 1 stop bit, and no parity. It should have input and output buffers that meet the guidelines described in “Buffer sizes” on page 485. Most transfer protocols require that some form of flow control be enabled in the comport component.

DestinationDirectory

property

```
property DestinationDirectory : string
```

↪ Determines the directory where received files are stored.

If the value specifies only a drive (e.g., “D:”), files are stored in the current directory on that drive. If the property is set to an empty string, as it is by default, received files are stored in the current directory.

See also: `FileName`


```
property ElapsedTicks : LongInt
```

↳ The time elapsed since the protocol started.

In order to provide accurate character per second transfer rates, the protocol engine doesn't start the timer until it receives the first block from the remote computer, or until it sends the first data block. ElapsedTicks is measured in ticks, which occur at roughly 18.2 per second.

This property is most useful within an OnProtocolStatus event handler. See "Protocol status" on page 489 for more information.

See also: EstimateTransferSecs

EstimateTransferSecs

method

```
function EstimateTransferSecs(const Size : LongInt) : LongInt;
```

↳ Returns the amount of time to transfer a file.

You can call EstimateTransferSecs in a status event handler to obtain the approximate number of seconds required to transfer Size bytes of data. Typically, a status routine calls it in two places. In the first place, which should generally be executed only one time when the status routine is first called, it passes the total size of the file to get the total transfer time. In the second place, which should be executed every time the status routine is called, it passes the number of bytes remaining to get the transfer time remaining.

EstimateTransferSecs automatically accounts for the baud rate of the port's connection and various internal details of the active protocol. The estimated transfer time is also affected by two approximate overhead factors that are specific to the type of protocol. See the Overhead and TurnDelay properties for more information about these factors. If the modem data rate is different from the comport data rate, also see ActualBPS.

To compute the transfer time, EstimateTransferSecs first computes an effective transfer rate using the following formulas:

```
ActualCPS      = ActualBPS div 10
Efficiency     = ratio of data bytes to highest possible number of
                  bytes, calculated as follows:
                  BlockLength
                  -----
                  BlockLength + Overhead + ((TurnDelay * ActualCPS)
                  div 1000)
EffectiveCPS   = ActualCPS * Efficiency
```

Then the estimated transfer time is Size divided by EffectiveCPS.

The following example calls EstimateTransferSecs in a status routine to get the total and remaining transfer times:

```
procedure TForm1.ProtocolStatus(CP : TObject; Options : Word);
var
    TotalTime, RemainingTime : LongInt;
begin
    with TApdProtocol1(CP) do begin
        ...
        TotalTime := EstimateTransferSecs(FileLength);
        RemainingTime := EstimateTransferSecs(BytesRemaining);
        ....
    end;
end;
```

See also: ActualBPS, OnProtocolStatus, Overhead, TurnDelay

FileDate	read-only, run-time property
-----------------	-------------------------------------

```
property FileDate : TDateTime
```

↪ Returns the date and time of the file being transferred.

For transmitted files the file timestamp is always known. For received files the timestamp is known only if the protocol supports this feature and the receiver has received this information. FileDate is accurate after FileName returns a non-empty string.

If the timestamp is not known, FileDate returns zero.

This property is most useful within an OnProtocolStatus event handler. See “Protocol status” on page 489 for more information.

See also: FileLength, FileName

FileLength

read-only, run-time property

```
property FileLength : LongInt
```

↳ Returns the size of the file being transferred.

For transmitted files the file size is always known. For received files the file size is known only if the protocol supports this feature and the receiver has received this information. FileLength is known after FileName returns a non-empty string. If the file size is not known, FileLength returns zero.

This property is most useful within an OnProtocolStatus event handler. See “Protocol status” on page 489 for more information.

See also: FileDate

FileMask

property

```
property FileMask : TFileName
```

↳ Determines the file mask to use when transmitting files.

FileMask can specify a single file or can contain DOS wildcards to transmit multiple files using a batch protocol such as Zmodem. If it does not specify a drive and directory, files are read from the current directory.

Only a single mask can be used for each transfer. To transfer a group of files that cannot be described by a single mask, see “NextFile processing” on page 494.

The following example transmits all files with a ZIP extension in the C:\UPLOAD directory:

```
ApdProtocol1.FileMask := 'C:\UPLOAD\*.ZIP';  
ApdProtocol1.StartTransmit;
```

See also: Batch

FileName

property

```
property FileName : string
```

↳ Determines the name of the file currently being received.

This should be considered a read-only property for all protocols except Xmodem and ASCII, which do not transfer a filename along with the file data. For these two protocols you must assign a value to FileName before calling StartReceive. For the remaining protocols supported by Async Professional, you can read the value of FileName within a protocol status routine to obtain the file name transferred by the protocol.

If FileName does not include drive or path information, the incoming file is stored in the current directory or the directory specified by DestinationDirectory. If FileName includes drive and/or path information and HonorDirectory is True, the incoming file is stored in that directory regardless of whether a value was assigned to DestinationDirectory.

The following example stores a file received via Xmodem to C:\DOWNLOAD\RECEIVE.TMP:

```
ApdProtocol1.ProtocolType := ptXmodem;  
ApdProtocol1.FileName := 'C:\DOWNLOAD\RECEIVE.TMP';  
ApdProtocol1.StartReceive;
```

See also: DestinationDirectory, HonorDirectory

FinishWait

property

property FinishWait : Word

Default: 364

↪ Determines how long the receiver waits for an end-of-transmission signal before timing out.

This property applies only to Xmodem, Ymodem, and Zmodem protocols.

At the end of an Xmodem or Ymodem file transfer the transmitter sends an <EOT> to the receiver to signal the end of the file and then waits FinishWait ticks (20 seconds by default) for a response. Normally this provides ample time. However, when Xmodem1KG and YmodemG are used over links with long propagation times or slow receivers, the default value might not be enough. Use FinishWait to extend the amount of time that the transmitter waits before timing out and reporting an error. Note that FinishWait is specified in ticks.

Similarly, in a Zmodem transfer the transmitter sends a ZFin packet to the receiver to signal the end of the file and then waits FinishWait ticks to receive an acknowledgement before timing out.

See also: ZmodemFinishRetry

HandshakeRetry

property

property HandshakeRetry : Word

Default: 10

↪ Determines the retry count for protocol handshaking.

This property controls how many times each protocol attempts to detect the initial handshake from its remote partner. HandshakeRetry applies to all protocols but ASCII, which does not perform handshaking.

See also: HandshakeWait

HandshakeWait

property

property HandshakeWait : Word

Default: 182

↪ Determines the wait between retries for protocol handshaking.

This property is the number of ticks a protocol waits when a handshake attempt fails before it tries again. HandshakeWait applies to all protocols but ASCII, which does not perform handshaking.

See also: HandshakeRetry

HonorDirectory

property

property HonorDirectory : Boolean

Default: False

↪ Determines whether a protocol honors the directory name of a file being received.

If HonorDirectory is set to True, a received file is stored in the directory specified by the transmitter, unless that directory does not already exist, in which case it is stored in the current directory or the DestinationDirectory. If HonorDirectory is set to False, the transmitter's directory is ignored.

See also: IncludeDirectory

IncludeDirectory

property

property IncludeDirectory : Boolean

Default: False

↪ Determines whether the complete pathname is transmitted.

If IncludeDirectory is set to True, the protocol sends the drive and directory along with the file name of each file it transmits. The receiver might use or ignore this information. If IncludeDirectory is False, only the file name is transmitted, even if the file is not found in the current directory.

See also: HonorDirectory

InitialPosition

read-only, run-time property

property InitialPosition : LongInt

↪ The initial file offset for a resumed transfer.

This property applies only to Zmodem protocols, which support resumed file transfers. For a transfer from scratch, InitialPosition returns zero. For a resumed transfer, InitialPosition returns the offset where the transfer was resumed. This offset should be subtracted from BytesTransferred to obtain the actual number of bytes transferred during the resumed session.

This property is most useful within an OnProtocolStatus event handler. See “Protocol status” on page 489 for more information.

The following example shows how to compute the character per second transfer rate in a protocol status routine. The constant values are used to convert ticks to seconds. Note that the same expression is valid whether or not the transfer has been resumed.

```
CPS :=  
  (91*(ApdProtocol.BytesTransferred-ApdProtocol.InitialPosition))  
  div (5*ApdProtocol.ElapsedTicks);
```

See also: BytesTransferred

```
property InProgress : Boolean
```

↳ Returns True if a protocol is currently in progress.

A property such as this is important because Async Professional protocols run in the background. A call to `StartTransmit` or `StartReceive` returns immediately to your code.

`InProgress` is True immediately after `StartTransmit` or `StartReceive` is called. `InProgress` is False immediately before the `OnProtocolFinish` event is generated.

Use `InProgress` to determine whether a file transfer is already taking place or not before trying to start another transfer.

See also: `OnProtocolFinish`

KermitCtlPrefix**property**

```
property KermitCtlPrefix : Char
```

Default: '#'

↳ Determines the character Kermit uses to quote control characters.

See "Character quoting" on page 513 for more information.

See also: `KermitHighbitPrefix`, `KermitRepeatPrefix`

KermitHighbitPrefix**property**

```
property KermitHighbitPrefix : Char
```

Default: 'Y'

↳ Determines the technique Kermit uses to quote characters that have their eighth bit set.

The value specified by this property is not always transmitted literally as a quote character. If it equals 'Y', the default, it means that the protocol won't use high bit quoting unless the remote requires it, in which case it uses the prefix character requested by the remote.

If `KermitHighbitPrefix` equals '&' or is in the ASCII range 33-62 or 96-126, it indicates that the protocol requires high bit quoting and that its value is the character used for the prefix.

If C equals 'N' or any other value not listed here, the protocol won't use high bit quoting at all, even if the remote requests it.

See "Character quoting" on page 513 for more information.

See also: `KermitCtlPrefix`, `KermitRepeatPrefix`

KermitLongBlocks

read-only, run-time property

property KermitLongBlocks : Boolean

↳ Returns True if Kermit long packets are in use.

See also: KermitMaxLen

KermitMaxLen

property

property KermitMaxLen : Word

Default: 80

↳ Determines the maximum number of bytes in one Kermit packet.

The normal maximum value is 94, but the default value of 80 is suggested by the Kermit Protocol Manual. If KermitMaxLen is set to a value in the range of 95 to 1024, long packets are enabled with the specified packet size. As with other Kermit settings, however, long packets will be used only if the remote partner also supports it.

See “Kermit options” on page 515 for more information.

See also: KermitMaxWindows

KermitMaxWindows

property

property KermitMaxWindows : Word

Default: 0

↳ Determines whether Kermit sliding windows control is enabled.

If KermitMaxWindows is set to a value between 1 and 27, sliding windows are enabled with the specified window count. This allows a Kermit transmitter to send additional packets without waiting for an acknowledgement from the receiver, thus improving throughput. As with other Kermit settings, however, sliding windows control will be used only if the remote partner also supports it.

See “Sliding Windows Control” on page 518 for more information.

See also: KermitWindowsTotal, KermitWindowsUsed

KermitPadCharacter

property

`property KermitPadCharacter : Char`

Default: ' ' (ASCII 32)

↪ Determines the character that Kermit uses to pad the beginning of each packet.

See “Kermit options” on page 515 for more information.

See also: KermitTerminator

KermitPadCount

property

`property KermitPadCount : Word`

Default: 0

↪ Determines the number of pad characters that Kermit transmits at the beginning of each packet.

See “Kermit options” on page 515 for more information.

See also: KermitPadCharacter

KermitRepeatPrefix

property

`property KermitRepeatPrefix : Char`

Default: '~'

↪ Determines the prefix that Kermit uses when compressing strings of repeated characters.

When Kermit sees four or more equal and adjacent characters, it compresses the sequence into a quote character (KermitRepeatPrefix), a length byte, and the repeated character. The default quote character rarely needs to be changed.

See “Character quoting” on page 513 for more information.

See also: KermitCtlPrefix, KermitHighbitPrefix

property KermitSWCTurnDelay : Word

Default: 0

- ↳ Determines the turnaround delay used by EstimateTransferSecs when a Kermit sliding windows protocol is in use.

This property is the time in milliseconds for a data block to transit from the sender to the receiver, for the receiver to send an acknowledgement, and for the acknowledgement to arrive back at the sender. It is used by the EstimateTransferSecs method to estimate the time to transfer a given amount of data.

When Kermit sliding windows control is enabled, the transmitter does not generally wait for acknowledgement of a packet before sending the next one. Hence, an appropriate default is zero milliseconds.

EstimateTransferSecs uses the value of the TurnDelay property for Kermit transfers when sliding windows control is not enabled, and the KermitSWCTurnDelay property when it is enabled.

See also: Overhead, TurnDelay

KermitTerminator**property**

property KermitTerminator : Char

Default: ^M (ASCII 13)

- ↳ Determines the character used to terminate a Kermit data packet.

This character is used only by Kermit hosts that cannot start processing a data line until a terminating character is received.

See “Kermit options” on page 515 for more information.

See also: KermitPadCharacter

KermitTimeoutSecs

property

property KermitTimeoutSecs : Word

Default: 5

↳ Determines how long Kermit waits for the next expected byte.

If a Kermit transmitter waits more than KermitTimeoutSecs for an acknowledgement, it resends the last packet. If a Kermit receiver waits more than KermitTimeoutSecs for the next byte, it sends an error packet to the transmitter.

See also: TransmitTimeout

KermitWindowsTotal

read-only, run-time property

property KermitWindowsTotal : Word

↳ Returns the total number of Kermit sliding windows negotiated for the current transfer.

If sliding windows control is disabled, KermitWindowsTotal returns 0.

See also: KermitMaxWindows, KermitWindowsUsed

KermitWindowsUsed

read-only, run-time property

property KermitWindowsUsed : Word

↳ Returns the number of Kermit sliding windows that currently contain data.

If sliding windows control is disabled, KermitWindowsUsed returns 0.

See also: KermitMaxWindows, KermitWindowsTotal

```
property OnProtocolAccept : TProtocolAcceptEvent  
  
TProtocolAcceptEvent = procedure(CP : TObject;  
    var Accept : Boolean; var FName : TPassString) of object;  
  
TPassString = string[255];
```

↳ Defines an event handler that is called as soon as the name of an incoming file is known.

This event handler provides an opportunity for the receiver to reject or rename the incoming file. If an OnProtocolAccept handler is not installed, all files are accepted (subject to the setting of the WriteFailAction property).

CP is the protocol component that is receiving the file. The event handler should set Accept to True to accept the file, False to reject it. FName is the name of the file to be received. The event handler can change the name if, for example, it would overwrite an existing file.

See “AcceptFile processing” on page 495 for more information.

See also: OnProtocolNextFile, WriteFailAction

OnProtocolError

```
property OnProtocolError : TProtocolErrorEvent  
  
TProtocolErrorEvent = procedure(  
    CP : TObject; ErrorCode : SmallInt) of object;
```

↳ Defines an event handler that is called when an unrecoverable protocol error occurs.

This event is generated only for unrecoverable errors. Most protocol errors caused by line noise are handled automatically by the protocol and are not reported to this event handler.

CP is the protocol component that generated the error. ErrorCode is a number indicating the type of error.

Note that the OnProtocolFinish event is generated soon after the OnProtocolError event and passes the same error code. OnProtocolFinish is generated for both successful and failed transfers, so you may want to use it instead of an OnProtocolError handler.

See “Error handling” on page 488 for more information.

See also: BlockErrors, OnProtocolFinish

```
property OnProtocolFinish : TProtocolFinishEvent
```

```
TProtocolFinishEvent = procedure(
  CP : TObject; ErrorCode : SmallInt) of object;
```

↳ Defines an event handler that is called when a protocol transfer ends.

This event is generated whether the protocol ends successfully or not. If it ends successfully, ErrorCode is zero. Otherwise, ErrorCode is a number indicating the type of error. CP is the protocol component that generated the error.

An application could use this handler to display a completion dialog box (needed only if a protocol status event handler is not also in use) or to enable the scheduling of another file transfer.

The following example displays a message whenever a protocol finishes, and enables an associated terminal window to accept data again:

```
procedure TForm1.ApProtocol1ProtocolFinish(
  CP : TObject; ErrorCode : SmallInt);
begin
  ShowMessage('Protocol finished: '+ErrorMsg(ErrorCode));
  ApTerminal1.Active := True;
end;
```

See also: InProgress, OnProtocolError

```
property OnProtocolLog : TProtocolLogEvent
```

```
TProtocolLogEvent = procedure(CP : TObject; Log : Word) of object;
```

↳ Defines an event handler that is called at well-defined points during a protocol transfer.

The primary purpose of this event is to give applications a chance to log statistical information about file transfers such as the transfer time and whether they succeeded or failed. Applications can also use this event for start-up and cleanup activities such as deleting partial files after unsuccessful downloads.

CP is the protocol component that needs to be logged. Log is a code that indicates the current state of the file transfer. The possible states are:

Log	State
lfReceiveStart	File receive is starting.
lfReceiveOK	File was received successfully.
lfReceiveFail	File receive failed.
lfReceiveSkip	File was skipped (rejected by receiver).
lfTransmitStart	File transmit is starting.
lfTransmitOK	File was transmitted successfully.
lfTransmitFail	File transmit failed.
lfTransmitSkip	File was skipped (rejected by receiver).

No other information is passed along with the event. Use protocol status properties such as FileName and ElapsedTicks to get additional information about the state of the transfer.

See “Protocol logging” on page 493 for more information.

See also: ProtocolLog

OnProtocolNextFile

event

```
property OnProtocolNextFile : TProtocolNextFileEvent
TProtocolNextFileEvent = procedure(
    CP : TObject; var FName : TPassString) of object;
TPassString = string[255];
```

↳ Defines an event handler that is called to determine the next file to transmit in a batch transfer.

If no handler is installed for this event, Async Professional transmits the files that match the DOS filemask assigned to the FileMask property. If you need to transmit a batch of files that cannot be described by a single filemask, you need to install an event handler for OnProtocolNextFile.

CP is the protocol component that is transmitting. The event handler should return the next file to transmit in FName, or an empty string to terminate the batch.

See “NextFile processing” on page 494 for more information.

See also: FileMask

```
property OnProtocolStatus : TProtocolStatusEvent
```

```
TProtocolStatusEvent = procedure(  
    CP : TObject; Options : Word) of object;
```

↳ Defines an event handler that is called regularly during a file transfer.

This event is generated for each block transmitted or received, after the completion of each major operation (e.g., renaming a file, detecting an error, ending the transfer), and at intervals of StatusInterval ticks (by default 18 ticks, or about 1 second). The program can use it to update a status display that informs the user about the protocol progress.

CP is the protocol component that is in progress. A number of the properties of this component can be read to establish the status of the transfer. Options is set to apFirstCall (1) on the first call to the handler, apLastCall (2) on the last call to the handler, and zero on all other calls.

A predefined status component called TApdProtocolStatus is supplied with Async Professional. For a standard protocol status window you can simply create an instance of this component and assign it to the StatusDisplay property of the TApdProtocol component. If you do so, there is no need to supply your own OnProtocolStatus event handler.

See “Protocol status” on page 489 for more information.

See also: StatusDisplay, StatusInterval

Overhead

property

```
property Overhead : Word
```

↳ Determines the number of overhead bytes per data block used by EstimateTransferSecs.

When a protocol transfers a data block, not all of the bytes are actually data from the file being transferred. Some of them are part of the packet header and others may be used to quote or escape data characters that cannot be transmitted as-is.

When you select a protocol by assigning to the ProtocolType property, the TApdProtocol component assigns a default value to Overhead that matches the characteristics of the protocol. For some protocols the exact value of Overhead depends on the actual data being transmitted, but the default is a reasonable estimate that gives reasonable transfer time estimates. If the estimates are consistently in error, you can assign a new value to Overhead.

See also: EstimateTransferSecs, TurnDelay

ProtocolError

read-only, run-time property

property ProtocolError : Integer

↳ Returns the code of the last error returned by the protocol.

This property returns zero except for the first call after an error is encountered. See “Error Handling and Exception Classes” on page 900 for a complete list of error codes.

See also: ProtocolStatus

ProtocolLog

property

property ProtocolLog : TApdProtocolLog

↳ An instance of a protocol logging component.

If ProtocolLog is nil, as it is by default, the protocol does not perform any automatic logging. You can install an OnProtocolLog event handler to perform logging in this case.

If you create an instance of a TApdProtocolLog class (see page 583), or a descendant thereof, and assign it to ProtocolLog, the protocol will log itself automatically.

ProtocolStatus

read-only, run-time property

property ProtocolStatus : Word

↳ Returns a code that indicates the current state of the protocol.

This property is most useful within an OnProtocolStatus event handler. See “Protocol status” on page 489 for more information.

See also: ProtocolError


```
property ProtocolType : TProtocolType
```

```
TProtocolType = (ptNoProtocol, ptXmodem, ptXmodemCRC, ptXmodem1K,  
    ptXmodem1KG, ptYmodem, ptYmodemG, ptZmodem, ptKermit, ptAscii);
```

Default: ptZmodem

✚ Determines the type of file transfer protocol.

Async Professional encapsulates all of the file transfer protocols that it supports into a single component. To select a particular type of protocol, you must assign the desired type to the ProtocolType property. You should generally assign to ProtocolType shortly after creating the TApdProtocol component and before assigning other properties, since various defaults are assigned whenever you change ProtocolType, and some properties are valid only when ProtocolType has a particular value.

Assigning a new value to ProtocolType first deallocates any protocol-specific memory used by the prior protocol, then allocates and initializes any structures required by the current protocol.

You should generally not assign ptNoProtocol to ProtocolType, but it can be used to deallocate previous protocol memory while temporarily not allocating new protocol memory.

See also: BlockCheckMethod

```
property RTSLowForWrite : Boolean
```

Default: False

✚ Determines whether protocols force RTS low while writing received data to disk.

When RTSLowForWrite is set to True, hardware flow control is used to prevent the transmitter from sending additional data while the receiver writes data to disk. As soon as the disk write is finished, RTS is raised again. This feature might be required if other Windows applications are being run at the same time as a protocol transfer or if the disk driver leaves interrupts disabled for an excessive time.

In order for this option to be effective, disk write caching must be disabled.

If the protocol is transferring files using a modem, it might also be necessary to configure the modem to react correctly to the RTS signal.

```
procedure StartReceive;
```

↳ Tells the protocol to start receiving files.

The steps leading up to calling StartReceive look something like this:

1. Create a port component.
2. Create a protocol component.
3. Set ProtocolType.
4. Set other properties to customize the protocol.
5. Write suitable handlers for protocol events.
6. Call StartReceive.

StartReceive returns immediately and receives files in the background, occasionally generating events to keep the application apprised of progress. When the protocol is finished, either successfully or with a fatal error, it generates an OnProtocolFinish event and its InProgress property starts returning False.

See also: ProtocolType, StartTransmit

```
procedure StartTransmit;
```

↳ Tells the protocol to start transmitting files.

The steps leading up to calling StartTransmit look something like this:

1. Create a port component.
2. Create a protocol component.
3. Set ProtocolType.
4. Set other properties to customize the protocol.
5. Write suitable handlers for protocol events.
6. Set FileMask or use an OnProtocolNextFile event handler to return a list of files to transmit.
7. Call StartTransmit.

StartTransmit returns immediately and transmits files in the background, occasionally generating events to keep the application apprised of progress. When the protocol is finished, either successfully or with a fatal error, it generates an OnProtocolFinish event and its InProgress property starts returning False.

See also: FileMask, OnProtocolNextFile, ProtocolType, StartReceive

StatusDisplay **property**

property StatusDisplay : TApdAbstractStatus

↳ An instance of a protocol status window.

If StatusDisplay is nil, as it is by default, the protocol does not provide an automatic status window. You can install an OnProtocolStatus event handler to display status in this case.

If you create an instance of a class derived from TApdAbstractStatus, such as the provided TApdProtocolStatus component (see page 582), and assign it to ProtocolStatus, the status window will be displayed and updated automatically.

StatusInterval **property**

property StatusInterval : Word

Default: 18

↳ The maximum number of clock ticks between OnProtocolStatus events.

The OnProtocolStatus event is generated for each block transmitted or received, after the completion of each major operation (e.g., renaming a file, detecting an error, ending the transfer), and at intervals of StatusInterval ticks.

This property also determines how frequently the StatusDisplay window is updated.

See also: OnProtocolStatus, StatusDisplay

StatusMsg

method

```
function StatusMsg(const Status : Word) : string;
```

↳ Returns an English string for a protocol status code.

This routine is intended primarily for use in protocol status routines. It returns a status string from the string table resource linked into your EXE. The string ID numbers correspond to the values of the psXxx protocol status constants (see page 491). If the string table doesn't contain a string resource with the requested ID, an empty string is returned.

The returned string is never longer than MaxMessageLen (80) characters.

See also: ProtocolStatus

TotalErrors

read-only, run-time property

```
property TotalErrors : Word
```

↳ The number of errors encountered since the current file transfer was started.

This error count is reset whenever a new file is started. This property is most useful within an OnProtocolStatus event handler. See “Protocol status” on page 489 for more information.

See also: BlockErrors

TransmitTimeout

property

```
property TransmitTimeout : Word
```

Default: 1092

↳ Determines the maximum time a sender will wait for the receiver to release flow control.

If the receiver blocks flow control for longer than TransmitTimeout ticks (60 seconds by default), the protocol is aborted.

```
property TurnDelay : Word
```

Default: 0

- ↳ Determines the turnaround delay, in milliseconds, per data block used by EstimateTransferSecs.

When a protocol transfers a data block, the transmitter must often wait for an acknowledgement from the receiver before it transmits the next block. This delay slows down the overall throughput of the protocol and must be accounted for by EstimateTransferSecs.

When you select a protocol by assigning to the ProtocolType property, the TAPdProtocol component assigns a default value to TurnDelay that is a good estimate for the given protocol. However, the actual TurnDelay often depends on the characteristics of the communications link between the sender and receiver (e.g., a satellite link would impose a longer delay than a null modem cable). If the values returned by EstimateTransferSecs are consistently in error, you can assign a new value to TurnDelay.

See also: EstimateTransferSecs, Overhead

```
property UppcaseFileNames : Boolean
```

Default: True

- ↳ Determines whether the protocol converts file names to upper case.

Applications provide file names to protocols in the OnProtocolNextFile event or by setting the FileName property. File names can also be received as part of the protocol transfer. Because the DOS/16-bit Windows file system stores all file names in upper case, the protocol component converts file and path names to uppercase.

Windows 95/98 and Windows NT preserve the specified case in file names, although they don't normally use case to distinguish between file names. For example, the file name "MixCase.Txt" is stored by the file system with the upper and lower case characters preserved, however, it can be accessed by any combination of upper and lower case (e.g., "MIXCASE.TXT" or "mIXCAsE.tXt"). If you want to display the preserved case in status and log routines, set UppcaseFileNames to False.

WriteFailAction

property

```
property WriteFailAction : TWriteFailAction
```

```
TWriteFailAction = (  
    wfWriteNone, wfWriteFail, wfWriteRename, wfWriteAnyway);
```

Default: wfWriteRename

↪ Determines the receiver's behavior when the destination file already exists.

You should assign one of the following values to WriteFailAction:

Value	Description
wfWriteFail	Fail the receive attempt.
wfWriteRename	Rename the incoming file.
wfWriteAnyway	Overwrite the existing file.

When wfWriteRename is selected and the destination file already exists, the first character in the incoming file name is replaced with '\$' (e.g., "SAMPLE.DOC" becomes "\$SAMPLE.DOC"). If that renamed file already exists, it is overwritten without warning.

The logic that handles these overwrite options is executed after the OnProtocolAccept event has been generated. If you write an event handler that deals with possible overwrites, be sure to set WriteFailAction to wfWriteAnyway before starting a transfer.

See also: OnProtocolAccept, ZmodemFileOption

XYmodemBlockWait

property

```
property XYmodemBlockWait : Word
```

Default: 91

↪ Determines the number of ticks Xmodem and Ymodem wait between blocks for a response from the remote.

If the wait exceeds XYmodemBlockWait ticks, a sending protocol retransmits the block and a receiving protocol aborts the transfer. The default wait is about 5 seconds.

See also: TransmitTimeout

```
property Zmodem8K : Boolean
```

Default: False

↳ Determines whether 8K blocks are enabled.

See “Large block support” on page 512 for more information.

ZmodemFileOption

property

```
property ZmodemFileOption : TZmodemFileOptions
```

```
TZmodemFileOptions = (zfoNoOption, zfoWriteNewerLonger,  
    zfoWriteCrc, zfoWriteAppend, zfoWriteClobber, zfoWriteNewer,  
    zfoWriteDifferent, zfoWriteProtect);
```

Default: zfoWriteNewer

↳ Determines the Zmodem file management options to use.

It should be assigned one of the following values:

Value	Description
zfoWriteNewerLonger	Transfer if new, newer or longer.
zfoWriteCrc	Not supported, treated same as WriteNewer.
zfoWriteAppend	Transfer if new, append if exists.
zfoWriteClobber	Transfer regardless.
zfoWriteNewer	Transfer if new or newer.
zfoWriteDifferent	Transfer if new or different dates or lengths.
zfoWriteProtect	Transfer only if new.

Regardless of the value of this property, new incoming files are accepted unless the ZmodemSkipNoFile property is set to False.

The logic that handles these file management options is executed after the OnProtocolAccept event has been generated. If you write an event handler that deals with possible overwrites, be sure to set ZmodemFileOption to zfoWriteClobber before starting to receive.

See also: ZmodemOptionOverride, ZmodemSkipNoFile

```
property ZmodemFinishRetry : Word;
```

Default: 0

↪ Specifies the number of times to retry the final handshake of a Zmodem protocol session.

A Zmodem transmitter signals that it has no more files to transmit by sending a ZFin frame. The receiver acknowledges this by sending its own ZFin frame. The transmitter then sends “OO” as the final frame of the transfer.

The Zmodem specification indicates that this portion of the protocol isn’t critical (since all files have already been completely received) and that a timeout while waiting for the response should be ignored. However, this strategy doesn’t work well with DSZ, a Zmodem implementation by Omen Technology, Inc.

DSZ retries after a ZFin timeout, which can sometimes cause unneeded packet transfers when the handshake timeout is 10 seconds or less. To handle this situation, Async Professional mimics DSZ when ZmodemFinishRetry is set a non-zero value. It waits FinishWait ticks for a response.

ZmodemFinishRetry is the number of times to resend the ZFin in response to a timeout. When ZmodemFinishRetry is zero the ZFin is sent only once. If no response is received the protocol finishes without an error.

See also: FinishWait

```
property ZmodemOptionOverride : Boolean
```

Default: False

↪ Determines whether a remote sender’s options are ignored.

If ZmodemOptionOverride is set to True, a receiving protocol component ignores the sender’s options and uses its own settings for ZmodemFileOption and ZmodemSkipNoFile. Otherwise, it uses the sender’s options.

See also: ZmodemFileOption

ZmodemRecover

property

```
property ZmodemRecover : Boolean
```

Default: False

↳ Determines whether Zmodem performs file recovery.

Zmodem is capable of resuming interrupted file transfers if the receiver kept the partial file when a previous transfer was interrupted. The transmitter requests this action by setting `ZmodemRecover` to `True`. The request is transmitted to the receiver along with the file name to be recovered. If the receiver has this file, it sends back the current file size. The transmitter then adjusts its file offset and starts sending data from that point. If the receiver doesn't already have this file, a normal file transfer takes place.

See “Transfer resume” on page 510 for more information.

See also: `InitialPosition`

ZmodemSkipNoFile

property

```
property ZmodemSkipNoFile : Boolean
```

Default: False

↳ Determines whether a Zmodem receiver should skip all files that don't already exist.

See also: `ZmodemFileOption`

TApdFtpClient Component

The `TApdFtpClient` component is a specialized `TApdWinsockPort` that implements client-side file transfer protocol (FTP) capabilities as defined by RFC 959. `TApdFtpClient` presents an intuitive interface that makes it easy to navigate and manipulate directories and files on an FTP server.

Connecting and logging on to an FTP server is performed by the `Login` method. Logging off and disconnecting is performed by `Logout`. Directory manipulation is performed by the `ChangeDir`, `Delete`, `ListDir`, `MakeDir`, and `Rename` methods. File transfer and manipulation is performed by the `Delete`, `Rename`, `Retrieve`, and `Store` methods. Server status and help information are performed by the `Help`, and `Status` methods, and an arbitrary FTP command string can be sent to the connected FTP server via `SendFtpCommand`.

Only one FTP operation is allowed at any given time, however these methods operate in an asynchronous (i.e., non-blocking) fashion. This means that when a method is called to initiate an FTP operation, it returns immediately and the operation is performed in the background. When the operation is completed, the `OnFtpStatus` event is fired to notify the application. During file transfer operations, the `OnFtpStatus` event is fired periodically to provide status updates.

Hierarchy

TComponent (VCL)

❶ TApdBaseComponent (OOMisc)	8
❷ TApdComPort (AdPort)	22
❸ TApdWinsockPort (AdWnPort)	106
TApdCustomFtpClient (AdFtp)	
TApdFtpClient (AdFtp)	

Properties

Account	❷ HWFlowOptions	ServerAddress
❷ AutoOpen	❷ InBuffFree	❷ StopBits
❷ BaseAddress	❷ InBuffUsed	❷ SWFlowOptions
❷ Baud	InProgress	❷ TapiMode
❷ BufferFull	❷ InSize	❷ TraceAllHex
BytesTransferred	❷ LineBreak	❷ TraceName
❷ ComHandle	❷ LineError	❷ TraceSize
Connected	❷ LogHex	❷ Tracing
ConnectTimeout	❷ LogName	TransferTimeout
❷ CTS	❷ LogSize	❷ UseEventWord
❷ DataBits	❷ ModemStatus	UserLoggedIn
❷ DCD	❸ Open	UserName
❷ DeltaCTS	PassiveMode	❶ Version
❷ DeltaDCD	Password	❸ WsAddress
❷ DeltaDSR	❷ OutBuffFree	❸ WsLocalAddresses
❷ DeltaRI	❷ OutBuffUsed	❸ WsLocalAddressIndex
❸ DeviceLayer	❷ Output	❸ WsMode
❷ DSR	❷ OutSize	❸ WsPort
❷ DTR	❷ Parity	❸ WsSocksServerInfo
FileLength	RestartAt	❸ WsTelnet
FileType	❷ RI	❷ XOffChar
❷ FlowState	❷ RS485Mode	❷ XOnChar
FtpLog	❷ RTS	

Methods

- Abort
- ② ActiveDeviceLayer
- ② AddDataTrigger
- ② AddStatusTrigger
- ② AddTimerTrigger
- ② AddTraceEntry
- ChangeDir
- ② CharReady
- ② CheckForString
- CurrentDir
- Delete
- ② FlushInBuffer
- ② FlushOutBuffer
- ② ForcePortOpen
- ② GetBlock
- ② GetChar
- Help
- ② InitPort
- ListDir
- Login
- Logout
- MakeDir
- ② PeekBlock
- ② PeekChar
- ② ProcessCommunications
- ② PutBlock
- ② PutChar
- ② PutString
- ② RemoveAllTriggers
- ② RemoveTrigger
- Rename
- Retrieve
- ② SendBreak
- SendFtpCommand
- ② SetBreak
- ② SetStatusTrigger
- ② SetTimerTrigger
- Status
- Store

Events

- OnFtpError
- OnFtpLog
- OnFtpReply
- OnFtpStatus
- ② OnPortClose
- ② OnPortOpen
- ② OnTrigger
- ② OnTriggerAvail
- ② OnTriggerData
- ② OnTriggerLineError
- ② OnTriggerModemStatus
- ② OnTriggerOutbuffFree
- ② OnTriggerOutbuffUsed
- ② OnTriggerOutSent
- ② OnTriggerStatus
- ② OnTriggerTimer
- ③ OnWsAccept
- ③ OnWsConnect
- ③ OnWsDisconnect
- OnWsError

Reference

Abort

method

```
procedure Abort;
```

↳ Terminates a file transfer in progress.

Calling Abort stops a file transfer that is in progress.

See also: Retrieve, Store

Account

property

```
property Account : string
```

↳ Specifies the user's account information.

Account information is required by some FTP servers for login or storing files. If the server requests account information to complete an operation, the Account string is automatically sent to the server.

See also: Password, Login, UserName

BytesTransferred

read-only, run-time property

```
property BytesTransferred : Longint
```

↳ The number of bytes transferred so far in the current file.

When sending a file, BytesTransferred is the number of bytes written to the Winsock buffer, which may be less than the number of bytes actually transmitted to the server. When receiving a file, BytesTransferred is the actual number of bytes received. This is useful within an OnFtpStatus event handler when the status code is scProgress.

See also: OnFtpStatus

```
function ChangeDir(const RemotePathName : string) : Boolean;
```

- ↪ Changes the current working directory on the FTP server.

RemotePathName specifies the directory at the server. If RemotePathName contains an empty string, or a change working directory operation is not allowed given the current protocol state, ChangeDir returns False, otherwise True is returned and the operation is initiated.

If the directory is successfully changed at the server, the OnFtpStatus event is fired with the scComplete status code. If the directory operation is rejected by the server, the OnFtpError event is fired and the operation is terminated.

See also: OnFtpError, OnFtpStatus

```
property Connected : Boolean
```

- ↪ Indicates whether a connection to an FTP server has been established.

While a control connection to an FTP server is open, Connected will return True. This does not indicate that the user is currently logged in to the server. To determine if the user is logged in, use the UserLoggedIn property.

See also: UserLoggedIn

```
property ConnectTimeout : Integer
```

Default: 0

- ↪ ConnectTimeout determines the connection timeout when establishing the control connection.

When establishing the initial control connection to the FTP server, the ConnectTimeout property determines the timeout (in ticks) associated with the connection attempt. If ConnectTimeout is 0 (the default), the TApdFTPClient will not timeout. If ConnectTimeout > 0, the connection attempt will be terminated if a connection is not made within ConnectTimeout ticks. If a timeout occurs, the OnFTPError event is generated with ErrorCode = ecFtpConnectTimeout.

See also: Login

```
function CurrentDir : Boolean;
```

↳ Obtains the path name of the current working directory from the FTP server.

If a directory operation is not allowed given the current protocol state, `CurrentDir` returns `False`, otherwise `True` is returned and the operation is initiated. When the server responds with the requested information, the `OnFtpStatus` event is fired with the `csCurrentDir` command status code, and the `InfoText` parameter will point to a null terminated string containing the path name of the current working directory.

If the operation is rejected by the server, the `OnFtpError` event is fired and the operation is terminated.

See also: `OnFtpError`, `OnFtpStatus`

```
function Delete(const RemotePathName : string) : Boolean;
```

↳ Removes a file or directory at an FTP server.

`RemotePathName` specifies the file or directory at the server. If `RemotePathName` is an empty string or a delete operation is not allowed given the current protocol state, `Delete` returns `False`, otherwise `True` is returned and the operation is initiated.

If the file or directory is successfully deleted at the server, the `OnFtpStatus` event is fired with the `scComplete` status code. If the delete operation is rejected by the server, the `OnFtpError` event is fired and the operation is terminated.

See also: `OnFtpError`, `OnFtpStatus`

```
property FileLength : Longint
```

↳ Returns the size of the file being transferred.

If the file size is not known, `FileLength` returns zero. This property is most useful within an `OnFtpStatus` event handler when the status code is `scProgress`.

See also: `BytesTransferred`, `OnFtpStatus`

FileType

property

```
property FileType : TFTPFileType  
TFTPFileType = (ftAscii, ftBinary);
```

Default: ftAscii

↪ Specifies the file data type.

Transferring a file in the wrong format can damage the file so that it becomes unusable. This is particularly true of binary files, which, if transferred using ASCII format, are no longer usable. Be sure to set FileType prior to initiating a file transfer. For text files use ftAscii; otherwise, use ftBinary.

FtpLog

property

```
property FtpLog : TAPdFtpLog
```

↪ An instance of a FTP logging component.

If FtpLog is nil (the default), TAPdFtpClient does not provide automatic logging. You can install an OnFtpLog event handler to provide logging services in this case.

FtpLog is usually set automatically at design time to the first TAPdFtpLog component that is found on the form. If necessary, use the Object Inspector to select a different logging component.

Setting the FtpLog property at run time is necessary only when using a dynamically created logging component or when selecting among several logging components.

See also: OnFtpLog, TAPdFtpLog


```
function Help(const Command : string) : Boolean;
```

↳ Obtains help information from an FTP server.

If `Command` is not an empty string, the FTP command syntax for the specified command is obtained, otherwise the names of all the commands supported by the server are retrieved. If the help operation is allowed given the current protocol state, `Help` returns `False`, otherwise `True` is returned and the operation is initiated.

If the help operation is successful, the `OnFtpStatus` event is fired with the `csDataAvail` command status code, and the `InfoText` parameter will point to a null terminated string containing the raw text of the help information received from the server.

If the help operation is rejected by the server, the `OnFtpError` event is fired and the operation is terminated.

See also: `OnFtpError`, `OnFtpStatus`

InProgress**read-only, run-time property**

```
property InProgress : Boolean
```

↳ Returns `True` if an FTP operation is currently in progress.

This property is important since a call to initiate a command to an FTP server returns immediately to your code. If you do not use an `OnFtpStatus` event handler to detect when the operation is complete, you can check `InProgress` in a polling loop.

See also: `OnFtpStatus`

ListDir**method**

```
function ListDir(  
    const RemotePathName : string; FullList : Boolean) : Boolean;
```

↳ Obtains a listing of contents of a remote directory.

`RemotePathName` specifies the remote directory at the server. If `RemotePathName` is an empty string, the contents of the current working directory will be obtained. Set `FullList` to `True` to request that full file information for each file in the directory be obtained from the server, otherwise only file names will be obtained. If a list operation is not allowed given the current protocol state, `List` returns `False`, otherwise `True` is returned and the operation is initiated.

If the list operation is successful, the OnFtpStatus event is fired with the csDataAvail command status code, and the InfoText parameter will point to a null terminated string containing the raw text of the directory listing received from the server.

If the list operation is rejected by the server, the OnFtpError event is fired and the operation is terminated.

See also: OnFtpError, OnFtpStatus

Login

method

```
function Login : Boolean;
```

↪ Establishes an FTP session with the FTP server specified by ServerAddress.

The logon procedure consists of opening a port to establish a control connection to an FTP server and logging on to the server with the user identification specified by UserName and Password. The ServerAddress, UserName, and Password properties must be set prior to calling Login. If login is allowed given the current protocol state the function returns True immediately. Otherwise False is returned.

If a connection to the server is established, the OnFtpStatus event is fired with the scOpen status code.

If the server authenticates the user identification, the OnFtpStatus event is fired with the scLogin status code and the UserLoggedIn property is set to True. Otherwise, the OnFtpError event is fired and the connection is left open. Subsequent calls to Login will send the user identification information to the server via the existing connection. Call Logout to close the existing connection.

If a connection to the server cannot be established, an EApdSocket exception is raised.

See also: ConnectTimeout, Logout, OnFtpError, OnFtpStatus, Password, ServerAddress, UserLoggedIn, UserName

```
function Logout : Boolean;
```

↳ Terminates the active FTP session and closes the control connection.

If a file transfer is in progress then the control connection will remain open until the transfer has completed.

If logout is allowed given the current protocol state the function returns `True` immediately. When the user is logged out by the server, the `OnFtpStatus` event is fired with the `scLogout` status code and the `UserLoggedIn` property is set to `False`.

When the control connection port has closed, the `OnFtpStatus` event is fired with the `scClose` status code.

See also: `Login`, `OnFtpStatus`, `UserLoggedIn`

MakeDir**method**

```
function MakeDir(const RemotePathName : string) : Boolean;
```

↳ Creates the specified directory on the FTP server.

`RemotePathName` specifies the new directory at the server. If `RemotePathName` contains an empty string, or a directory operation is not allowed given the current protocol state, `ChangeDir` returns `False`, otherwise `True` is returned and the operation is initiated.

If the directory is successfully created at the server, the `OnFtpStatus` event is fired with the `scComplete` status code. If the directory operation is rejected by the server, the `OnFtpErrors` event is fired and the operation is terminated.

See also: `OnFtpError`, `OnFtpStatus`

OnFtpError**event**

```
property OnFtpError : TFtpErrorEvent
```

```
TFtpErrorEvent = procedure(  
    Sender : TObject; ErrorCode : Integer;  
    ErrorText : PChar) of object;
```

↳ Defines an event handler that is called an FTP protocol error occurs.

The server has rejected the FTP operation attempted and the operation is terminated. `ErrorCode` contains the FTP error code returned by the server, and `ErrorText` points to a null terminated string containing the text of the error.

See also: `FTP Error Codes`

```
property OnFtpLog : TFtpLogEvent

TFtpLogEvent = procedure(
    Sender : TObject; LogCode : TFtpLogCode) of object;

TFtpLogCode = (lcClose, lcOpen, lcLogin,
    lcLogout, lcDelete, lcRename, lcReceive, lcStore,
    lcComplete, lcRestart, lcTimeout, lcUserAbort);
```

↳ Defines an event handler that is called at designated points during an FTP file operation.

The primary purpose of this event is to give the application a chance to log auditing information about file operations during a FTP session.

See also: TApdFtpLog

```
property OnFtpReply : TFtpReplyEvent

TFtpReplyEvent = procedure(Sender : TObject;
    ReplyCode : Integer; ReplyText : PChar) of object;
```

↳ Defines an event handler that is called when an FTP server returns a reply.

An FTP reply consists of a 3-digit alphanumeric code as defined in RFC 959, followed by some text. ReplyCode contains the integer form of the 3-digit alphanumeric code, and ReplyText points to a null terminated string containing the entire reply text.

The primary purpose of this event is to monitor the server's response to the operations initiated by the application. This event can be useful during debugging.

```
property OnFtpStatus : TFtpStatusEvent

TFtpStatusEvent = procedure(Sender : TObject;
    StatusCode : TFtpStatusCode; InfoText : PChar) of object;

TFtpStatusCode = (scClose, scOpen, scLogout, scLogin, scComplete,
    scCurrentDir, scDataAvail, scProgress, scTransferOK, scTimeout);
```

↳ Defines an event handler that is called when the state of the FTP protocol changes.

StatusCode indicates the current state of the FTP client. When StatusCode equals csDataAvail, InfoText points to a null terminated string containing raw text received from the server. Otherwise InfoText is nil.

The following describes the possible status codes:

Status	Meaning
scClose	The control connection port to the FTP server is closed. The Login method must be called to re-connect to the FTP server. All other functions are disabled.
scOpen	Connection to an FTP server has been established and the control port is open. Login, Help, Status, and SendFtpCommand functions are enabled. This status event does not indicate that the server has authenticated the login identification and so should not be used invoke another FTP operation.
scLogin	The FTP server has authenticated the user login identification. All functions except Login are enabled.
scLogout	The FTP server has logged the user out. If this status is the result of a call to Logout, then it will be followed by scClose status when the port closes and should not be used in this case to log on as another user.
scComplete	This status event is fired upon the successful completion of an FTP operation initiated by ChangeDir, Delete, MakeDir, Rename, or SendFtpCommand.
scCurrentDir	Indicates that the current working directory information initiated by CurrentDir is available. InfoText points to the null-terminated text string containing the full path name of the current working directory.
scDataAvail	Indicates that an information request initiated by Help, ListDir, or Status has completed. InfoText points to requested text.
scProgress	This status event is fired periodically during a file transfer initiated by Retrieve or Store. The BytesTransferred property contains the number of file data bytes sent or received so far.
scTransferOk	A file transfer initiated by Retrieve or Store is complete.
scTimeout	The server has not responded during the interval defined by TransferTimeout. The file transfer operation in progress is terminated.

```
property OnWsError : TWsErrorEvent  
  
TWSErrorEvent = procedure (  
    Sender : TObject; ErrorCode : Integer) of object;
```

↳ Defines an event handler that is generated when a Winsock error occurs.

This event handler is generated when an unhandled Winsock error occurs within the control or data connection. ErrorCode contains the error code returned by Winsock. See “Error Handling and Exception Classes” on page 900 for a list of error codes.

Password**property**

```
property Password : string
```

↳ Specifies the user’s login password.

FTP requires users to log in with a user name and password to gain access to that computer. Set Password prior to calling Login when connecting to an FTP server.

Users who do not have a personal login account can gain access an FTP site with an anonymous account. To log in with the anonymous account, set UserName to ANONYMOUS and the password is your e-mail address.

See also: Account, Login, UserName

Rename**method**

```
function Rename(  
    const RemotePathName, NewPathName : string) : Boolean;
```

↳ Renames a remote file or directory at an FTP server.

RemotePathName specifies the file or directory at the server. If RemotePathName or NewPathName is an empty string or a rename operation is not allowed given the current protocol state, Rename returns False, otherwise True is returned and the operation is initiated.

If the file or directory is successfully renamed at the server, the OnFtpStatus event is fired with the scComplete status code. If the rename operation is rejected by the server, the OnFtpError event is fired and the operation is terminated.

See also: OnFtpError, OnFtpStatus

```
property RestartAt : Longint
```

↳ Specifies where to resume an interrupted file transfer.

If the FTP server supports resumable file transfer, it can be restarted at somewhere other than the beginning of the file by calling Retrieve with `rmRestart`, or Store with `smRestart`. The value contained by `RestartAt` is used to determine the byte location in the file to resume the transfer.

When restarting a Retrieve operation, if `RestartAt` is zero, then the transfer will resume at the end of the local file which is the point where the original transfer was interrupted. Otherwise, the transfer will resume at the location specified by `RestartAt` and subsequent data in the local file will be overwritten. If `RestartAt` is greater than the size of the local file, then no transfer will take place and the call to Retrieve will return `False`.

When restarting a Store operation, the transfer will resume at the location specified by `RestartAt` and subsequent data in the remote file will be overwritten. If `RestartAt` is zero the entire file will be transferred. If `RestartAt` is greater than the size of the local file, then no transfer will take place and the call to Store will return `False`.

See also: Retrieve, Store

Retrieve

method

```
function Retrieve(const RemotePathName, LocalPathName : string;
  RetrieveMode : TFTPRetrieveMode) : Boolean;

TFTPRetrieveMode = (rmAppend, rmReplace, rmRestart);
```

↳ Retrieve transfers a file from the FTP server to the local machine.

`RemotePathName` specifies the file at the server, and `LocalPathName` specifies the pathname of the file on the local machine. `RetrieveMode` specifies how data will be written to an existing local file.

The file will be transferred according to the file type specified by the `FileType` property.

If the local file already exists: `rmAppend` specifies that the incoming file data will be appended to the end of the file; `rmReplace` specifies that the contents of the local file will be replaced; `rmRestart` specifies that either the contents of the local file will be replaced starting at the location specified by the `RestartAt` property, or if `RestartAt = 0`, the incoming data will be appended to the end of the file. See `RestartAt` for more information about restarting a file transfer.

If the local file does not exist, it will be created.

If either RemotePathName or LocalPathName contain an empty string, or a retrieve operation is not allowed given the current protocol state, Retrieve returns False, otherwise the file transfer is initiated, the InProgress property is set to True, and the function returns True. Periodically during the transfer, the OnFtpStatus event is fired with the scProgress status code. The BytesTransferred property contains the number of bytes written to LocalFile so far.

If the file is successfully transferred, the InProgress property is set to False and the OnFtpStatus event is fired with the scTransferOk status code.

If the transmission times out, then the OnFtpStatus event is fired with the scTimeout status code. If the server rejects the transfer command, the OnFtpError event is fired and the operation is terminated.

See also: BytesTransferred, FileType, InProgress, OnFtpError, OnFtpStatus, RestartAt, TransferTimeout

SendFtpCommand

method

```
function SendFtpCommand(const FtpCmd : string) : Boolean;
```

↳ Sends a FTP protocol command to the server.

FtpCmd is an FTP command string as specified in RFC 959. The FTP commands that can be issued via this method are restricted to those not requiring a data connection. Thus all file transfer commands (e.g., STOR, RETR, etc.) and the LIST and NLST commands are prohibited. To illustrate, here are a few accepted commands:

```
SendFtpCommand('CWD pub/apro');  
SendFtpCommand('STAT pub');  
SendFtpCommand('HELP RETR');
```

The function returns True immediately if the command is initiated, otherwise False is returned. Upon successful completion, the OnFtpStatus event is fired with the scComplete status code.

If the server rejects the command for some reason, then the OnFtpError event is fired and the operation is terminated.

See also: OnFtpError, OnFtpStatus


```
property ServerAddress : string
```

↳ Specifies the FTP server's IP address or host name.

ServerAddress accepts the IP address in dot notation (e.g., 165.212.210.10) or as a host name (e.g., ftp.turbopower.com). If a host name is used, a DNS lookup is performed to determine whether a DNS entry exists for the host name. If an IP address can be found, the port is opened to establish an FTP control connection. If an IP address cannot be found, an `EApdSocketException` is raised.

```
function Status(const RemotePathName : string) : Boolean;
```

↳ Obtains status information from the FTP server.

`RemotePathName` specifies a file or directory at the server. If `RemotePathName` is an empty string, general server status information is requested. If `RemotePathName` specifies a directory at the server, a full listing of the directory contents is requested. If `RemotePathName` specifies a file at the server, then the file size and timestamp are requested.

If a status operation is not allowed given the current protocol state, `Status` returns `False`. Otherwise, `True` is returned and the operation is initiated. When the server responds with the requested status information, the `OnFtpStatus` event is fired with the `csDataAvail` command status code, and the `InfoText` parameter will point to a null terminated string containing the raw text of the status information.

If the operation is rejected by the server, the `OnFtpError` event is fired and the operation is terminated.

See also: `OnFtpError`, `OnFtpStatus`

```
function Store(const RemotePathName, LocalPathName : string;  
    StoreMode : TFTPStoreMode) : Boolean;  
  
TFTPStoreMode = (smAppend, smReplace, smUnique, smRestart);
```

↪ Transfers a file from the local machine to the FTP server.

RemotePathName specifies the file at the server. LocalPathName specifies the file on the local machine. StoreMode identifies how the file will be written to an existing remote file.

The file will be transferred according to the file type specified by the FileType property. Be sure to set FileType prior to initiating a file transfer. For text files use ftAscii, otherwise use ftBinary.

If the remote file specified by RemotePathName already exists in the server's working directory, StoreMode controls the effect of the transfer according to the following values:

Value	Effect
smAppend	The local file data will be appended at the end of the remote file.
smReplace	The contents of the remote file will be replaced.
smUnique	A remote file will be created with a unique name and the local file will be written to it.
smRestart	Either the contents of the remote file will be replaced starting at the location specified by the RestartAt property, or, if RestartAt equals 0, the file data will be appended to the end of the remote file. See RestartAt for more information about restarting a file transfer.

If the remote file does not exist, it will be created in the server's current working directory. See ChangeDir for information about changing the server's working directory.

If either `RemotePathName` or `LocalPathName` contain an empty string, or a store operation is not allowed given the current protocol state, `Store` returns `False`, otherwise the file transfer is initiated, the `InProgress` property is set to `True`, and the `Store` returns `True`. Periodically during the transfer, the `OnFtpStatus` event is fired with the `scProgress` status code. The `BytesTransferred` property contains the number of bytes accepted so far by Winsock. When the file has been successfully transferred, the `InProgress` property is set to `False` and the `OnFtpStatus` event is fired with the `scTransferOk` status code.

If the transmission times out, then the `OnFtpStatus` event is fired with the `scTimeout` status code. If the server rejects the transfer command, the `OnFtpError` event is fired and the operation is terminated.

See also: `BytesTransferred`, `FileType`, `InProgress`, `OnFtpError`, `OnFtpStatus`, `RestartAt`, `TransferTimeout`

TransferTimeout property

property `TransferTimeout` : `Integer`

Default: 1092

↪ Determines the maximum time (ticks) to wait during file transfer.

During a file transfer operation, each time a block of data is written out to, or read in from the FTP data connection, a timer is started with the time-out value specified by `TransferTimeout`. If the timer times out before the next block of data is received or accepted by Winsock, then the transfer operation is terminated, and the `OnFtpStatus` event is fired with the `scTimeout` status code.

See also: `OnFtpStatus`, `Retrieve`, `Store`

UserLoggedIn read-only, run-time property

property `UserLoggedIn` : `Boolean`

↪ Indicates whether or not an FTP session is active.

This property can be checked periodically to determine if the user is logged in to an FTP server and an FTP session currently underway.

See also: `Login`, `Logout`

```
property UserName : string
```

↪ Specifies the user's login name.

FTP requires users to log in with a user name and password to gain access to the server. Be sure to set Password prior to calling Login when connecting to an FTP server.

Users who do not have a personal login account can gain access an FTP site with an anonymous account. To log in with the anonymous account, set UserName to ANONYMOUS and the password is to the user's e-mail address.

See also: Password, UserLogin

TapdAbstractStatus Class

TapdAbstractStatus is an abstract class that defines the methods and properties needed by a component that automatically displays status while a TapdProtocol component is in the process of transferring a file. You generally won't need to create a descendent class of your own, since Async Professional supplies one, the TapdProtocolStatus component described on page 582.

However, TapdProtocolStatus shows a particular set of information about a transfer in a predefined format, and you may find that this format is not suitable for your needs. If that is that case, you need to create your own descendant of TapdAbstractStatus. Probably the best way to do so is to study the source code of TapdProtocolStatus (in the AdPStat unit) and its associated form, TStandardDisplay.

The TapdAbstractStatus class contains an instance of a TForm that holds various controls used to display the protocol status. You design this form, create an instance, and assign the instance to the Display property of TapdAbstractStatus.

TapdAbstractStatus overrides the standard VCL properties Ctl3D, Position, and Visible and the standard VCL method Show. When these routines are used in the status component, the overridden versions perform the same actions on the associated Display form. Thus you can display the status form by calling Show, erase it by setting Visible to False, adjust its position by assigning to Position, and use 3D effects by setting Ctl3D to True.

Once you have created an instance of your TapdAbstractStatus descendant, you must assign it to the StatusDisplay property of your TProtocol component. When the protocol needs to update the status display it calls the UpdateDisplay method of TapdAbstractStatus, which you must override in order to update your particular kind of status window.

The source code for the TapdProtocolStatus component (in the AdPStat unit) serves as a comprehensive example of writing a TapdAbstractStatus descendant.

Hierarchy

TComponent (VCL)

 ❶ TApdBaseComponent (OOMisc) 8

 TApdAbstractStatus (AdProtcl)

Properties

Display

Protocol

❶ Version

Methods

CreateDisplay

DestroyDisplay

UpdateDisplay

Reference Section

CreateDisplay

virtual abstract method

```
procedure CreateDisplay; virtual; abstract;
```

↳ An abstract method that creates a form to display protocol status.

A descendant of `TApdAbstractStatus` must override this method with a routine that creates a `TForm` component that contains various controls (typically of type `TLabel`) for displaying the protocol status. The `TForm` should usually also contain a `TButton` control and associated `CancelClick` event handler that allows the user to cancel the protocol.

`CreateDisplay` must then assign the instance of this form to the `Display` property.

See also: `DestroyDisplay`, `Display`

DestroyDisplay

virtual abstract method

```
procedure DestroyDisplay; virtual; abstract;
```

↳ An abstract method that destroys the display form.

A descendant of `TApdAbstractStatus` must override this method to destroy the `TForm` instance created by `CreateDisplay`.

Display

run-time property

```
property Display : TForm
```

↳ A reference to the form created by `CreateDisplay`.

`CreateDisplay` must assign a properly initialized instance of a `TForm` to this property. `UpdateDisplay` can refer to this property to update the status window.

Protocol

property

```
property Protocol : TApdCustomProtocol
```

↳ The protocol component that is using the status component.

When deriving your own components from `TApdAbstractStatus` you will probably want to reference `TApdProtocol` properties to display information about the progress of the protocol. Use this property to do so. It is automatically initialized when you assign the status component to the `StatusDisplay` property of `TApdProtocol`.

```
procedure UpdateDisplay(First, Last : Boolean); virtual; abstract;
```

↳ An abstract method that writes the contents of the status window.

A descendant of `TApdAbstractStatus` must override this method to update the display form. The `TApdProtocol` component calls this method regularly from its `OnProtocolStatus` event handler.

On the very first call to `UpdateDisplay`, `First` equals `True` and `UpdateDisplay` should typically call the `Show` method of `Display` to draw the outline and background of the status form. On the very last call to `UpdateDisplay`, `First` equals `False` and `UpdateDisplay` should typically set the `Visible` property of `Display` to `False` to erase the status window.

For all other calls to `UpdateDisplay`, `First` and `Last` both equal `False`. During these calls, `UpdateDisplay` must update the various labels in the `Display` form. To get information about the protocol status, it should use the `Protocol` field of `TApdAbstractStatus` to read the values of various properties such as `FileName` and `BytesTransferred`. See “Protocol status” on page 489 for a list of the most commonly used properties.

The `CancelClick` event handler, if one is provided, should call the `CancelProtocol` method of `TApdProtocol` to terminate the protocol because the user clicked the Cancel button.

TApdProtocolStatus Component

TApdProtocolStatus is a descendant of TApdAbstractStatus that implements a standard protocol status display. All you need to do is create an instance of a TApdProtocolStatus component and assign it to the StatusDisplay property of your TApdProtocol component. TApdProtocolStatus includes all of the most-often used information about a protocol transfer and it also provides a Cancel button so that the user can stop the protocol at any time.

TApdProtocolStatus overrides all the abstract methods of TApdAbstractStatus. TApdProtocolStatus has no methods that you must call or properties that you must adjust. You might want to change the settings of the Ctl3D and Position properties to modify the appearance of the window. Figure 14.6 shows the TStandardDisplay form that is associated with a TApdProtocolStatus component.

Protocol Status	
Protocol:	Zmodem
Block check:	Crc32
File name:	WARPEACE.TXT
File size:	3265062
Block size:	1024
Total Blocks:	3188
Bytes transferred:	502021
Bytes remaining:	2763041
Blocks transferred:	490
Blocks remaining:	2699
Block errors:	0
Total errors:	1
Estimated time:	28:55
Elapsed time:	2:14
Remaining time:	24:28
Throughput:	3738 CPS
Efficiency:	195%
Kermit windows:	0
Status:	OK
Progress:	<div><div></div></div>
<div>Cancel</div>	

Figure 14.6: The TApdProtocolStatus component's TStandardDisplay form.

For an example of using a TApdProtocolStatus component, see the introduction to TApdProtocol on page 523.

Hierarchy

TComponent (VCL)

 TApdBaseComponent (OOMisc) 8

 TApdAbstractStatus (AdProtcl)..... 578

 TApdProtocolStatus (AdPStat)

TApdProtocolLog Component

TApdProtocolLog is a small class that can be associated with a TApdProtocol component to provide automatic protocol logging services. Just create an instance of TApdProtocolLog and assign it to the ProtocolLog property of the TApdProtocol component.

TApdProtocolLog creates or appends to a text file whose name is given by the HistoryName property. Each time the OnProtocolLog event of TApdProtocol is generated, the associated TApdProtocolLog instance opens the file, writes a new line to it, and closes the file.

TApdProtocolLog also deletes the partial file that exists whenever a receive fails and the protocol type is not Zmodem (which can resume interrupted transfers).

Following is a sample of the text file created by TApdProtocolLog:

```
Zmodem transmit started on 7/6/01 8:33:21 AM : C:\TEMP\PROJ1.EXE
Zmodem transmit finished OK 7/6/01 8:33:28 AM : C:\TEMP\PROJ1.EXE
Elapsed time: 0:07 CPS: 1792 Size: 12547

Zmodem transmit started on 7/6/01 8:33:28 AM : C:\TEMP\PROJ2.EXE
Zmodem transmit finished OK 7/6/01 8:33:37 AM : C:\TEMP\PROJ2.EXE
Elapsed time: 0:08 CPS: 1971 Size: 15775

Zmodem transmit started on 7/6/01 8:33:37 AM : C:\TEMP\PROJ2.EXE
Zmodem transmit failed C:\TEMP\PROJ2.EXE Cancel requested

Zmodem receive started on 7/6/01 8:34:03 AM : ZIPVO.PAS
Zmodem receive failed ZIPVO.PAS Cancel requested
```

Hierarchy

TComponent (VCL)

- TApdBaseComponent (OOMisc) 8
 - TApdProtocolLog (AdProtcl)

Properties

- DeleteFailed
- Protocol
- HistoryName
- Version

Methods

- UpdateLog

Reference Section

DeleteFailed

property

```
property DeleteFailed : TDeleteFailed
TDeleteFailed = (dfNever, dfAlways, dfNonRecoverable);
Default: dfNonRecoverable
```

↪ Determines whether received files are deleted after a protocol failure.

When a protocol receive session fails, there might be a partially received file in the destination directory (depending on when and why the session failed). DeleteFailed controls whether a partial file is automatically deleted by the TApdProtocol OnProtocolLog event handler. DeleteFailed can have one of the following values:

Value	Description
dfNever	Partial files are never deleted.
dfAlways	Partial files are always deleted.
dfNonRecoverable	Partial files are deleted if the protocol cannot resume a failed transfer (all protocols except Zmodem).

Regardless of the value of DeleteFailed, received files are never deleted when the protocol error is ecCantWriteFile, since that error usually indicates that the receiver doesn't want to disturb an existing file with the same name.

HistoryName

property

```
property HistoryName : string
Default: "APRO.HIS"
```

↪ Determines the name of the file used to store the protocol log.

You should generally set the value of HistoryName before calling TApdProtocol's StartReceive or StartTransmit methods. However, because the log file is opened and closed for each update, you can change HistoryName at any time you wish. If you set HistoryName to an empty string, automatic logging is disabled until you assign a non-empty string.

Protocol

property

```
property Protocol : TApdCustomProtocol
```

↳ The protocol component that is using the log component.

Protocol is automatically initialized when the ProtocolLog property of the owning protocol component is set. You can change Protocol to assign the log component to a different protocol component.

UpdateLog

virtual method

```
procedure UpdateLog(const Log : Word); virtual;
```

↳ Call for each protocol logging event.

The Log parameter has the same values passed to the OnProtocolLog event handler of TApdProtocol. UpdateLog creates or appends to the log file, builds and writes a text string for each event, and closes the log file. Additionally, it deletes the partially received file if Log equals lfReceiveFail and the protocol type is not Zmodem.

Note that TApdProtocolLog contains a field named Protocol that UpdateLog uses to obtain additional information about the protocol such as the FileName, FileLength, ElapsedTicks, and ProtocolType.

See also: TApdProtocol.OnProtocolLog

Chapter 15: Fax Components

Document transfer using facsimile (fax) machines has become quite common—you might even say pervasive—in today's business environment. Almost all of the currently manufactured modems are faxmodems. A faxmodem is a standard data modem that also has the ability, when used with appropriate software, to send and receive faxes. Since the faxmodem is under program control, it can provide more sophisticated capabilities than a dedicated fax machine. A few of the possibilities include database storage, editing, and forwarding of received documents, as well as scheduled fax transmissions to multiple recipients.

Unfortunately, controlling a faxmodem is a relatively complex task. As is typical for the communications industry, faxmodem technology is governed by multiple, evolving, incomplete standards. With the exception of the terse technical specifications offered by the TIA/EIA committee that controls faxmodem standards, little has been written about controlling faxmodems. The TIA/EIA specifications describe the bare necessities of faxmodem behavior; many ambiguities must be resolved by research and experimentation.

Microsoft has provided various levels of fax services in successive versions of Windows and is trying to broaden and homogenize its fax services over all of its Windows operating systems and environments. Parts of the fax services are available to programmers, but the documentation is both scarce and sparse, causing many programmers to look elsewhere for fax services.

Async Professional overcomes this information shortage by providing a complete set of routines for computer control of faxmodems. These routines cover all phases of faxmodem usage including document conversion, fax printer drivers, and send/receive support for the current generation of faxmodems. These routines have a structure similar to the file transfer protocols, complete with the programming hooks that allow you to write full-featured applications.

Faxmodem Control from an Application

Integrating faxmodem support into your application involves two central tasks:

- Document conversion
- Faxmodem send/receive

Document conversion means converting a file into a format suitable for fax transmission or converting a received fax into a format suitable for further processing (viewing, printing, etc.). Faxmodem send/receive covers all the steps needed to control a faxmodem when sending and receiving fax documents.

Document conversion

Document conversion is the process of creating a compressed bitmap image suitable for fax transmission. Async Professional can convert the following file formats:

- ASCII text files
- Windows bitmap image files (BMP)
- PC Paintbrush image files (PCX)
- Multi-page PCX files (DCX)
- Tagged Image File Format image files (TIFF)

Files that have been converted to fax format in this way, as well as files that have been received by Async Professional's fax routines, are given the extension APF (Async Professional Fax).

Async Professional also includes a printer driver for Windows 95/98/ME, and Windows NT 4.0/2000 (see the README.TXT file for an up-to-date list of the supported environments) that provides convenient conversion of virtually any document file by “printing” that file to the fax printer driver. The fax printer driver converts the printed image to an APF file and can optionally alert a fax transmit program that a file is now available for fax transmit.

Received faxes are stored in the compressed bitmap image (APF format), so the data must be unpacked before you can view, print, edit, or otherwise process the fax. Async Professional provides an unpacker component that can unpack fax file to image files, to memory bitmaps, or for special processing by your application. A viewer component and a printer component make it easy to view or print the fax.

Faxmodem send/receive

Faxmodem send/receive is the process of sending the appropriate commands to a faxmodem to prepare it for sending or receiving faxes, initiating or receiving a call, transferring a bitmap image, and terminating the connection.

Like a file transfer protocol, a successful fax transmission requires cooperation between the sender and receiver. A number of standards have been developed for this purpose. Early fax machines communicated using what was known as the Group 1 and Group 2 protocols. Although your fax machine may have a Group 1 button, which allows it to receive faxes (very slowly) from an old Group 1 fax machine, all fax machines sold today support the Group 3 facsimile protocol. The faxmodems that Async Professional can control do not support Group 1 or Group 2 at all. If you need to send or receive faxes with a Group 1 partner, keep your old fax machine!

Within Group 3, there are currently two EIA/TIA standards for computer control of faxmodems: Class I and Class II. To identify these classes, Async Professional uses Arabic numerals (Class 1 and Class 2) rather than Roman numerals because the numbers are clearer to read in the source code and documentation. Class 2 depends on somewhat more sophisticated chips within the faxmodem than Class 1. Class 2 chips are capable of negotiating certain fax transmission parameters without any feedback from the computer.

Many modem manufacturers began producing Class 2 faxmodems before the specification was complete, using an interim version of the specification. When the specification was finally ratified, it differed significantly from the interim Class 2 specification. In the meantime, interim Class 2 modems had become a defacto standard and could not be ignored by the specification committee. To distinguish between interim Class 2, which is referred to as simply Class 2, the final Class 2 specification is formally identified as Class 2.0. Over the past few years, the Class 1 standard has been improved upon to a slight degree also. Recently, the EIA/TIA committee began the formalization process for the Class 1.0 standard. While this new standard introduces a few new optional features, it is primarily a simple renaming of the old Class 1 standard.

Faxmodem specifications

Wherever possible the faxmodem components insulate you from the details of document conversion and faxmodem control. Just as with the file transfer protocols, you don't need to read and understand all of the faxmodem technical specifications to use the faxmodem routines. However, it does help to have a basic understanding of the specifications. For further information, see the technical specifications listed in Table 15.1.

Table 15.1: *Technical specification documents*

Document Number	Description
RS-465	Group 3 Facsimile Apparatus for Document Transmission.
RS-466	Procedures for Document Facsimile Transmission.
EIA/TIA-578	Asynchronous Facsimile DCE Control Standard.
EIA/TIA-592	Asynchronous Facsimile DCE Control Standard—Service Class II.

These documents are available from the EIA and TIA organizations directly. They can also be obtained from Global Engineering Documents, a company that distributes engineering specifications of all kinds. You can reach them by telephone at 800-854-7179 or 303-792-2181. Their fax number is 303-397-2740.

Document Conversion

Faxmodems don't transmit documents directly. Instead, they transmit a compressed bitmap image in a format that is specific to Group 3 fax devices. The TApdFaxConverter component provides methods for converting standard image file formats into this format. Async Professional uses a proprietary image file format (APF) that stores bitmap data in this Group 3 format. The TApdFaxUnpacker component provides methods for unpacking APF files into standard image file formats.

Fax file format

The Group 3 compressed bitmap format was designed specifically for transmission of data over possibly noisy lines. The compression technique results in a file that combines relatively small size with ease of recovery from missing or garbled data. As the receiving fax machine or faxmodem software detects bad compression strings, it simply discards them and starts collecting valid strings again. No transmission time is spent calculating and sending block check characters. A document received with line errors is usually missing just a few pixels; if the received document looks bad enough, the sender must transmit again.

Although the format of each compressed raster line is defined by the Group 3 facsimile specification, the APF file format used by Async Professional is proprietary. The data within each raster line follows the Group 3 specification, but the file contains additional information that makes it easier to manage and transmit the image.

An APF file is formatted as shown in Table 15.2.

Table 15.2: *APF file format*

fax header
header for page 1
data for page 1
.
.
.
header for page N
data for page N

You usually don't need to understand this format in any detail, but the information is documented here in case you need to write APF manipulation routines that aren't provided with Async Professional.

The file always begins with a header that contains, among other information, the number of pages in the fax. Each page of the document follows, including a page header and the compressed page data. The fax header and page header structures are defined in detail later in this section.

The page data is a series of compressed raster line images. The line image optionally begins with a word containing the number of bytes in the compressed line. (This word is stored only in APF files that are ready for transmission; it is used to aid in padding each line to match timing parameters of the receiving fax machine.) The length word is followed by the line image in Group 3 compression format. If a pixel is set, it corresponds to a black dot on the original image; if it is clear, it corresponds to a white dot.

Fax images can be converted and stored using two different resolutions. Standard resolution is 200 horizontal dots per inch by 100 vertical dots per inch. High resolution has the same horizontal resolution, but uses 200 vertical dots per inch. In some fax documentation, the resolutions are described as 98 dots per inch and 196 dots per inch. Those numbers are actually more exact, but 100 and 200 are easier to remember and are commonly used in most fax documentation.

The standard width of a fax page is 1728 pixels, or about 8.5 inches. Several optional widths are also available. Async Professional supports only one of the optional widths: 2048 pixels per row, or about 10 inches. There is no fundamental limit on fax page length. Even so, you'll probably want to limit it to 11 inches, or 14 inches if you want to mimic legal size paper. This is especially important when you consider that many faxes are now printed on sheet-fed laser printers.

You specify the fax resolution and horizontal width when a document is converted to an APF file. The resolution and width of each page are stored in the page header. When the APF file is later transmitted, the TAPdSendFax component reads the page header to determine the resolution and width to use to transmit the fax.

APF file header

Table 15.3 shows the fields in the APF file header structure, `TFaxHeaderRec`.

Table 15.3: *APF file header fields*

Field	Purpose
Signature	A unique string that identifies the file as an APF file.
FDateTime	Date and time that the file was created (in DOS format).
SenderID	The station ID of the fax device that transmitted the fax data.

Table 15.3: *APS file header fields (continued)*

Field	Purpose
Filler	A dummy byte used to align the rest of the header on a word boundary.
PageCount	The number of pages in the APF file.
PageOfs	The offset, in bytes, of the first page in the file.
Padding	26 bytes of extra data, leaving room for future expansion and forcing the size of the fax header to 64 bytes.

APF page header

Table 15.4 shows the fields in the APF page header structure, TPageHeaderRec.

Table 15.4: *APF page header fields*

Field	Purpose
ImgLength	The length of the compressed Group 3 data on the page.
ImgFlags	Flags describing the content of the page.
Padding	10 bytes of extra data, leaving room for future expansion and forcing the size of the fax header to 16 bytes.

TGraphic registration of the APF format

The APF format has been registered as a TGraphic descendent. Components that make use of TGraphic, such as TPicture, will be able to load and save the APF format.

The TGraphic descendent of the APF format can be used to convert other graphics formats, like JPG, ICO, EMF and WMF to and from the APRO fax format. If you have additional third party TGraphic descendents installed in your Delphi or C++ Builder environment, these additional formats can be converted to and from the APRO APF format.

TApdFaxConverter Component

Converting a document to APF format is the first step in the fax transmission process. You can convert your documents just before you transmit them or you can convert them in advance. If you like, this conversion process can be completely transparent to your users or you can convert your documents in a separate step that is not immediately followed by transmission. Async Professional also provides a Windows printer driver that can create APF files.

The TApdFaxConverter component can be used to convert ASCII text, BMP, PCX, DCX, and TIFF files to Async Professional's proprietary file format. For input images not directly supported by Async Professional, events and methods are published by the component that allow the conversion of user-defined input images. Additionally, TApdFaxConverter can be used for generically reading input image file for user-defined tasks.

ASCII text documents

The TApdFaxConverter component converts ASCII text files into APF files when the `InputDocumentType` property of the converter is set to `idText` or `idTextEx`. Each text line of the input file must end with a carriage return and a line feed. The converter can handle all 256 characters in the OEM character set. The converter cannot convert files that contain embedded word processor formatting commands.

The text converter reads each line of the text file and converts the line into an appropriate number of bitmapped raster lines. In essence, it converts each line into a picture of itself. To do this, it uses a font table that contains a bitmap of each ASCII character. The number of raster lines per text line depends on the pixel height of each character and the vertical resolution of the fax conversion. The number of pixels in each raster line depends on the pixel width of each character. For each raster line and each character in the line, the converter finds the character bitmap in the font table and extracts the appropriate horizontal pixel row. After looping through all the rows in the font, the converter has created a bitmap image of the text line.

Async Professional can use any of the fonts available to Windows (such as the TrueType fonts) when `InputDocumentType` is set to `idTextEx`, or it uses a set of built-in bitmapped fonts when the `InputDocumentType` property is set to `idText`. The `EnhFont` property controls which font is used when `InputDocumentType` is set to `idTextEx`.

There are two built-in fonts available when `InputDocumentType` is set to `idText` – a standard font (`ffStandard`) and a smaller font (`ffSmall`). `ffSmall` is a small 12x8 (12 pixels wide by 8 pixels high) font used for creating header lines at the top of each transmitted fax page. `ffStandard` is a 20x16 font used for all other text. `ffStandard` was chosen to provide text lines at 6 lines per inch (66 lines on a standard 11 inch page). If you specify a high resolution image, the fonts are scaled vertically to 16 and 32 pixels, respectively.

The built-in fonts are stored in `APFAX.FNT`, which is 16KB. This font file can be distributed with your applications. Alternatively, you can bind `APFAX.FNT` directly into your program by activating the compiler define `BindFaxFont` in `AWCVTFAX.PAS` (it's defined by default). When `BindFaxFont` is defined, `APFAX.RES` (created from `APFAX.FNT` using Borland's Resource Workshop and a user-defined resource type) is linked into your program at compile time. This adds about 16KB to your EXE file.

BMP, PCX, DCX, and TIFF graphic images

Async Professional provides document conversion routines for four popular graphics image formats: BMP, PCX, DCX, and TIFF. The PCX format originated with the PC Paintbrush program. The Async Professional conversion routines are tested with PCX images up through version 3.0. DCX files are special container files that contain one or more PCX images. TIFF (Tagged Image File Format) is a multi-platform format that is designed to allow easy migration between platforms such as the Macintosh and the IBM PC. The Async Professional conversion routines are tested with TIFF images up through version 4.0. BMP files are standard Windows bitmap files.

Most Async Professional conversion routines work with monochrome images only. The exception to this rule is the bitmap converter, which has the ability to dither color images. Note that the dithering process is slower, so you should keep your images monochrome if possible. Converting a BMP, PCX, or TIFF image file always produces an APF file containing one fax page. Converting a DCX file produces a fax containing as many pages as are contained in the DCX file.

Because BMP, PCX, DCX, and TIFF images are already a sequence of compressed raster lines, the job of the `TApdFaxConverter` component is different than that done in the text conversion process. For these files, the conversion routines unpack the images, then repack them into the format required for faxing.

The images supported by Async Professional are stored assuming a pixel aspect ratio of 1 to 1, which means that the height of a pixel is the same as its width. For example, a 10 by 10 box of pixels would appear on screen as a square, not a rectangle. Fax images, on the other hand, have one of two aspect ratios. In high resolution, the aspect ratio is 1 to 1. In standard resolution, the aspect ratio is 2 to 1. Hence, in a standard resolution fax, a 10 by 10 box of pixels would not appear as a square. Instead it would be a rectangle twice as high as it is wide.

Given the differences in aspect ratios, an image converted to a standard resolution APF image appears distorted—tall and thin. To solve this problem, the `TApdFaxConverter` component can either double the width of an image or halve the height of an image when it is converted to a standard resolution fax and the modified image still fits on the page.

These behaviors are enabled by turning on the `coDoubleWidth` and `coHalfHeight` options, respectively. The `coDoubleWidth` option is on by default. `coDoubleWidth` was chosen over `coHalfHeight`, since doubling the width of the image does not discard data, whereas halving the height of the image causes every other line to be discarded.

The `TApdFaxConverter` also includes another option, `coCenterImage`, that is used during image file conversions. When this option is enabled, as it is by default, graphic images are automatically centered horizontally on the fax page. See the `Options` property on page 619 for more information about these options.

Processing image files

There are times when you might find it useful to be able to process an image file (i.e., any file that can be represented in the form of a bitmap) within your program. For instance, you might want to implement a cover page editor that allows the user to open image files and put them on the cover page, without having to convert the image into an APF file as an interim step.

The `TApdFaxConverter` allows you to manually process image files, doing whatever you wish with the raster data read from the file. This is done using the `OpenFile`, `GetRasterLine`, and `CloseFile` methods. The first step in manually processing an image file is to set the `DocumentFile` and `InputDocumentType` properties of the converter.

The next step is to call the component's `OpenFile` method. This opens the image file, reads any header data for the image, validates the header data, and returns control to you. One of several exceptions can be raised depending on whether or not the input file exists and whether or not the input file is a valid image file of the type specified by the `InputDocumentType` property.

After opening the image file, make one or more calls to the `GetRasterLine` method of the converter. `GetRasterLine` takes four parameters: `Buffer`, `BufLen`, `EndOfPage`, and `MorePages`.

`Buffer` is the buffer that receives the raster data you are reading. You should make this buffer at least 512 bytes long. The actual length, in bytes, of the raster line is in `BufLen` upon return. If the `EndOfPage` parameter is set to `True` on return, the end of the current page has been reached. If `MorePages` is `True`, there are more pages in the file to be processed.

Lastly, you must call `CloseFile` to close the image file when you are done processing it. Calling `CloseFile` closes the physical image file and disposes of several internal data structures that are used to read the image file.

The following pseudo-code shows a typical use of the manual image processing methods of the converter:

```
procedure ProcessImageFile(
  FName : string; DocType : TFaxInputDocumentType);
var
  Cvt : TApdFaxConverter;
  Buffer : PByteArray; {type defined in SysUtils}
  BufLen : Integer;
  EndOfPage : Boolean;
  MorePages : Boolean;
begin
  GetMem(Buffer, 512);

  try
    Cvt := TApdFaxConverter.Create(nil);
  except
    FreeMem(Buffer, 512);
    raise;
  end;

  Cvt.InputDocumentType := DocType;
  Cvt.DocumentFile := FName;

  try
    Cvt.OpenFile;
  except
    Cvt.Free;
    FreeMem(Buffer, 512);
    raise;
  end;
```



```

MorePages := True;
try
  while MorePages do begin
    ...code for handling beginning of new page...

    EndOfPage := False;
    while not EndOfPage do begin
      Cvt.GetRasterLine(Buffer^, BufLen, EndOfPage, MorePages);
      ...code to process data in Buffer...
    end;

    ...code for handling end of page...
  end;
finally
  Cvt.CloseFile;
  Cvt.Free;
  FreeMem(Buffer, 512);
end;
end;

```

For a more complete demonstration of these features, see the EXIMAGE example program.

Converting user-defined image files

The TApdFaxConverter component allows you to convert image types that are not directly supported. This is done through the OnOpenUserFile, OnCloseUserFile, and OnReadUserLine events. When InputDocumentType equals idUser, the TApdFaxConverter calls these events to open, close, and read the image file, instead of using its own internal routines. These events allow you to create your own fax converter, if you know the format of the image you need to convert.

OnOpenUserFile is called to open the user-defined image file. When this event is called, you should open the physical image file, read its headers (if any), and allocate any buffers needed for reading and processing the data.

OnReadUserLine is called to read a single line of raster data from the user-defined image file. The two Boolean parameters passed to the event tell the converter when the end of an input page is reached and whether there are any more pages to convert.

OnCloseUserFile is called to close the user-defined image file. When this event is called, you should close the physical input file and deallocate any buffers that were allocated by the OnOpenUserFile event.

The following example assumes a hypothetical image file type. This image file has a 4-byte header. The first two bytes of the header are the width (in pixels) of the image, the next two bytes contain the height (in pixels) of the image. The rest of the image file is raw,

uncompressed, raster data. Each raster line in the file is just long enough to hold each line (i.e., no padding). The OnOpenUserFile, OnReadUserLine, and OnCloseUserFile events open, read, and close image files of this hypothetical image type.

```

type
  TImageHeader = packed record
    Width : Word;
    Height : Word;
  end;

var
  Header : TImageHeader; InputFile : File; ReadLen : Integer;
  BytesInFile : LongInt; BytesProcessed : LongInt;
procedure Form1.ApdFaxConverter1OpenUserFile(
  F : TObject; FName : string);
begin
  {open the physical file}
  AssignFile(InputFile, FName);
  Reset(InputFile, 1);

  {read the header}
  BlockRead(InputFile, Header, SizeOf(TImageHeader));

  {calculate the length, in bytes, of each raster line}
  ReadLen := (Header.Width + 7) shl 3;

  {calculate the number of bytes in the file, for status info}
  BytesInFile := FileSize(InputFile) - SizeOf(TImageHeader);
  BytesProcessed := 0;
end;

procedure Form1.ApdFaxConverter1ReadUserLine(
  F : TObject; Data : PByteArray; var Len : Integer;
  var EndOfPage, MorePages : Boolean; var BytesRead,
  BytesToRead : LongInt);
begin
  {if we're at the end of the file, we're done}
  EndOfPage := Eof(InputFile);
  MorePages := False;
  if EndOfPage then
    Exit;

  {read the next block of raster data}
  BlockRead(InputFile, Data, ReadLen);
  Len := ReadLen;

```

```

        {update status information}
        Inc(BytesProcessed, ReadLen);
        BytesRead := BytesProcessed;
        BytesToRead := BytesInFile;
    end;

    procedure Form1.ApdxFaxConverter1CloseUserFile(F : TObject);
    begin
        {close image file}
        CloseFile(InputFile);
    end;

```

Example

This simple example demonstrates the steps involved in creating an APF file from an ASCII text file. Create a new project, add the following components, and set the property values as indicated in the following table:

Table 15.5: *Example components and property values*

Component	Property	Value
TApdxFaxConverter	DocumentFile	C:\MYFILE.TXT (or some other existing text file)
TApdxFaxConverter	InputDocumentType	idText
TLabel		
TButton		

Double click on the TButton component. A shell for an OnClick event is generated for you. Modify the generated code to match the following code:

```

procedure TForm1.Button1Click(Sender : TObject);
var
    SaveCursor : TCursor;
begin
    SaveCursor := Cursor;
    Cursor := crHourglass;
    try
        ApdxFaxConverter1.ConvertToFile;
    finally
        Cursor := SaveCursor;
    end;
end;

```

This event changes the form's cursor to an hourglass, converts the file you specified in the DocumentFile property, and then changes the cursor back to what it was before the conversion.

Next, click on the TApdFaxConverter component, then click on the “Events” tab in the Object Inspector. Double click on the “OnStatus” event. A shell of an OnStatus event handler is generated for you. Modify the shell to match the following code:

```
procedure TForm1.ApdxFaxConverter1Status(  
    F : TObject; Starting, Ending : Boolean;  
    PagesConverted, LinesConverted : Integer;  
    BytesToConvert, BytesConverted : LongInt;  
    var Abort : Boolean);  
begin  
    if (BytesConverted <> 0) then begin  
        Label1.Caption := Format('Conversion is %d%% complete',  
            [(BytesToConvert * 100) div BytesConverted]);  
        Label1.Refresh;  
    end;  
    Abort := False;  
end;
```

This procedure displays the progress of the conversion operation. You could also take advantage of the “PagesConverted” and “LinesConverted” parameters to display additional status information.

Now, save the project and run it. Click on the button. After a few moments, the hourglass cursor should disappear and you will have, in the same directory as your input file, a file called MYFILE.APF (where MYFILE is the first part of the filename that you chose for the DocumentFile property). You can view this file with the VIEWER demonstration program, or with the viewer in TCom.

The CVT2FAX demonstration program provides a more extensive example of converting files to APF format.

Using shell execute

The TApdFaxConverter component can convert many file formats into APF files when the InputDocumentType property is idShell. When the InputDocumentType property is idShell, the TApdFaxConverter component will use the ShellExecute API method to execute the application associated with the selected file type and print the document to the TurboPower fax printer driver. Unlike other InputDocumentTypes that you may have used in the past, idShell uses the application that created the document to print to the printer driver (i.e. Microsoft Word would be the one to send a .DOC file). Essentially, this would use the ShellExecute with the “printto” parameter to print the specified document to our “APF Fax Printer” or “Print To Fax” depending on your operating system.

If the application associated with the selected file format does not support the “printto” verb, but does support the “print” verb, then `idShell` will change the default printer to the fax printer driver, print the document using `ShellExecute` and the “print” verb, and then change back to the original default printer. For example, in Windows 95 Notepad does not support the “printto”, but does support “print”, but both are supported in Windows 98 and 2000. An exception is raised if the application does not support the “printto” or “print” verbs.

The following example converts `C:\MYDOC.DOC` to an APF file:

```
OpenDialog1.Filter := 'Any file(*.*)|*.*';
if OpenDialog1.Execute then begin
    ApdFaxConverter1.DocumentFile := OpenDialog1.FileName;
    ApdFaxConverter1.InputDocumentType := idShell;
    ApdFaxConverter1.ConvertToFile;
end; //End if
```

The printer driver will not generate `TApdFaxDriverInterface` events when a document is printed using the `TApdFaxConverter` component. When a document is being converted, two registry keys are added. One is the window handle that will receive an `APW_ENDDOC` message indicating that the print job is complete. The other is the name of the output file. See the protected `ConvertShell` method (not documented) in `AdFaxCnv.pas` for details on the specific registry keys. If either of these keys is present, the `TApdFaxDriverInterface` component's `OnDocStart` and `OnDocEnd` events are not generated.

Hierarchy

TComponent (VCL)	
❶ TApdBaseComponent (OOMisc)	8
TApdCustomFaxConverter (AdFaxCvt)	
TApdFaxConverter (AdFaxCvt)	

Properties

DefUserExtension	LeftMargin	TabStop
DocumentFile	LinesPerPage	TopMargin
EnhFont	Options	❶ Version
FontFile	OutFileName	Width
FontType	Resolution	
InputDocumentType	StationID	

Methods

CloseFile	ConvertBitmapToFile	MakeEndOfPage
CompressRasterLine	ConvertToFile	OpenFile
Convert	GetRasterLine	

Events

OnCloseUserFile	OnOutputLine	OnStatus
OnOpenUserFile	OnReadUserLine	

Reference Section

CloseFile

method

```
procedure CloseFile;
```

↳ Closes an image file.

If you need to manually process input image files without converting them to APF files, use `CloseFile` to close a previously opened (using `OpenFile`) image file. See “Processing image files” on page 596 for more information.

See `CompressRasterLine` for an example of the use of `CloseFile`.

See also: `CompressRasterLine`, `GetRasterLine`, `OpenFile`

CompressRasterLine

method

```
procedure CompressRasterLine(  
    var Buffer, OutputData; var OutLen : Integer);
```

↳ Performs Group 3 compression on the data line.

If you need to write your own fax utilities and routines that output fax files, you must be able to output Group 3 compressed raster data. `Buffer` should contain one raster line of data. `CompressRasterLine` compresses the data in `Buffer` in Group 3 format and puts the result in `OutputData`. `OutputData` should be at least 512 bytes long. `OutLen` is the length of the compressed data.

For more information about APF files, see “Fax file format” on page 591.

The following example reads a line of raster data from an input file, compresses it, and writes it to disk:

```
var  
    Buffer    : array[1..512] of Byte;  
    OutBuf   : array[1..512] of Byte;  
    BufLen   : Integer;  
    OutLen   : Integer;  
    EOP      : Boolean;  
    More     : Boolean;  
    OutFile  : File;  
...  
...
```

```

AssignFile(OutFile, 'C:\COMPRESS.IMG');
Rewrite(OutFile, 1);
...
ApdFaxConverter1.DocumentFile := 'C:\COMPRESS.IMG';
ApdFaxConverter1.OpenFile;
EOP := False;
while not EOP do begin
    ApdFaxConverter1.GetRasterLine(Buffer, BufLen, EOP, More);
    {make sure buffer length is 1728 pixels}
    if (BufLen < 216) then
        FillChar(Buffer[BufLen + 1], 0, 216 - BufLen);
    ApdFaxConverter1.CompressRasterLine(Buffer, OutBuf, OutLen);
    BlockWrite(OutFile, OutBuf, OutLen);
end;
CloseFile(OutFile);
ApdFaxConverter1.CloseFile;

```

See also: [MakeEndOfPage](#)

Convert

method

```
procedure Convert;
```

↪ Converts the input image file, outputting raster data to a user event.

Convert reads each raster line from the input image file (specified by the `DocumentFile` property), compresses it in Group 3 format, and passes the compressed data to the `OnOutputLine` event. In that event, process the compressed data (either output it to a file or use it for some other purpose).

If `InputDocumentType` equals `idUser`, the `OnOpenUserFile`, `OnReadUserLine`, and `OnCloseUserFile` events are called to open, read, and close the image file. See “Converting user-defined image files” on page 598 for more information.

The following example converts an image file and writes the compressed Group 3 data to a file:

```
const
  OutFileOpened : Boolean = False; OutFile : File;
procedure TForm1.ApdfFaxConverter1OutputLine(
  F : TObject; Data : PByteArray; Len : Integer;
  EndOfPage, MorePages : Boolean);
var
  EOPBuf : array[1..64] of Byte;
  EOPLen : Integer;
begin
  if not OutFileOpened then begin
    AssignFile(OutFile, 'C:\OUTPUT.IMG');
    Rewrite(OutFile, 1);
    OutFileOpened := True;
  end;

  if not EndOfPage then
    BlockWrite(OutFile, Data^, Len);

  if EndOfPage then begin
    ApdfFaxConverter1.MakeEndOfPage(EOPBuf, EOPLen);
    BlockWrite(OutFile, EOPBuf, EOPLen);
    if not MorePages then begin
      CloseFile(OutFile);
      OutFileOpened := False;
    end;
  end;
end;

...
ApdfFaxConverter1.DocumentFile := OpenDialog.FileName;
ApdfFaxConverter1.Convert;
```

See also: ConvertToFile, DocumentFile, InputDocumentType, OnCloseUserFile, OnOpenUserFile, OnOutputLine, OnReadUserLine

```
procedure ConvertBitmapToFile(Bmp : TBitmap);
```

↪ Converts a memory bitmap to an APF file.

ConvertToFile reads each raster line from the input bitmap (specified by the Bmp parameter), compresses it in Group 3 format, and writes the compressed data (along with the relevant document and page headers) in the APF file specified by OutFileName.

If the input bitmap is a color image, it will automatically get dithered to a monochrome image as part of the conversion process. If the OutFileName property does not specify a file extension, the default extension (DefFaxFileExt = "APF") is appended to the output filename.

See also: OutFileName

```
procedure ConvertToFile;
```

↪ Converts the input image file to an APF file.

ConvertToFile reads each raster line from the input image file (specified by DocumentFile), compresses it in Group 3 format, and writes the compressed data (along with the relevant document and page headers) in the APF file specified by OutFileName. If the InputDocumentType is idBmp and the input bitmap is a color image, it will automatically get dithered to a monochrome image as part of the conversion process.

If the OutFileName property does not specify a file extension, the default extension (DefFaxFileExt = "APF") is appended to the output filename. If InputDocumentType equals idUser, the OnOpenUserFile, OnReadUserLine, and OnCloseUserFile events are called to open, read, and close the image file. See "Converting user-defined image files" on page 598 for more information.

The following example converts a TIFF file to an APF file:

```
ApdFaxConverter1.DocumentFile := 'C:\MYIMAGE.TIF';  
ApdFaxConverter1.InputDocumentType := idTiff;  
ApdFaxConverter1.OutFileName := 'C:\FAX.APF';  
ApdFaxConverter1.ConvertToFile;
```

See also: DocumentFile, InputDocumentType, OutFileName

```
property DefUserExtension : string
```

↳ The default extension for user-defined input image files.

When a user-defined image file is converted and the `DocumentFile` property does not specify an extension, `DefUserExtension`, if any, is appended to the filename.

See “Converting user-defined image files” on page 598 for more information.

See also: `OnCloseUserFile`, `OnOpenUserFile`, `OnReadUserLine`

DocumentFile

property

```
property DocumentFile : string
```

↳ Specifies the name of the input image file.

When `Convert`, `ConvertToFile`, or `OpenFile` is called, the `TApdFaxConverter` component attempts to open the file specified in `DocumentFile`.

If the filename specified in `DocumentFile` does not contain an extension, a default extension is appended when the file is opened. The default extension depends on the value of `InputDocumentType`:

InputDocumentType	Default Extension
<code>idNone</code>	No default
<code>idText</code>	TXT
<code>idTextEx</code>	TXT
<code>idTiff</code>	TIF
<code>idPcx</code>	PCX
<code>idDcx</code>	DCX
<code>idBmp</code>	BMP
<code>idUser</code>	The value of the <code>DefUserExtension</code> property

Unless you are sure of the current directory, the value of `DocumentFile` should be a fully-qualified (i.e., containing drive and directory information) filename.

The following example sets up a `TApdFaxConverter` to convert a TIFF file:

```
ApdFaxConverter1.DocumentFile := 'C:\MYIMAGE.TIF';
ApdFaxConverter1.InputDocumentType := idTiff;
```

See also: `Convert`, `ConvertToFile`, `OpenFile`

```
property EnhFont : TFont
```

↪ Determines the font used by the fax converter.

If `InputDocumentType` is `idTextEx`, the `FontFile` and `FontType` properties are ignored and the font specified by `EnhFont` is used by the fax converter instead. Any font available to Windows can be used (double click on the property to invoke the font dialog and see a list of the fonts). Only one font can be used for a document (i.e., font sizes and types cannot be mixed within a single document).

There is an upper limit on the size of the font, but this limit is not typically reached unless a very large font is used (e.g., greater than 72 pt). If the limit is exceeded, an `ecEnhFontTooBig` error occurs during the conversion process.

The fax converter makes no attempt to keep all text on the page when the size of the font is changed. You must ensure that the line length in the text file fits on the page in the desired font. You might also need to adjust the `LinesPerPage` property to keep the lines on the page.

See also: `FontFile`, `InputDocumentType`

```
property FontFile : string
```

↪ Specifies the filename of the font file used by the ASCII text converter.

When an ASCII text file is opened or converted and `InputDocumentType` is `idText`, built-in fonts supplied in `APFAX.FNT` are used. If the compiler define `BindFaxFont` in `AWFAXCVT.PAS` is activated (the default), `APFAX.FNT` is bound directly into your program.

If `BindFaxFont` is not activated, the file specified in `FontFile` is loaded into memory. `FontFile` must be the fully-qualified name of the font file.

See also: `Convert`, `ConvertToFile`, `EnhFont`, `FontType`, `InputDocumentType`, `OpenFile`

```
property FontType : TFaxFont
TFaxFont = (ffStandard, ffSmall);
```

Default: ffStandard

☞ Specifies the size of the font used to convert ASCII text files.

FontType is used only if InputDocumentType is idText (meaning that the default Async Professional font file is used). The default font file contains two sizes of fonts. If you want to use other fonts, see the EnhFont property.

ffStandard is a 20x16 font (20 pixels wide by 16 pixels high). This font allows about 8.5 characters per horizontal inch (about 85 characters per line in a standard width fax), and about 12.5 lines per vertical inch.

ffSmall is a 12x8 font that allows for about 14 characters per horizontal inch (about 144 characters per line in a standard width fax), and about 25 lines per vertical inch.

The following example converts a text file to an APF file using a small font:

```
ApdFaxConverter1.DocumentFile := 'C:\MYFILE.TXT';
ApdFaxConverter1.InputDocumentType := idText;
ApdFaxConverter1.OutFileName := 'C:\FAX.APF';
ApdFaxConverter1.FontType := ffSmall;
ApdFaxConverter1.ConvertToFile;
```

See also: EnhFont, FontFile, InputDocumentType

```
procedure GetRasterLine(var Buffer; var BufLen : Integer;
    var EndOfPage, MorePages : Boolean);
```

☞ Reads a raster line from an input image file.

GetRasterLine is used to manually read a line of raster data from an input image file. A call to GetRasterLine must be preceded by a call to OpenFile.

GetRasterLine returns the raster data in Buffer. BufLen contains the length, in bytes, of the raster data. EndOfPage is set to True if the end of the input page has been reached. MorePages is set to True if there are additional pages in the input file.

See “Processing image files” on page 596 for more information about reading image files.

The following example opens an image file, reads the data from it, and closes it:

```
var
  Buffer : array[1..512] of Byte;
  BufLen : Integer;
  EOP    : Boolean;
  More   : Boolean;

...
ApdFaxConverter1.DocumentFile := OpenDialog.FileName;
ApdFaxConverter1.OpenFile;
EOP := False;
while not EOP do begin
  ApdFaxConverter1.GetRasterLine(Buffer, BufLen, EOP, More);
  ...process the image data...
end;
ApdFaxConverter1.CloseFile;
```

See also: [OpenFile](#)

InputDocumentType

property

```
property InputDocumentType : TFaxInputDocumentType

TFaxInputDocumentType = (idNone, idText, idTextEx, idTiff,
  idPcx, idDcx, idBmp, idBitmap, idUser);
```

Default: idNone

☞ Specifies the type of the input image file.

The TApdFaxConverter component can read and convert a variety of input image files. With the exception of idBmp and idBitmap, all input image files must be monochrome images. That is, they can contain no more than one bit per pixel.

The value of InputDocumentType specifies the type of image. The following table shows the possible values of InputDocumentType:

InputDocumentType	Image Type
idNone	No input image
idText	ASCII text file (using built-in fonts)
idTextEx	ASCII text file (using font specified by EnhFont)
idTiff	Tagged Image File Format (TIFF)
idPcx	PC Paintbrush image file (PCX)
idDcx	Multi-page PCX file
idBmp	Windows 3.x bitmap file

InputDocumentType	Image Type
idBitmap	Memory bitmap
idUser	User-defined input image

TApdFaxConverter, using the default font file, can convert CR/LF delimited ASCII text files that have characters in the Windows OEM character set. If the text file contains non-OEM characters, they will not be converted correctly unless you provide another font.

The idText and idTextEx values are both used for ASCII text files. If InputDocumentType is idText, the Async Professional built-in fonts are used to convert the text file. This means that the font in the file specified by FontFile is used. If InputDocumentType is idTextEx, one of the Windows fonts is used to convert the text file. This means that the font specified by EnhFont is used (FontFile and FontType are ignored).

TIFF files can be single or multi-strip images, but must contain either uncompressed raster data or MacPaint compressed raster data. The byte-order of the file can be either Intel (little endian, where words are stored in byte-reversed order) or Motorola (big endian, where words are stored high byte first).

Input BMP files must be uncompressed.

User-defined input images can be in any format. You must ensure, however, that raster data passed back to the TApdFaxConverter component is encoded such that one bit of raster data represents one pixel of input image. See “Converting user-defined image files” on page 598 for more information about converting unsupported image files.

The following example sets a TApdFaxConverter component up to convert a text file:

```
ApdFaxConverter1.DocumentFile := 'C:\MYFILE.TXT';
ApdFaxConverter1.InputDocumentType := idText;
```

See also: Convert, ConvertToFile, EnhFont, FontFile, OpenFile

property LeftMargin : Cardinal

Default: 50

↪ Specifies the width in pixels of the left margin in the APF file.

To make output faxes look more attractive, the TApdFaxConverter can add a fixed left margin to all pages in the fax. This is necessary for some fax machines (and some viewing software) that print or display the left edge of the fax too close to the edge of the page or the visible screen. The default left margin of 50 pixels provides a reasonable amount of white space at the left edge of the page, without using too much of the horizontal space. On a standard width fax, 50 pixels consumes only 3% of the horizontal space.

See also: TopMargin

property LinesPerPage : Cardinal

Default: 60

↪ The number of ASCII text lines on each fax page.

The TApdFaxConverter can convert a text file into a fax and leave all of the data on one page. With large text files, however, this becomes a problem. Some continuous roll paper fax machines can print a page like this on one sheet of paper, but it becomes extremely long if the fax file is more than a few hundred lines.

It is probably more reasonable to break large text files up into multiple fax pages. To do this, set the LinesPerPage property to the number of lines you want on each page. If you are using the default font, the default LinesPerPage of 60 creates standard letter-sized pages about 10 inches long. If you are using one of the Windows fonts, you may have to experiment with LinesPerPage to determine what fits on a page. If you want all of the text to appear on a single fax page, set LinesPerPage to 0.


```
procedure MakeEndOfPage(var Buffer; var BufLen : Integer);
```

↳ Generates an end-of-page code.

Each fax page ends with a sequence of eight end-of-line fax codes. These codes indicate to the receiving fax machine that the end of a page has been reached. If you are creating APF files yourself, you must put these codes at the end of each page you create.

MakeEndOfPage puts eight end-of-line codes into Buffer. BufLen contains the length of the codes (this length varies, depending on whether you are creating a standard width or wide fax). You should write the data contained in Buffer to the end of the fax page you are creating. The buffer passed to MakeEndOfPage should be at least 64 bytes in length.

For more information about APF files, see the “Fax file format” on page 591.

See the Convert method on page 605 for an example.

See also: Convert, Width

OnCloseUserFile**event**

```
property OnCloseUserFile : TFaxCloseFileEvent
```

```
TFaxCloseFileEvent = procedure(F : TObject) of object;
```

↳ Defines an event handler that is called to close a user-defined image file.

When InputDocumentType is idUser, the TApdFaxConverter calls event handlers to open, read, and close a user-defined image file. OnCloseUserFile is called to close the input file. F contains a pointer to the fax converter component that called the event. When this event is called, you should close the image file and destroy any buffers related to reading and processing the image data.

See “Converting user-defined image files” on page 598 for more information.

See also: OnOpenUserFile, OnReadUserLine

```
property OnOpenUserFile : TFaxOpenFileEvent
```

```
TFaxOpenFileEvent = procedure(  
    F : TObject; FName : string) of object;
```

↳ Defines an event handler that is called to open a user-defined image file.

When `InputDocumentType` is `idUser`, the `TApdFaxConverter` calls event handlers to open, read, and close a user-defined image file. `OnOpenUserFile` is called to open the input file. `F` contains a pointer to the fax converter component that called the event. `FName` contains the name of the file.

Use the value passed in `FName` to open the file. Do not use the value in `DocumentFile` because it is not guaranteed to have a file extension. `FName` is generated from the value in `DocumentFile` and `DefUserExtension` (if there is not extension in `DocumentFile`).

See “Converting user-defined image files” on page 598 for an example of converting an unsupported image file type.

The following example demonstrates the use of the `OnOpenUserFile`, `OnReadUserLine`, and `OnCloseUserFile` events:

```
var  
    InputFile : File;  
    LineLen : Integer;  
    BytesProcessed : LongInt;  
    TotalBytes : LongInt;  
  
procedure Form1.ApdFaxConverter1OpenUserFile(  
    F : TObject; FName : string);  
begin  
    AssignFile(InputFile, FName);  
    Reset(InputFile, 1);  
    ...read file header...  
    LineLen := WidthInBytesAsReadFromImageHeader;  
    BytesProcessed := 0;  
    TotalBytes := FileSize(InputFile) - SizeOf(Header);  
end;
```

```

procedure Form1.ApdxFaxConverter1ReadUserLine(
  F : TObject; Data : PByteArray; var Len : Integer;
  var EndOfPage, MorePages : Boolean; var BytesRead,
  BytesToRead : LongInt);
begin
  BlockRead(InputFile, Data^, LineLen, Len);
  Inc(BytesProcessed, Len);
  EndOfPage := Eof(InputFile);
  MorePages := False;
  BytesRead := BytesProcessed;
  BytesToRead := TotalBytes;
end;

procedure Form1.ApdxFaxConverter1CloseUserFile(F : TObject);
begin
  CloseFile(InputFile);
end;

```

See also: OnCloseUserFile, OnReadUserLine

OnOutputLine

event

```

property OnOutputLine : TFaxOutputLineEvent

TFaxOutputLineEvent = procedure(
  F : TObject; Data : PByteArray; Len : Integer;
  EndOfPage, MorePages : Boolean) of object;

```

✚ Defines an event handler that is called to output a line of Group 3 compressed data.

When the Convert method is called, each line of raster data is read from the input image, compressed in Group 3 format, and passed to the OnOutputLine event. You can then process the compressed data.

F contains a pointer to the fax converter component that generated the event. Data is a pointer to an array of bytes that contain the compressed data. Len is the length of the compressed data.

If EndOfPage is True, the end of a page of input data has been reached. You should call MakeEndOfPage at this point, to output an end-of-page code. If MorePages is True, there are more pages of data to compress and output. If MorePages is False, you can dispose of any buffers and other data that are required for your output.

See also: Convert, MakeEndOfPage

```
property OnReadUserLine : TFaxReadLineEvent

TFaxReadLineEvent = procedure(
    F : TObject; Data : PByteArray; var Len : Integer;
    var EndOfPage, MorePages : Boolean) of object;
```

↳ Defines an event handler that is called to read a line of data from a user-defined image file.

When `InputDocumentType` is `idUser`, the `TApdFaxConverter` calls event handlers to open, read, and close a user-defined image file. `OnReadUserLine` is called each time a new line of raster data must be compressed. `Data` is a pointer to a 0-based array of bytes. On return from this function, it should contain a 1-bit-per-pixel representation of the data to be compressed. The bits should be on for black pixels and off for white pixels. `Len` should be equal to the length of the data.

`EndOfPage` should be set to `True` if the end of the user-defined page has been reached. If `EndOfPage` is `True`, the value of `MorePages` should indicate whether any more pages are available. If `MorePages` is `True`, the `TApdFaxConverter` begins a new page and begins calling `OnReadUserLine` for more data. If `MorePages` is `False`, the conversion process ends. See “Converting user-defined image files” on page 598 for more information.

See also: `OnCloseUserFile`, `OnOpenUserFile`, `OnStatus`

OnStatus

event

```
property OnStatus : TFaxStatusEvent

TFaxStatusEvent = procedure(
    F : TObject; Starting, Ending : Boolean;
    PagesConverted, LinesConverted : Integer;
    BytesConverted, BytesToConvert : LongInt;
    var Abort : Boolean) of object;
```

↳ Defines an event handler that is called to notify the user of the status of a conversion operation.

During the conversion process, the `TApdFaxConverter` regularly calls the `OnStatus` event to notify the user of the progress of the conversion.

If `Starting` is `True`, the conversion of the document is just beginning. This is the appropriate time for you to do pre-conversion work (e.g., show the status display form).

If `Ending` is `True`, the conversion of the document is about to end. This is the appropriate time for you to do post-conversion work (e.g., destroy the form that you were using to display the conversion status).

PagesConverted is the number of pages that have been processed in the input document. PagesConverted is equal to 1 after the conversion of the first page is started, then equal to 2 after the conversion of the second page is started, and so on. LinesConverted is the number of raster lines that have been read and compressed on the current page.

BytesConverted is the number of image bytes that have been read from the input file. BytesToConvert is the total number of bytes that will be read from the input file. These two values can be used to create a “percent complete” style progress bar for the conversion process.

Abort determines whether the conversion process will terminate prematurely. Set Abort to True if you need to abort the conversion process.

The following example shows how to implement a percent complete indicator for a fax converter:

```
procedure Form1.ApdxFaxConverter1Status(  
    F : TObject; Starting, Ending : Boolean;  
    PagesConverted, LinesConverted : Integer;  
    BytesConverted, BytesToConvert : LongInt; var Abort : Boolean);  
const  
    Frm : TConvertStatusForm = nil;  
begin  
    if Starting then begin  
        Frm := TConvertStatusForm.Create(Application);  
        Frm.Show;  
    end else if Ending then begin  
        Frm.Close;  
        Frm.Free;  
    end else begin  
        if Frm.AbortBtnClicked then  
            Abort := True  
        else  
            {show progress}  
            Frm.Label1.Caption := Format(  
                'Conversion is %d percent complete',  
                [(BytesConverted * 100) div BytesToConvert]);  
        end;  
    end;  
end;
```

```
procedure OpenFile;
```

↳ Opens an image file.

If you need to process an image file without converting it to an APF file, use `OpenFile` to open the file specified by `DocumentFile` and `InputDocumentType`. Then use `GetRasterLine` to read raster data from the image file. When you are finished reading the image, use `CloseFile` to close the image file.

See “Processing image files” on page 596 for more information.

See also: `CloseFile`, `CompressRasterLine`, `DocumentFile`, `GetRasterLine`, `InputDocumentType`

Options

property

```
property Options : TFaxCvtOptionsSet
```

```
TFaxCvtOptionsSet = Set of TFaxCvtOptions;
```

```
TFaxCvtOptions = (coDoubleWidth, coHalfHeight, coCenterImage,  
  coYield, coYieldOften);
```

Default: [`coDoubleWidth`, `coCenterImage`, `coYield`]

↳ Sets optional features for the fax converter.

The `TApdFaxConverter` optional features are turned on and off by adding or subtracting elements from the `Options` property. The valid `Options` are:

Option	Result
<code>coDoubleWidth</code>	Double the width of the input image.
<code>coHalfHeight</code>	Halve the height of the input image.
<code>coCenterImage</code>	Center images in page.
<code>coYield</code>	Occasionally relinquish control to Windows.
<code>coYieldOften</code>	Same as <code>coYield</code> , but done more frequently.

`coDoubleWidth` and `coHalfHeight` adjust for the difference between standard and high resolution faxes. If neither of these options are on, and an image is converted to a standard resolution fax (Resolution equals `frNormal`), the resulting fax looks vertically elongated. If `coDoubleWidth` is on, this effect is compensated for by doubling the width of the input image (if possible), causing the standard resolution fax to look normal. Doubling is not

possible if the new width would be wider than the width specified by the Width property. If coHalfHeight is on, this effect is compensated for by discarding every other raster line of input image.

coDoubleWidth and coHalfHeight are mutually exclusive (they cannot both be on). Attempts to add one of these options to the option set when the other is already in the set are ignored.

If coCenterImage is on (the default), converted image files (not text files) are centered on the fax page. If coCenterImage is not on, converted image files are placed flush left on the page.

If coYield is on (the default), the TApdFaxConverter yields to Windows at the end of every converted page, giving other applications a chance to run. If coYield is not on, the TApdFaxConverter hogs the system for the amount of time required to convert the input file. This results in a faster conversion, but is not recommended.

coYieldOften is the same as coYield, except that yielding is much more frequent. Control is relinquished by the converter at the end of every converted raster line. This results in a slower conversion, but Windows runs better. If coYieldOften is turned on, coYield is turned on automatically.

The following example turns on the yielding features of the converter:

```
{make sure the converter yields regularly}
ApdFaxConverter1.Options := ApdFaxConverter1.Options +
  [coYield, coYieldOften];
```

See also: Width

OutFileName **property**

property OutFileName : string

↳ Specifies the name of the output fax file.

When ConvertToFile is called, it creates a file with the name specified by OutFileName and puts the compressed fax data in that file.

OutFileName should be a fully qualified path name. If the file specified by this property already exists, it is overwritten without warning.

The following example converts a text file to an APF file that is stored in C:\FAX.APF:

```
ApdFaxConverter1.DocumentFile := 'C:\MYFILE.TXT';
ApdFaxConverter1.InputDocumentType := idText;
ApdFaxConverter1.OutFileName := 'C:\FAX.APF';
ApdFaxConverter1.ConvertToFile;
```

See also: ConvertToFile

```
property Resolution : TFaxResolution
```

```
TFaxResolution = (frNormal, frHigh);
```

Default: frNormal

↪ Specifies the resolution of output fax data.

Fax images can be converted and stored using two different resolutions. Standard resolution is 200 horizontal dots per inch by 100 vertical dots per inch. High resolution has the same horizontal resolution, but uses 200 vertical dots per inch. In some fax documentation, the resolutions are described as 98 dots per inch and 196 dots per inch. Those numbers are actually more exact, but 100 and 200 are easier to remember and are commonly used in most fax documentation.

Resolution specifies which of the two resolutions is used. If Resolution is frNormal, the output resolution is 200x100. If Resolution is frHigh, the output resolution is 200x200.

If the file being converted is an image (i.e., not an ASCII text file), then the resultant fax might appear stretched in the vertical aspect. To deal with this, use the coDoubleWidth or coHalfHeight options.

See also: Options, Width

```
property StationID : string
```

↪ The station ID of the faxmodem.

A fax device can identify itself to another fax device with a 20 character name, called the station ID. The Class 1, Class 2, and Class 2.0 specifications indicate that the station ID should contain just a phone number; therefore they limit it to just the digits 0 through 9 and space. However, the station ID is frequently used to store an alphabetic name. Most faxmodems support this convention by allowing upper and lower case letters, as well as other special characters in the station ID. This can cause problems for some fax machines, though, since they cannot print these characters.

Async Professional does not filter the characters stored in the station ID. If your software must be compatible with the broadest possible range of fax hardware, you might want to limit the characters stored in StationID.

StationID is stored in the header of the converted APF file. This string is not used when the fax is transmitted, since the StationID property of TApdAbstractFax is used to determine the string that is sent through the faxmodem. If you want to send the station ID embedded in the APF file, you should read the SenderID field from the fax header and use it to set the StationID property of TApdAbstractFax before sending.

See also: TApdAbstractFax.StationID

TabStop **property**

property TabStop : Cardinal

Default: 4

↳ Specifies the size of expanded tabs in ASCII text files.

During a fax conversion, tab characters (\$09) in the input text are expanded to one to TabStop space characters (\$20).

To demonstrate how space characters are inserted, these examples use the default TabStop of 4. If the input data is:

```
<tab>This is a test
```

The tab character is expanded to:

```
<space><space><space><space>This is a test
```

If the input data is:

```
This is a<tab>test
```

The tab character is expanded to:

```
This is a<space><space><space>test
```

Only three spaces are needed because the word “test” is only three spaces away from a tabstop.

```
property TopMargin : Cardinal
```

Default: 0

↪ Specifies the size in raster lines of the top margin in the APF file.

To avoid problems with fax machines that print faxes too close to the top of the paper (thereby distorting the image/text near the top of the page), the TApdFaxConverter can add a fixed-sized region of white space at the top of every page.

TopMargin is the number of blank raster lines added to the top of every converted page. The visible amount of white space varies depending on the value of Resolution (i.e., the margin appears smaller on high resolution faxes).

See also: LeftMargin, Resolution

```
property Width : TFaxWidth
```

```
TFaxWidth = (fwNormal, fwWide);
```

Default: fwNormal

↪ Specifies the width of output faxes.

The standard width of a fax page is 1728 pixels per row (about 8.5 inches). Async Professional supports one additional width: 2048 pixels per row (about 10 inches).

In most cases, the standard width of 1728 pixels is adequate. The larger width of 2048 is provided for special cases in which you are certain that the remote fax device can handle a wider fax. The Group 3 specification lists any width other than 1728 pixels as “optionally” supported, so support for 2048 pixel wide faxes varies from manufacturer to manufacturer.

See also: Resolution

TApdFaxUnpacker Component

Fax images are transmitted and received in a compressed bitmap image format. The compressed data must then be unpacked before you can view, print, or edit the fax.

The unpacking methods of the TApdFaxUnpacker component handle all of the work of opening the APF file, finding the desired page, and uncompressing each raster line—but that's as far as the general purpose routines can go. The unpacked raster lines are passed to your application through an event handler. Your application can print, display, or otherwise process the line.

Async Professional provides additional components to perform some of the most common operations for fax files: viewing, printing and converting to a graphics format. To view faxes, use the TApdFaxViewer component (see page 649). Printing is performed by the TApdFaxPrinter component (see page 674). The TApdFaxUnpacker itself can convert a fax to a different graphics format. It can unpack a fax into a memory bitmap (i.e., a VCL TBitmap instance), as well as a BMP, PCX, DCX, or TIFF image file. If you need to perform any other processing on a fax file, you must implement an OnOutputLine event handler.

OnOutputLine event

When the UnpackPage or UnpackFile methods of the TApdFaxUnpacker component are called, the data in the APF file is decompressed and passed to an OnOutputLine event handler.

In the following example an OnOutputLine event handler writes the raw raster data to a file:

```
var
    OutFile : File;
...
procedure Form1.ApdFaxUnpackerOutputLine(
    Sender : TObject; Starting, Ending : Boolean;
    Data : PByteArray; Len, PageNum : Integer);
begin
    if Starting then begin
        AssignFile(OutFile, 'C:\MYIMAGE.IMG');
        Rewrite(OutFile, 1);
    end else if Ending then begin
        CloseFile(OutFile);
    end else
        BlockWrite(OutFile, Data^, Len);
end;
```

Sender is the object instance of the TApdFaxUnpacker component that generated the event. If Starting is True, the fax unpack process is just beginning. In this example, Starting is used to determine when to open the output file.

If Ending is True, the fax unpack process is ending (due to successful completion of the unpack or an error condition). In this example, Ending is used to determine when to close the output file.

Data is a pointer to a 0-based array of bytes that contains the decompressed data. Len is the length of the data. In this example, Len bytes are written from Data to the file OutFile.

PageNum contains the number of the page that is currently being unpacked. This can be used to determine when a page change occurs (this is important when converting multi-page faxes to multi-page image formats).

Memory bitmaps

The TApdFaxUnpacker can unpack a fax file (or page) to a TBitmap class. This is useful if you want to unpack a fax and manipulate the image before using it, or if you simply want to copy a fax to a TCanvas (or to the Picture property of a TImage component).

The UnpackFileToBitmap and UnpackPageToBitmap methods of the TApdFaxUnpacker component unpack a fax file and copy the uncompressed data into a VCL TBitmap instance. The following code demonstrates the use of these routines by displaying the fax data in a TImage component:

```
var
    Bmp : TBitmap;
    Imgel : TImage;

begin
    ApdFaxUnpacker1.InFileName := 'D:\MYFAX.APF';
    try
        Bmp := ApdFaxUnpacker1.UnpackFileToBitmap;
        Imgel.Picture.Bitmap := Bmp;
    except
        MessageDlg('Unpack failed!', mtError, [mbOK], 0);
    end;
end;
```

Additionally, you can use the Canvas property of the Bitmap class to place additional graphics onto a fax image (for example, to create a custom cover page). The following example demonstrates the technique:

```
Form1 = class(TForm)
    Image1 : TImage;
    ...
procedure TForm1.BitBtn1Click(Sender : TObject);
var
    Bmp1 : TBitmap;
    Bmp2 : TBitmap;
begin
    ApdFaxUnpacker1.InFileName := 'D:\MYFAX1.APF';
    Bmp1 := ApdFaxUnpacker1.UnpackPageToBitmap(1);

    ApdFaxUnpacker1.InFileName := 'D:\MYFAX2.APF';
    Bmp2 := ApdFaxUnpacker1.UnpackPageToBitmap(1);

    Bmp1.Canvas.Draw(0, 0, Bmp2);
    Bmp2.Free;

    Image1.Picture.Bitmap := Bmp1;
end;
```

This example unpacks MYFAX1.APF and MYFAX2.APF into memory bitmaps. The bitmap of MYFAX2.APF is then placed on top of the bitmap of MYFAX1.APF by the call to Draw. Finally, the image is drawn on the form by placing the new bitmap (in Bmp1) into the Picture property of Image1 (a TImage component).

Scaling

The TApdFaxUnpacker component can scale (i.e., make the size larger or smaller) a fax as it is unpacked. When you call one of the TApdFaxUnpacker UnpackXxx methods, the Scaling property is examined to see if the output image should be scaled (Scaling equals True).

The image is scaled depending on the values of four properties. HorizMult and HorizDiv are combined to form a fraction (HorizMult/HorizDiv) that is multiplied by the width (horizontal aspect) of the fax to determine its new width. VertMult and VertDiv are combined to form a fraction (VertMult/VertDiv) that is multiplied by the height (vertical aspect) of the fax to determine its new height.

For example, assume a standard resolution fax (200x100) is being unpacked. When the fax is unpacked and converted into square pixels, the resulting image looks shorter than it should (since, in the original, the height of the pixels was twice as large as width of the pixels). To compensate for this, you can use the following:

```
ApdFaxUnpacker1.Scaling      := True;  
ApdFaxUnpacker1.HorizMult   := 1;  
ApdFaxUnpacker1.HorizDiv    := 1;  
ApdFaxUnpacker1.VertMult    := 2;  
ApdFaxUnpacker1.VertDiv     := 1;
```

This specifies that the unpacked fax data is to be scaled to be twice as tall ($\text{VertMult} / \text{VertDiv} = 2/1 = 2$) as it normally would be. This makes the unpacked standard resolution fax look normal.

Similarly, the following code achieves the same effect, but the resultant image is smaller:

```
ApdFaxUnpacker1.Scaling      := True;  
ApdFaxUnpacker1.HorizMult   := 1;  
ApdFaxUnpacker1.HorizDiv    := 2;  
ApdFaxUnpacker1.VertMult    := 1;  
ApdFaxUnpacker1.VertDiv     := 1;
```

This specifies that the width of the unpacked fax is to be halved ($\text{HorizMult} / \text{HorizDiv} = 1/2$). This, too, compensates for the difference in aspect ratio between a standard and high resolution fax, but the resulting image is smaller than the one produced in the previous example.

To make it easier to compensate for the aspect ratio of standard resolution faxes, the `AutoScaleMode` property allows you to specify that the scaling should be performed automatically. You can request either method—doubling the height or halving the width.

The `Scaling` property can be used in many ways to produce a nearly unlimited range of images. For instance, you can create an image that is 1/3 the size of the original fax with the following code:

```
ApdFaxUnpacker1.Scaling      := True;  
ApdFaxUnpacker1.HorizMult   := 1;  
ApdFaxUnpacker1.HorizDiv    := 3;  
ApdFaxUnpacker1.VertMult    := 1;  
ApdFaxUnpacker1.VertDiv     := 3;
```

You could create a thumbnail 32x42 image of the fax (assuming an 8.5" x 11" fax) with the following code:

```
ApdFaxUnpacker1.Scaling      := True;  
ApdFaxUnpacker1.HorizMult   := 1;  
ApdFaxUnpacker1.HorizDiv    := 54;  
ApdFaxUnpacker1.VertMult    := 1;  
ApdFaxUnpacker1.VertDiv     := 54;
```

White space compression

To make it easier to view large faxes that have a lot of white space, the TApdFaxUnpacker can compress a specified number of blank raster lines into a smaller number of blank raster lines. This feature can be used to save paper when printing a large volume of faxes. For example, if a page is slightly longer than 11 inches, the white space compression feature can often make it fit nicely on an 11 inch piece of paper, eliminating the need for an extra sheet of paper.

To use the white space compression feature, set WhitespaceCompression to True. Every occurrence of WhitespaceFrom or more consecutive blank lines is replaced with WhitespaceTo blank lines. For example, if WhitespaceFrom is 20 and WhiteSpaceTo is 5, then any occurrence of 20 or more consecutive blank lines is compressed to 5 blank lines.

Example

This example demonstrates the steps involved in unpacking a fax file to a memory bitmap. Create a new project, add the following components, and set the property values as indicated in Table 15.6.

Table 15.6: *Example components and property values*

Component	Property	Value
TApdFaxUnpacker	InFileName	<the name of an existing APF file>
TImage	Align	alClient

After adding the components and setting their properties, click on the combo box at the top of the Object Inspector and select “Form1.” Next, click on the “Events” tab at the bottom of the Object Inspector. From the events page, double click on the “OnCreate” event. A shell for an OnCreate event is generated for you. Modify the generated code to match this:

```
procedure TForm1.FormCreate(Sender : TObject);
var
    Bmp : TBitmap;
begin
    Bmp := ApdFaxUnpacker1.UnpackPageToBitmap(1);
    Image1.Picture.Bitmap := Bmp;
end;
```

This event loads the fax specified by InFileName into the TBitmap Bmp. This TBitmap is assigned to the Picture property of the TImage component and is automatically displayed on the screen.

Hierarchy

TComponent (VCL)

- TApdBaseComponent (OOMisc) 8
 - TApdCustomFaxUnpacker (AdFaxCvt)
 - TApdFaxUnpacker (AdFaxCvt)

Properties

AutoScaleMode
FaxResolution
FaxWidth
HorizDiv
HorizMult
InFileName

NumPages
Options
OutFileName
Scaling
Version
VertDiv

VertMult
WhitespaceCompression
WhitespaceFrom
WhitespaceTo

Methods

ExtractPage
UnpackFile
UnpackFileToBitmap
UnpackFileToBmp
UnpackFileToDcx

UnpackFileToPcx
UnpackFileToTiff
UnpackPage
UnpackPageToBitmap
UnpackPageToBmp

UnpackPageToDcx
UnpackPageToPcx
UnpackPageToTiff

Events

OnOutputLine

OnStatus

Reference Section

AutoScaleMode

property

```
property AutoScaleMode : TAutoScaleMode
TAutoScaleMode = (asNone, asDoubleHeight, asHalfWidth);
```

↪ Determines whether standard resolution faxes are automatically scaled.

When a standard resolution fax is unpacked, the resulting image looks shorter than it should. That is because in the fax file each pixel is 1/100th of an inch tall, but just 1/200th of an inch wide.

AutoScaleMode can be used to automatically adjust the width or height of a standard resolution fax so that it looks normal when unpacked into memory or to an image file.

AutoScaleMode can contain any of the following values:

Value	Result
asNone	No automatic scaling is performed.
asDoubleHeight	The height of the unpacked fax is doubled. If the Scaling property is True, this has the effect of doubling the value of the VertMult property.
asHalfWidth	The width of the unpacked fax is halved. If the Scaling property is True, this has the effect of multiplying HorizMult/HorizDiv by 1/2.

See Also: Scaling

ExtractPage

method

```
procedure ExtractPage(const Page : Cardinal);
```

↪ Extracts a single page in a fax file to an APF file.

ExtractPage reads and extracts the page specified by Page in the fax file specified by InFileName and writes the extracted page to the file specified by OutFileName. If the file name in OutFileName does not have an extension, a default extension of APF is appended. No scaling or conversion is performed on the extracted APF data.

If InFileName and OutFileName are the same, an ecAccessDenied exception will be raised. If the file specified by InFileName does not exist, an ecFileNotFound exception will be raised. If the file specified by OutFileName exists, it will be overwritten without warning.

The following example extracts each page in a fax to a separate fax file:

```
var
  I : Integer;
....
  ApdFaxUnpacker1.InFileName := OpenFileDialog1.FileName;
  for I := 1 to ApdFaxUnpacker1.NumPages do begin
    ApdFaxUnpacker1.OutFileName := 'PAGE' + IntToStr(I) + '.APF';
    ApdFaxUnpacker1.ExtractPage(I);
  end;
....
```

See also: InFileName, OutFileName

FaxResolution

read-only, run-time property

```
property FaxResolution : TFaxResolution
TFaxResolution = (frNormal, frHigh);
```

↳ The resolution of the fax.

If the file name specified in InFileName is valid, FaxResolution is the resolution of the first page of the fax (it is theoretically possible for each page in the fax to have a different resolution, but this is rarely the case). If FaxResolution is frNormal, the resolution of the fax is 200x100. If FaxResolution is frHigh, the resolution of the fax is 200x200.

The following example examines the resolution of a fax and sets the scaling properties appropriately:

```
ApdFaxUnpacker1.InFileName := OpenFileDialog.FileName;
ApdFaxUnpacker1.Scaling := False;

{double the height of the fax if it's in 200x100 resolution}
if (ApdFaxUnpacker1.FaxResolution = frNormal) then begin
  ApdFaxUnpacker1.HorizMult := 1;
  ApdFaxUnpacker1.HorizDiv := 1;
  ApdFaxUnpacker1.VertMult := 2;
  ApdFaxUnpacker1.VertDiv := 1;
  ApdFaxUnpacker1.Scaling := True;
end;
```

See also: FaxWidth, InFileName

FaxWidth

read-only, run-time property

```
property FaxWidth : TFaxWidth  
TFaxWidth = (fwNormal, fwWide);
```

↪ The width of the fax.

If the file name specified in `InFileName` is valid, `FaxWidth` is the width of the first page of the fax (it is theoretically possible for each page in the fax to have a different width, but this is rarely the case). If `FaxWidth` is `fwNormal`, the width of the fax is 1728 pixels. If `FaxWidth` is `fwWide`, the width of the fax is 2048 pixels.

The following example examines the width of a fax and allocates a buffer large enough to hold a line of uncompressed fax data:

```
ApdFaxUnpacker1.InFileName := OpenDialog.FileName;  
if (ApdFaxUnpacker1.FaxWidth = fwNormal) then  
  GetMem(Buffer, 1728 div 8)  
else  
  GetMem(Buffer, 2048 div 8);
```

See also: `FaxResolution`, `InFileName`

HorizDiv

property

```
property HorizDiv : Cardinal
```

Default: 1

↪ Determines the horizontal divisor component for scaling.

Attempts to set the value of `HorizDiv` to 0 are ignored.

For a detailed explanation of scaling, see “Scaling” on page 626.

See also: `HorizMult`, `Scaling`, `VertDiv`, `VertMult`

```
property HorizMult : Cardinal
```

Default: 1

↳ Determines the horizontal multiplier component for scaling.

Attempts to set the value of HorizMult to 0 are ignored.

For a detailed explanation of scaling, see “Scaling” on page 626.

See also: HorizDiv, Scaling, VertDiv, VertMult

```
property InFileName : string
```

↳ Specifies the name of the APF file to be unpacked.

The TApdFaxUnpacker reads compressed Group 3 data from a file (in APF format) and decompresses it. The file specified by InFileName must be a valid APF file. If it is not, an EFaxBadFormat exception is raised when the file is accessed.

When InFileName is set, OutFileName is automatically set to the value of InFileName with the extension removed. For instance, if InFileName is set to “C:\MYFAX.APF”, then OutFileName is set to “C:\MYFAX”. Since OutFileName does not contain an extension, the UnpackXxxToXxx routines automatically append an appropriate extension to OutFileName before creating an output image file.

The following example specifies an input file and unpacks it to MYFAX.BMP:

```
ApdFaxUnpacker1.InFileName := 'C:\MYFAX.APF';  
ApdFaxUnpacker1.UnpackFileToBmp;
```

See also: InFileName, OutFileName

```
property NumPages : Cardinal
```

↳ The number of pages in the fax.

If the file specified in `InFileName` is valid, `NumPages` is the number of pages in the fax.

The following example unpacks each page of a fax into individual memory bitmaps and processes them:

```
var
    I : Integer;
    B : TBitmap;

...
ApdFaxUnpacker1.InFileName := OpenDialog.FileName;
for I := 1 to ApdFaxUnpacker1.NumPages do begin
    B := UnpackPageToBitmap(I);
    ...process bitmap image...
    B.Free;
end;
```

See also: `InFileName`

OnOutputLine**event**

```
property OnOutputLine : TUnpackOutputLineEvent
```

```
TUnpackOutputLineEvent = procedure(Sender : TObject;
    Starting, Ending : Boolean; Data : PByteArray;
    Len, PageNum : Integer) of object;
```

↳ Defines an event handler that is called to output a line of decompressed raster data.

As each line of data in an APF file is decompressed, the `OnOutputLine` event handler is called to output the decompressed data. Once the data is passed to `OnOutputLine`, it is discarded to make room for the next line of decompressed data. You can do anything you like with the decompressed data: display it to the screen, write it to a file, etc.

If `Starting` is `True`, no data is passed to the event—it is simply a notification that the unpacking process is beginning. Any resources needed for handling the unpacked data (buffers, output files, etc.) should be allocated.

If `Ending` is `True`, no data is passed to the event—it is simply a notification that the unpacking process is ending. Any resources that were allocated at the beginning of the unpack process should be freed.

Data is a pointer to a zero-based array of bytes that contains the decompressed data. Each byte in the array represents 8 pixels of raster data. The raster data is represented as one-bit-per-pixel. The bits are on for black pixels and off for white pixels. Len contains the length of the data. PageNum is the number of the page that is currently being unpacked.

The following example writes each line of decompressed data to a file:

```
var
    OutFile : File;
...
procedure Form1.ApdFaxUnpackerOutputLine(
    Sender : TObject; Starting, Ending : Boolean;
    Data : PByteArray; Len, PageNum : Integer);
begin
    if Starting then begin
        AssignFile(OutFile, 'C:\MYIMAGE.IMG');
        Rewrite(OutFile, 1);
    end else if Ending then begin
        CloseFile(OutFile);
    end else
        BlockWrite(OutFile, Data^, Len);
end;
```

See also: OnStatus

OnStatus	event
-----------------	--------------

```
property OnStatus : TUnpackStatusEvent
TUnpackStatusEvent = procedure(
    Sender : TObject; FName : string; PageNum : Integer;
    BytesUnpacked, BytesToUnpack : LongInt); of object;
```

✚ Defines an event handler that is called to display the progress of an unpack operation.

The OnStatus event handler is called after each line of a fax is read and decompressed. You can use the information passed to the event to create a status display for the user.

FName is the name of the file that is being unpacked. PageNum is the number of the page that is currently being unpacked.

BytesUnpacked is the number of bytes that have been unpacked so far. BytesToUnpack is the total number of bytes that will be unpacked in the file. You can use these two values to determine the percent completion of the unpack operation.

The following example uses the OnStatus event to display a percent complete to the user:

```
procedure Form1.ApdFaxUnpacker1Status(  
  Sender : TObject; FName : string; PageNum : Integer;  
  BytesUnpacked, BytesToUnpack : LongInt) : Boolean;  
begin  
  Label1.Caption := Format(  
    'Converting page %d of %s, %d%% complete',  
    [PageNum, FName, (BytesUnpacked * 100) div BytesToUnpack];  
end;
```

See also: OnOutputLine

Options

property

```
property Options : TUnpackerOptionsSet  
TUnpackerOptionsSet = Set of TUnpackerOptions;  
TUnpackerOptions = (uoYield, uoAbort);
```

Default: [uoYield]

🔗 A set of optional behaviors for the fax unpacker.

If uoYield is on (the default), the TApdFaxUnpacker yields to Windows regularly, giving other applications a chance to run. If uoYield is not on, the TApdFaxUnpacker hogs the system for the amount of time required to unpack the file. This results in a faster unpack, but is not recommended.

There is one case in which yielding isn't necessary—if your application is 32-bit and your unpack operation is in a separate thread. To turn the yield option off, either set it off in the Object Inspector or use the following code:

```
ApdFaxUnpacker1.Options := [];
```

To turn the yield option on, turn it on in the Object Inspector or use the following code:

```
ApdFaxUnpacker1.Options := [uoYield];
```

To abort a fax unpacking operation at run time, set Options to uoAbort.

property OutFileName : string

↳ Specifies the name of the output image file.

When the TApdFaxUnpacker component creates an image file (BMP, PCX, DCX, or TIF), the image is written to the file specified in OutFileName. You should specify a fully qualified path name, otherwise the file is created in the current directory and in Windows that changes frequently. If the specified file already exists, it is overwritten without warning.

When InFileName is set, OutFileName is automatically set to the value of InFileName with the extension removed. For instance, if InFileName is set to “C:\MYFAX.APF”, then OutFileName is set to “C:\MYFAX”. Since OutFileName does not contain an extension, the UnpackXxxToXxx routines automatically append an appropriate extension to OutFileName before creating an output image file.

The following example demonstrates the use of the OutFileName property:

```
ApdFaxUnpacker1.OutFileName := 'C:\MYFILE.BMP';  
ApdFaxUnpacker1.UnpackFileToBmp;
```

See also: InFileName, UnpackXxxToXxx

Scaling

property

property Scaling : Boolean

↳ Specifies whether image scaling is performed.

If Scaling is True, unpacked images are scaled to a new size determined by the values of four properties. HorizMult and HorizDiv are combined to form a fraction (HorizMult/HorizDiv) that is multiplied by the width of the fax to determine its new width. VertMult and VertDiv are combined to form a fraction (VertMult/VertDiv) that is multiplied by the height of the fax to determine its new height.

For example, assume that a fax is 1728 pixels wide and 2200 pixels tall. If HorizMult equals 1, HorizDiv equals 2, VertMult equals 4, and VertDiv equals 3, the fax would be scaled to 864 pixels wide ($1728 * 1 / 2$) and 2933 pixels high ($2200 * 4 / 3 = 2933$).

See “Scaling” on page 626 for more information.

See also: HorizDiv, HorizMult, VertDiv, VertMult

```
procedure UnpackFile;
```

↳ Unpacks all pages in a fax file.

UnpackFile reads and unpacks every line of the fax file specified by InFileName and passes the unpacked data to the OnOutputLine event handler. If Scaling is True, the data is scaled before it is passed to the OnOutputLine event handler.

The following example unpacks a fax and writes the unpacked data to a file:

```
var
    OutFile : File;

...

procedure Form1.ApdFaxUnpacker1OutputLine(
    Sender : TObject; Starting, Ending : Boolean;
    Data : PByteArray; Len, PageNum : Integer);
begin
    if Starting then begin
        AssignFile(OutFile, 'C:\MYIMAGE.IMG');
        Rewrite(OutFile, 1);
    end else if Ending then begin
        CloseFile(OutFile);
    end else
        BlockWrite(OutFile, Data^, Len);
end;

...

ApdFaxUnpacker1.InFileName := 'C:\MYFAX.APF';
ApdFaxUnpacker1.UnpackFile;
```

See also: InFileName, OnOutputLine, Scaling, UnpackPage

```
function UnpackFileToBitmap : TBitmap;
```

↳ Unpacks all pages in a fax file to a memory bitmap.

UnpackFileToBitmap reads and unpacks every line of the fax file specified by InFileName and puts the unpacked data in a memory bitmap. The memory bitmap is returned by this function in the form of a VCL TBitmap instance. If Scaling is True, the data is scaled before the memory bitmap is created. You are responsible for freeing the bitmap when you are finished with it.

Since the TBitmap class does not include the concept of pages, all pages in the input APF file are placed into the TBitmap in a single image. If the input fax contains 9 pages, those 9 pages are placed into the TBitmap as a single image, one right after the other.

The following example unpacks a fax and puts the unpacked data in a memory bitmap:

```
var
    Bmp : TBitmap;

...

ApdFaxUnpacker1.InFileName := 'C:\MYFAX.APF';
Bmp := ApdFaxUnpacker1.UnpackFileToBitmap;
Form1.Image1.Picture.Bitmap := Bmp;
Bmp.Free;
```

See also: InFileName, Scaling, UnpackPageToBitmap

```
procedure UnpackFileToBmp;
```

↳ Unpacks all pages in a fax file to a Windows bitmap file.

UnpackFileToBmp reads and unpacks every line of the fax file specified by InFileName and writes the unpacked data to a Windows bitmap (BMP) file. The data is written to the file specified by OutFileName. If the file name in OutFileName does not have an extension, a default extension of BMP is appended. If Scaling is True, the data is scaled before it is written to the file.

Since the Windows bitmap file format does not include the concept of pages, all pages in the input APF file are placed into a single image in the BMP file. If the input fax contains 9 pages, those 9 pages are written to a single bitmap, one right after the other.

The following example creates a bitmap file called C:\MYIMAGE.BMP:

```
ApdFaxUnpacker1.InFileName := 'C:\MYFAX.APF';  
ApdFaxUnpacker1.OutFileName := 'C:\MYIMAGE.BMP';  
ApdFaxUnpacker1.UnpackFileToBmp;
```

See also: InFileName, OutFileName, Scaling, UnpackPageToBmp

```
procedure UnpackFileToDcx;
```

↳ Unpacks all pages in a fax file to a DCX file.

UnpackFileToDcx reads and unpacks every line of the fax file specified by InFileName and writes the unpacked data to a DCX file. The data is written to the file specified by OutFileName. If the file name in OutFileName does not have an extension, a default extension of DCX is appended. If Scaling is True, the data is scaled before it is written to the file.

The DCX image file format is a multi-page file format. Each page in the input APF file is placed in a separate page in the DCX file.

The following example creates a DCX file called C:\MYIMAGE.DCX:

```
ApdFaxUnpacker1.InFileName := 'C:\MYFAX.APF';  
ApdFaxUnpacker1.OutFileName := 'C:\MYIMAGE.DCX';  
ApdFaxUnpacker1.UnpackFileToDcx;
```

See also: InFileName, OutFileName, Scaling, UnpackPageToDcx

```
procedure UnpackFileToPcx;
```

↳ Unpacks all pages in a fax file to a PCX file.

UnpackFileToPcx reads and unpacks every line of the fax file specified by InFileName and writes the unpacked data to a PC Paintbrush (PCX) file. The data is written to the file specified by OutFileName. If the file name in OutFileName does not have an extension, a default extension of PCX is appended. If Scaling is True, the data is scaled before it is written to the file.

Since the PCX file format does not include the concept of pages, all pages in the input APF file are placed into a single image in the PCX file. If the input fax contains 9 pages, those 9 pages are output to a single PCX image, one right after the other.

The following example creates a PCX file called C:\MYIMAGE.PCX:

```
ApdFaxUnpacker1.InFileName := 'C:\MYFAX.APF';  
ApdFaxUnpacker1.OutFileName := 'C:\MYIMAGE.PCX';  
ApdFaxUnpacker1.UnpackFileToPcx;
```

See also: InFileName, OutFileName, Scaling, UnpackPageToPcx

```
procedure UnpackFileToTiff;
```

↳ Unpacks all pages in a fax file to a TIFF file.

UnpackFileToTiff reads and unpacks every line of the fax file specified by InFileName and writes the unpacked data to a Tagged Image File Format (TIFF) file. The data is written to the file specified by OutFileName. If the file name in OutFileName does not have an extension, a default extension of TIF is appended. If Scaling is True, the data is scaled before it is written to the file.

Since the TIFF file format does not include the concept of pages, all pages in the input APF file are placed into a single image in the TIFF file. If the input fax contains 9 pages, those 9 pages are written to a single TIFF image, one right after the other.

The following example creates a TIFF file called C:\MYIMAGE.TIF:

```
ApdFaxUnpacker1.InFileName := 'C:\MYFAX.APF';  
ApdFaxUnpacker1.OutFileName := 'C:\MYIMAGE.TIF';  
ApdFaxUnpacker1.UnpackFileToTiff;
```

See also: InFileName, OutFileName, Scaling, UnpackPageToTiff

```
procedure UnpackPage(const Page : Cardinal);
```

↳ Unpacks a single page in a fax file.

UnpackPage reads and unpacks the page specified by Page in the fax file specified by InFileName. The unpacked data is passed to the OnOutputLine event handler. If Page is invalid (i.e., if it is 0 or greater than the total number of pages in the fax file), an EInvalidPageNumber exception is raised. If Scaling is True, the data is scaled before it is passed to the OnOutputLine event handler.

The following example unpacks the first page of a fax and writes the unpacked data to a file:

```
var
    OutFile : File;

...

procedure Form1.ApdxFaxUnpacker1OutputLine(
    Sender : TObject; Starting, Ending : Boolean;
    Data : PByteArray; Len, PageNum : Integer);
begin
    if Starting then begin
        AssignFile(OutFile, 'C:\MYIMAGE.IMG');
        Rewrite(OutFile, 1);
    end else if Ending then begin
        CloseFile(OutFile);
    end else
        BlockWrite(OutFile, Data^, Len);
end;

...

ApdxFaxUnpacker1.InFileName := 'C:\MYFAX.APF';
ApdxFaxUnpacker1.UnpackPage(1);
```

See also: InFileName, OnOutputLine, Scaling, UnpackFile

```
function UnpackPageToBitmap(const Page : Cardinal) : TBitmap;
```

↳ Unpacks a single page in a fax file to a memory bitmap.

UnpackPageToBitmap reads and unpacks the page specified by Page in the fax file specified by InFileName and puts the unpacked data in a memory bitmap. The memory bitmap is returned by this function in the form of a VCL TBitmap instance. If Scaling is True, the data is scaled before the memory bitmap is created. You are responsible for freeing the bitmap when you are finished with it.

The following example unpacks the first page of a fax and puts the unpacked data in a memory bitmap:

```
var
    Bmp : TBitmap;
...

ApdFaxUnpacker1.InFileName := 'C:\MYFAX.APF';
Bmp := ApdFaxUnpacker1.UnpackPageToBitmap(1);
Form1.Imagel.Picture.Bitmap := Bmp;
Bmp.Free;
```

See also: InFileName, Scaling, UnpackFileToBitmap

```
procedure UnpackPageToBmp(const Page : Cardinal);
```

↳ Unpacks a single page in a fax file to a Windows bitmap file.

UnpackPageToBmp reads and unpacks the page specified by Page in the fax file specified by InFileName and writes the unpacked data to a Windows bitmap (BMP) file. The data is written to the file specified by OutFileName. If the file name in OutFileName does not have an extension, a default extension of BMP is appended. If Scaling is True, the data is scaled before it is written to the file.

The following example writes the first page of a fax file to C:\MYIMAGE.BMP:

```
ApdFaxUnpacker1.InFileName := 'C:\MYFAX.APF';
ApdFaxUnpacker1.OutFileName := 'C:\MYIMAGE.BMP';
ApdFaxUnpacker1.UnpackPageToBmp(1);
```

See also: InFileName, OutFileName, Scaling, UnpackFileToBmp

```
procedure UnpackPageToDcx(const Page : Cardinal);
```

↳ Unpacks a single page in a fax file to a DCX file.

UnpackPageToDcx reads and unpacks the page specified by Page in the fax file specified by InFileName and writes the unpacked data to a DCX file. The data is written to the file specified by OutFileName. If the file name in OutFileName does not have an extension, a default extension of DCX is appended. If Scaling is True, the data is scaled before it is written to the file.

The DCX file format is a multi-page file format. Since only one page is unpacked by the UnpackPageToDcx method, the resultant DCX file contains only one page.

The following example writes the first page of a fax file to C:\MYIMAGE.DCX:

```
ApdFaxUnpacker1.InFileName := 'C:\MYFAX.APF';  
ApdFaxUnpacker1.OutFileName := 'C:\MYIMAGE.DCX';  
ApdFaxUnpacker1.UnpackPageToDcx(1);
```

See also: InFileName, OutFileName, Scaling, UnpackFileToDcx

```
procedure UnpackPageToPcx(const Page : Cardinal);
```

↳ Unpacks a single page in a fax file to a PCX file.

UnpackPageToPcx reads and unpacks the page specified by Page in the fax file specified by InFileName and writes the unpacked data to a PC Paintbrush (PCX) file. The data is written to the file specified by OutFileName. If the file name in OutFileName does not have an extension, a default extension of PCX is appended. If Scaling is True, the data is scaled before it is written to the file.

The following example writes the first page of a fax file to C:\MYIMAGE.PCX:

```
ApdFaxUnpacker1.InFileName := 'C:\MYFAX.APF';  
ApdFaxUnpacker1.OutFileName := 'C:\MYIMAGE.PCX';  
ApdFaxUnpacker1.UnpackPageToPcx(1);
```

See also: InFileName, OutFileName, Scaling, UnpackFileToPcx


```
procedure UnpackPageToTiff(const Page : Cardinal);
```

↳ Unpacks a single page in a fax file to a TIFF file.

UnpackPageToTiff reads and unpacks the page specified by Page in the fax file specified by InFileName and writes the unpacked data to a Tagged Image File Format (TIFF) file. The data is written to the file specified by OutFileName. If the file name in OutFileName does not have an extension, a default extension of TIF is appended. If Scaling is True, the data is scaled before it is written to the file.

The following example writes the first page of a fax file to C:\MYIMAGE.TIF:

```
ApdFaxUnpacker1.InFileName := 'C:\MYFAX.APF';  
ApdFaxUnpacker1.OutFileName := 'C:\MYIMAGE.TIF';  
ApdFaxUnpacker1.UnpackPageToTiff(1);
```

See also: InFileName, OutFileName, Scaling, UnpackFileToTiff

VertDiv**property**

```
property VertDiv : Cardinal
```

Default: 1

↳ Determines the vertical divisor component for scaling.

Attempts to set the value of VertDiv to 0 are ignored.

For a detailed explanation of scaling, see “Scaling” on page 626.

See also: HorizDiv, HorizMult, Scaling, VertMult

VertMult**property**

```
property VertMult : Cardinal
```

Default: 1

↳ Determines the vertical multiplier component for scaling.

Attempts to set the value of VertMult to 0 are ignored.

For a detailed explanation of scaling, see “Scaling” on page 626.

See also: HorizDiv, HorizMult, Scaling, VertDiv

property WhitespaceCompression : Boolean

Default: False

↪ Determines whether vertical white space is compressed.

When WhitespaceCompression is True, large blocks of vertical white space are replaced with smaller blocks of vertical white space. If the unpacker encounters WhitespaceFrom or more blank raster lines while unpacking the fax, they are replaced with WhitespaceTo blank lines. For instance, if WhitespaceFrom is 20 and WhitespaceTo is 5, and the unpacker encounters 100 blank lines, only 5 blank lines are written. The same thing happens if only 20 blank lines are encountered. If, however, the unpacker encounters 19 blank lines, those lines remain as is.

By default, compression of vertical white space is not enabled.

See “White space compression” on page 628 for more information.

See also: WhitespaceFrom, WhitespaceTo

property WhitespaceFrom : Cardinal

Default: 0

↪ Specifies the number of consecutive blank lines that are compressed if white space compression is enabled.

When WhitespaceCompression is True, WhitespaceFrom is used to determine the number of consecutive lines of vertical white space that are compressed.

The value of WhitespaceFrom must be greater than the value of WhitespaceTo. If it is not, an EBadArgument exception is raised when an UnpackXxx method is called.

See “White space compression” on page 628 for more information.

See also: WhitespaceCompression, WhitespaceTo

property WhitespaceTo : Cardinal

Default: 0

- ↪ Specifies the number of blank lines that are substituted for every occurrence of WhitespaceFrom or more consecutive blank lines.

When WhitespaceCompression is True, WhitespaceTo is used to determine the number of blank lines that should be written for occurrences of WhitespaceFrom consecutive blank lines.

The value of WhitespaceTo must be less than the value of WhitespaceFrom. If it is not, an EBadArgument exception is raised when any of the UnpackXxx methods are called.

See “White space compression” on page 628 for more information.

See also: WhitespaceCompression, WhitespaceFrom

TApdFaxViewer Component

The TApdFaxViewer component makes it easy to view received faxes (or any APF file). When the FileName property is set to the name of a fax file, the TApdFaxViewer loads the fax into memory, converting each page into a memory bitmap which can then be displayed on the screen. If the viewer is too small to view the fax, scrollbars are provided automatically.

The viewer has capabilities for scaling faxes, so you can implement features like “zoom in and out.” It also provides the ability to rotate faxes, allowing you to adjust for faxes that are sent upside down or sideways. The viewer can compress white space within the fax so that it is more easily viewed. It allows “drag and drop” of fax files onto the viewer. And, you can copy all or part of a fax to the Windows clipboard for use in other applications.

Scaling

The TApdFaxViewer component can scale (i.e., make the size larger or smaller) a fax that is viewed. The fax is scaled only on the screen, not in the input file. This preserves the original size of the fax, while making it more convenient to view. The scaling capabilities can also be used to implement a “Zoom In/Zoom Out” feature of the sort found in many image viewer and editing programs.

If Scaling is set to True (either when the fax is loaded or later), the image displayed on the screen is immediately scaled to the new dimensions. The image is scaled depending on the values of four properties. HorizMult and HorizDiv are combined to form a fraction (HorizMult/HorizDiv) that is multiplied by the width (horizontal aspect) of the fax to determine its new width. VertMult and VertDiv are combined to form a fraction (VertMult/VertDiv) that is multiplied by the height (vertical aspect) of the fax to determine its new height.

For example, assume a standard resolution fax (200x100) is being viewed. When the fax is converted into a memory bitmap (which has an aspect ratio of 1:1) for viewing, the resulting image looks shorter than it should (since, in the original, the height of the pixels was twice as large as width of the pixels). To compensate for this, use the following:

```
ApdFaxViewer1.Scaling      := True;  
ApdFaxViewer1.HorizMult    := 1;  
ApdFaxViewer1.HorizDiv     := 1;  
ApdFaxViewer1.VertMult     := 2;  
ApdFaxViewer1.VertDiv      := 1;
```

This specifies that the viewed fax is scaled to be twice as tall ($\text{VertMult} / \text{VertDiv} = 2/1 = 2$) as it normally would be. This makes the viewed standard resolution fax look normal.

Because changes to the scaling properties cause an immediate repaint of the image displayed on the screen, the above code would actually cause the screen image to be repainted up to five times (and probably with some pretty weird effects). Therefore, when you are changing the scaling properties, you should first turn off the immediate repaint of the display by calling `BeginUpdate`. After you make all the necessary changes to the scaling properties, call `EndUpdate` to repaint the screen. So a better version of the above code is as follows:

```
ApdFaxViewer1.BeginUpdate;
ApdFaxViewer1.Scaling    := True;
ApdFaxViewer1.HorizMult  := 1;
ApdFaxViewer1.HorizDiv   := 1;
ApdFaxViewer1.VertMult   := 2;
ApdFaxViewer1.VertDiv    := 1;
ApdFaxViewer1.EndUpdate;
```

Here is another way to deal with the problem of the standard resolution fax looking shorter than it should:

```
ApdFaxViewer1.BeginUpdate;
ApdFaxViewer1.Scaling    := True;
ApdFaxViewer1.HorizMult  := 1;
ApdFaxViewer1.HorizDiv   := 2;
ApdFaxViewer1.VertMult   := 1;
ApdFaxViewer1.VertDiv    := 1;
ApdFaxViewer1.EndUpdate;
```

This specifies that the width of the fax is to be halved ($\text{HorizMult} / \text{HorizDiv} = 1/2$). This, too, compensates for the difference in aspect ratio between a standard and high resolution fax, but the resulting image is smaller than the one produced in the previous example.

To make it easier to compensate for the aspect ratio of standard resolution faxes, the `AutoScaleMode` property allows you to specify that the scaling should be performed automatically. You can request either method—doubling the height or halving the width.

The `Scaling` property can be used in many ways to produce a nearly unlimited range of images. For instance, you can create an image that is 1/3 the size of the original fax with the following code:

```
ApdFaxViewer1.BeginUpdate;
ApdFaxViewer1.Scaling    := True;
ApdFaxViewer1.HorizMult  := 1;
ApdFaxViewer1.HorizDiv   := 3;
ApdFaxViewer1.VertMult   := 1;
ApdFaxViewer1.VertDiv    := 3;
ApdFaxViewer1.EndUpdate;
```

Rotation

Occasionally a fax is received upside down or sideways, making it difficult to view on the screen. To make it easier to deal with such faxes, the TApdFaxViewer can rotate the fax on the screen.

By default, faxes are displayed as they were received (0 degree rotation). Faxes can be rotated on the screen in 90 degree increments by setting the Rotation property. The TApdFaxViewer updates the screen immediately to reflect the new setting.

The following example demonstrates the use of the Rotation property in viewing upside down faxes:

```
procedure Form1.RotateBtnClick(Sender : TObject);
begin
    ApdFaxViewer1.Rotation := vr180;
end;
```

White space compression

To make it easier to view large faxes that have a lot of white space, the TApdFaxViewer can compress a specified number of blank raster lines into a smaller number of blank raster lines. For example, if the fax contains a header, followed by a lot of blank space, followed by the text of the fax, the white space compression feature will likely allow the entire fax to be viewed on one screen.

To use the white space compression feature, set WhitespaceCompression to True. Every occurrence of WhitespaceFrom or more consecutive blank lines is replaced with WhiteSpaceTo blank lines. For example, if WhitespaceFrom is 20 and WhiteSpaceTo is 5, then any occurrence of 20 or more consecutive blank lines is compressed down to 5 blank lines.

The white space compression feature is active only when a fax is loaded into the viewer. If you need to change the white space compression settings, you must change the appropriate properties and then reload the fax from the file.

Drag and drop

The Windows File Manager program and the folders in Windows 95/98 support file drag and drop. If you click on a file, you can drag it over other windows. If a window is registered using the DragAcceptFiles API call, the file can be dropped on the window. The window can then process (or ignore) the file.

If AcceptDragged is True, the TApdFaxViewer calls the DragAcceptFiles API and allows APF files to be dropped on it. The APF file is automatically loaded into the viewer and the scaling and rotation settings are reset to their defaults.

Your application can be notified of a dropped file if you implement an `OnDropFile` event handler. When a file is dropped on the viewer and the viewer accepts the file, the `OnDropFile` event is called. You can use this notification to update a status display or to change viewer settings to reflect the new file.

Navigation in the viewer

The `TApdFaxViewer` uses keyboard or mouse input to scroll viewed faxes on the screen, change pages within the fax, and select portions of the fax for copying to the clipboard.

Using the mouse to navigate

When the mouse is clicked on a horizontal scrollbar arrow, the display is scrolled right or left by `HorizScroll` pixels (the default is 8). Similarly, when the mouse is clicked on a vertical scrollbar arrow, the display is scrolled up or down by `VertScroll` pixels (the default is 8).

When the mouse is clicked to the left or the right of the scroll thumb on a horizontal scrollbar, the display is scrolled left or right by `HorizScroll*10` pixels. Similarly, when the mouse is clicked above or below the scroll thumb on a vertical scrollbar, the display is scrolled up or down by `VertScroll*10` pixels. The scroll thumb of a horizontal or vertical scrollbar can also be dragged to scroll the display.

Using the keyboard to navigate

The `TApdFaxViewer` recognizes the keystrokes in Table 15.7 for navigation in viewed faxes.

Table 15.7: *TApdFaxViewer* recognized keystrokes

Key	Action
Up arrow	Scroll the display up by <code>VertScroll</code> pixels.
Down arrow	Scroll the display down by <code>VertScroll</code> pixels.
Ctrl+Up arrow	Scroll the display up by <code>VertScroll*10</code> pixels.
Ctrl+Down arrow	Scroll the display down by <code>VertScroll*10</code> pixels.
Left arrow	Scroll the display left by <code>HorizScroll</code> pixels.
Right arrow	Scroll the display right by <code>HorizScroll</code> pixels.
Ctrl+Left arrow	Scroll the display left by <code>HorizScroll*10</code> pixels.
Ctrl+Right arrow	Scroll the display right by <code>HorizScroll*10</code> pixels.
Home	Scroll the display horizontally to the left edge of the fax (i.e., the left edge of the fax is displayed at the left edge of the viewer).

Table 15.7: *TApdFaxViewer* recognized keystrokes (continued)

Key	Action
End	Scroll the display horizontally to the right edge of the fax (i.e., the right edge of the fax is displayed at the right edge of the viewer).
Ctrl+Home	Display the first page of the fax.
Ctrl+End	Display the last page of the fax.
PgDn	Display the next page of the fax.
PgUp	Display the previous page of the fax.
Ctrl+PgDn	Scroll the display vertically to the bottom edge of the fax (i.e., the bottom edge of the fax is displayed at the bottom edge of the viewer).
Ctrl+PgUp	Scroll the display vertically to the top edge of the fax (i.e., the top edge of the fax is displayed at the top edge of the viewer).

Copying a fax to the clipboard

The fax viewer allows you to copy all or part of a viewed fax to the Windows clipboard for later use by other applications. The desired part of the fax must first be selected. Selection can be done in three ways:

1. Call `SelectImage` to select the entire current page.
2. Call `SelectRegion` to select a specified rectangle in the current page.
3. Portions of the current page can be manually selected by clicking on a section of the image and dragging the mouse to form a selection rectangle. If the mouse is dragged outside the borders of the `TApdFaxViewer`, the viewer is automatically scrolled to reflect the position of the mouse.

When you make a selection, it is displayed in reverse video. The selection can then be copied to the Windows clipboard by calling the `CopyToClipboard` method.

Example

The following example demonstrates the steps involved in viewing an APF file. Create a new project, add the following components, and set the property values as indicated in Table 15.8.

Table 15.8: *Example components and property values*

Component	Property	Value
TApdFaxViewer	Align	alClient
TOpenDialog	DefaultExt	APF
	Filter	APF files *.apf All files *.*

Click on the combo box at the top of the Object Inspector and select “Form1.” Next, click Events tab at the bottom of the Object Inspector. From the events page, double-click the “OnCreate” event. A shell for an OnCreate event is generated for you. Modify the generated code to match this:

```
procedure TForm1.FormCreate(Sender : TObject);
begin
    if OpenDialog1.Execute then
        ApdFaxViewer1.FileName := OpenDialog1.FileName
    else
        Halt(1);
end;
```

This event prompts the user to enter the name of an APF file. If a file name is entered (i.e., Execute returns True), the file is opened in the viewer and displayed. If no file name is entered, the program simply terminates.

Hierarchy

TWinControl (VCL)

 TApdCustomFaxViewer (AdFView)

 TApdFaxViewer (AdFView)

Properties

AcceptDragged	HorizMult	Scaling
ActivePage	HorizScroll	VertDiv
AutoScaleMode	LoadWholeFax	VertMult
BGColor	NumPages	VertScroll
BusyCursor	PageBitmaps	WhitespaceCompression
FGColor	PageHeight	WhitespaceFrom
FileName	PageWidth	WhitespaceTo
HorizDiv	Rotation	

Methods

BeginUpdate	FirstPage	PrevPage
CopyToClipboard	LastPage	SelectImage
EndUpdate	NextPage	SelectRegion

Events

OnDropFile	OnPageChange	OnViewerError
------------	--------------	---------------

Reference Section

AcceptDragged

property

property AcceptDragged : Boolean

Default: True

✎ Determines whether files dropped onto the viewer are automatically loaded.

If AcceptDragged is True, the viewer allows APF files to be dragged and dropped on it. The dropped file is automatically loaded into the viewer using the current scaling, white space compression, and rotation settings.

See “Drag and drop” on page 651 for more information.

See also: OnDropFile

ActivePage

run-time property

property ActivePage : Cardinal

✎ The fax page that is currently being viewed.

The TApdFaxViewer displays one page of the fax at a time. ActivePage can be used for a status display (e.g., “Viewing page 3 of 10”) or as an index into the PageBitmaps property to get a TBitmap class for the current page.

ActivePage can also be used to change the page currently being viewed by setting it to the desired page number. If ActivePage is set to an invalid page number (0 or a value larger than the number of pages in the fax), an EInvalidPageNumber exception is raised.

The following example demonstrates how to retrieve the bitmap for the currently viewed page:

```
var
    Bmp : TBitmap;

...
Bmp := ApdFaxViewer1.PageBitmaps[ApdFaxViewer1.ActivePage];
```

See also: PageBitmaps, NumPages

```
property AutoScaleMode : TAutoScaleMode
TAutoScaleMode = (asNone, asDoubleHeight, asHalfWidth);
```

Default: asDoubleHeight

↪ Determines whether standard resolution faxes are automatically scaled.

When a standard resolution fax is loaded into the viewer, it looks shorter than it should. That is because in the fax file each pixel is 1/100th of an inch tall, but just 1/200th of an inch wide.

AutoScaleMode can be used to automatically adjust the width or height of a standard resolution fax so that it looks normal when displayed in the viewer.

AutoScaleMode can contain any of the following values:

Value	Result
asNone	No automatic scaling is performed.
asDoubleHeight	The height of the viewed fax is doubled. If the Scaling property is True, this has the effect of doubling the value of the VertMult property.
asHalfWidth	The width of the viewed fax is halved. If the Scaling property is True, this has the effect of multiplying HorizMult/HorizDiv by 1/2.

See Also: Scaling

```
procedure BeginUpdate;
```

↪ Indicates the beginning of an update of the viewer's scaling properties.

When one of the scaling properties (Scaling, HorizMult, HorizDiv, VertMult, or VertDiv) is modified, the screen is repainted immediately. Since you will usually need to change more than one scaling property, you can use BeginUpdate to temporarily turn off immediate repainting. After you make all the necessary updates to the scaling properties, call EndUpdate to repaint the display.

The following example demonstrates the use of the `BeginUpdate` method by scaling a fax 2-to-1:

```
procedure TForm1.ScaleImage2To1;
begin
  ApdFaxViewer1.BeginUpdate;
  ApdFaxViewer1.Scaling := True;
  ApdFaxViewer1.HorizMult := 2;
  ApdFaxViewer1.HorizDiv := 1;
  ApdFaxViewer1.VertMult := 2;
  ApdFaxViewer1.VertDiv := 1;
  ApdFaxViewer1.EndUpdate;
end;
```

See also: `EndUpdate`, `HorizDiv`, `HorizMult`, `Scaling`, `VertDiv`, `VertMult`

BGColor property

property `BGColor` : `TColor`

Default: `clWhite`

✚ The background color of a displayed fax.

Faxes contain only two colors of pixels: black and white. Normally, when a fax is displayed by the `TApdFaxViewer`, black pixels are displayed in black and white pixels are displayed in white.

`BGColor` changes the display color of white pixels. If `BGColor` is set to `clBlue`, the fax is displayed with black text and images on a blue background.

See also: `FGColor`

BusyCursor property

property `BusyCursor` : `TCursor`

✚ The cursor that is displayed during lengthy operations.

At times, the `TApdFaxViewer` component must load portions of the displayed fax into memory. This can be time-consuming, especially if the fax is large. To let the user know that a lengthy operation is taking place, the `TApdFaxViewer` changes the mouse cursor to the cursor specified by `BusyCursor` during the operation. After the operation is complete, the cursor is changed back to the cursor specified by the `Cursor` property.

The following example tells the fax viewer to display the hourglass cursor during lengthy operations:

```
ApdFaxViewer1.BusyCursor := crHourglass;
```

CopyToClipboard

method

```
procedure CopyToClipboard;
```

↳ Copies the selected image in the viewer to the Windows clipboard.

Fax images copied to the Windows clipboard are stored in cf_Bitmap format. A program using the clipboard treats the fax data like a regular Windows bitmap image. For more information about the clipboard, the cf_Bitmap format, and the SetClipboardData routine, see the Windows API help file.

See “Copying a fax to the clipboard” on page 653 for more information.

See also: SelectImage, SelectRegion

EndUpdate

method

```
procedure EndUpdate;
```

↳ Indicates the end of an update of the viewer’s scaling properties and repaints the screen, reflecting the new settings.

When one of the scaling properties (Scaling, HorizMult, HorizDiv, VertMult, or VertDiv) is modified, the screen is repainted immediately. Since you will usually need to change more than one scaling property, you can use BeginUpdate to temporarily turn off immediate repainting. After you make all the necessary updates to the scaling properties, call EndUpdate to repaint the display.

See also: BeginUpdate, HorizDiv, HorizMult, Scaling, VertDiv, VertMult

FGColor

property

```
property FGColor : TColor
```

Default: clBlack

↳ The foreground color of a displayed fax.

Faxes contain only two colors of pixels: black and white. Normally, when a fax is displayed by the TApdFaxViewer, black pixels are displayed in black and white pixels are displayed in white.

FGColor changes the display color of black pixels. If FGColor is set to clGreen, the fax is displayed with green text and images on a white background.

See also: BGColor

```
property FileName : string
```

↳ The name of the file being viewed.

FileName is the name of the APF file that the fax viewer is viewing. Setting FileName causes the viewer to immediately load the specified fax. Before the fax is loaded, the Scaling and Rotation properties are reset to their defaults.

FileName should be a fully qualified file name. Unqualified file names are assumed to be in the current directory, which changes frequently under Windows.

If LoadWholeFax is False, only the first page of the fax is loaded into memory and displayed. If LoadWholeFax is True, the whole fax is loaded into memory and the first page of the fax is displayed.

If FileName is set to an empty string, the currently loaded fax (if any) is discarded.

The following example demonstrates the use of the FileName property:

```
procedure Form1.OpenItemClick(Sender : TObject);
begin
    if OpenFileDialog.Execute then
        ApdFaxViewer1.FileName := OpenFileDialog.FileName;
end;
```

See also: LoadWholeFax, Rotation, Scaling

```
procedure FirstPage;
```

↳ Displays the first page in the fax.

The TApdFaxViewer displays one page of the fax at a time. Calling FirstPage changes the display to the first page in the fax.

If LoadWholeFax is False, calling this method causes the current page to be discarded and the new page to be loaded into memory. This operation can take some time, depending on the size of the page.

The following example demonstrates the use of `FirstPage`:

```
procedure Form1.FirstPageBtnClick(Sender : TObject);
begin
    {move to first page}
    ApdFaxViewer1.FirstPage;
end;
```

See also: `ActivePage`, `LastPage`, `LoadWholeFax`, `NextPage`, `PrevPage`

HorizDiv property

```
property HorizDiv : Cardinal
```

Default: 1

↪ Determines the horizontal divisor component for scaling.

Attempts to set the value of `HorizDiv` to 0 are ignored. For a detailed explanation of scaling, see “Scaling” on page 626.

See also: `BeginUpdate`, `EndUpdate`, `HorizMult`, `Scaling`, `VertDiv`, `VertMult`

HorizMult property

```
property HorizMult : Cardinal
```

Default: 1

↪ Determines the horizontal multiplier component for scaling.

Attempts to set the value of `HorizMult` to 0 are ignored. For a detailed explanation of scaling, see “Scaling” on page 626.

See also: `BeginUpdate`, `EndUpdate`, `HorizDiv`, `Scaling`, `VertDiv`, `VertMult`

HorizScroll property

```
property HorizScroll : Cardinal
```

Default: 8

↪ Determines the number of pixels that are scrolled during horizontal scrolling.

When the right or left arrow keys are pressed or the right or left arrows on the horizontal scrollbar are clicked, the `TApdFaxViewer` scrolls the display to the left or right. `HorizScroll` determines the number of pixels that are scrolled. The default `HorizScroll` is 8 pixels.

See also: `VertScroll`


```
procedure LastPage;
```

↳ Displays the last page in the fax.

The TApdFaxViewer displays one page of the fax at a time. Calling LastPage changes the display to the last page in the fax.

If LoadWholeFax is False, calling this method causes the current page to be discarded and the new page to be loaded into memory. This operation can take some time, depending on the size of the page.

The following example demonstrates the use of LastPage:

```
procedure Form1.LastPageBtnClick(Sender : TObject);
begin
    {move to last page}
    ApdFaxViewer1.LastPage;
end;
```

See also: ActivePage, FirstPage, LoadWholeFax, NextPage, PrevPage

LoadWholeFax**property**

```
property LoadWholeFax : Boolean
```

Default: False

↳ Determines whether the entire fax is loaded into memory.

The TApdFaxViewer can load faxes one page at a time or all at once. If LoadWholeFax is False (the default), fax pages are loaded into memory as they are needed. If LoadWholeFax is True, all pages of the fax are loaded into memory when the FileName property is set.

Navigating through a fax is slower when LoadWholeFax is False, but it saves a considerable amount of memory, especially if you are viewing large faxes.

See also: FileName

```
procedure NextPage;
```

↳ Displays the next page in the fax.

The TApdFaxViewer displays one page of the fax at a time. Calling NextPage changes the display to the next page in the fax. If the current page is the last page in the fax, calling NextPage has no effect.

If LoadWholeFax is False, calling this method causes the current page to be discarded and the new page to be loaded into memory. This operation can take some time, depending on the size of the page.

The following example demonstrates the use of NextPage:

```
procedure Form1.NextPageBtnClick(Sender : TObject);
begin
    {show next page}
    ApdFaxViewer1.NextPage;
end;
```

See also: ActivePage, FirstPage, LastPage, LoadWholeFax, PrevPage

```
property NumPages : Integer
```

↳ The number of pages in the fax that is currently being viewed.

NumPages can be used as an upper limit when accessing the PageBitmaps property, or in a status display (e.g., "Viewing page 1 of 3").

The following example performs an operation on the bitmap for each page in the fax:

```
procedure Form1.ProcessPages;
var
    I    : Integer;
    Bmp  : TBitmap;
begin
    for I := 1 to ApdFaxViewer1.NumPages do begin
        Bmp := ApdFaxViewer1.PageBitmaps[I];
        ...process the bitmap...
        Bmp.Free;
    end;
end;
```

See also: ActivePage, PageBitmaps

```
property OnDropFile : TViewerFileDropEvent  
TViewerFileDropEvent = procedure(  
    Sender : TObject; FileName : string) of object;
```

↳ Defines an event handler that is called when a file is dropped on the viewer.

When a file is dropped onto a TApdFaxViewer and AcceptDragged is True, the file is loaded into the viewer and the OnDropFile event is called. This notification is useful for updating status displays or for viewing sessions in which the bitmaps for each page are being manually processed and you need notification of changes in the viewer.

The following example demonstrates the use of the OnDropFile event:

```
procedure Form1.ApdxViewer1DropFile(  
    Sender : TObject; FileName : string);  
var  
    I      : Integer;  
    Bmp    : TBitmap;  
begin  
    Label1.Caption := Format('Now viewing %s, page 1 of %d',  
        [FileName, ApdxViewer1.NumPages]);  
    {process bitmaps}  
    for I := 1 to ApdxViewer1.NumPages do begin  
        Bmp := ApdxViewer1.PageBitmaps[I];  
        ...process bitmap for new fax...  
        Bmp.Free;  
    end;  
end;
```

See also: AcceptDragged

```
property OnPageChange : TNotifyEvent
```

↳ Defines an event handler that is called when the active page changes.

The user of the TApdFaxViewer component can change pages by pressing the PgUp and PgDn keys. When that happens, the TApdFaxViewer component notifies you that the active page is changing by calling the OnPageChange event.

The following example updates a label on a form to display information about the current page:

```
procedure TMainForm.ApdxViewer1PageChange(Sender : TObject);
begin
  if (ApdxViewer1.FileName <> '') then
    StatusPanel.Caption :=
      Format(' Viewing page %d of %d in %s',
        [ApdxViewer1.ActivePage, ApdxViewer1.NumPages,
        ApdxViewer1.FileName])
  else
    StatusPanel.Caption := ' No file loaded';
end;
```

OnViewerError

```
property OnViewerError : TViewerErrorEvent
```

```
TViewerErrorEvent = procedure(
  Sender : TObject; ErrorCode : Integer) of object;
```

↳ Defines an event handler that reports fax viewer errors.

If LoadWholeFax is False, the TApdFaxViewer is sometimes forced to load faxes in the background. When this happens, the component cannot raise exceptions, so the OnViewerError event is called to notify you of an error.

ErrorCode contains the number of the error that occurred. It can be any of the ecXxx error codes.

See also: LoadWholeFax

```
property PageBitmaps [const Index : Integer] : TBitmap
```

↳ An indexed property containing TBitmap representations of the pages in a viewed fax.

PageBitmaps is used to obtain a bitmap representation of the fax loaded by the viewer. Each element of the PageBitmaps indexed property represents a page in the fax. The first element of the property is a representation of the first page of the fax. If an attempt is made to access an invalid page (a page number less than or equal to zero, or greater than the number of pages in the fax), an EInvalidPageNumber exception is raised.

When an element of the PageBitmaps property is accessed, a copy of the bitmap for the specified page is made and that copy is the value of the property. You must free bitmaps that you get from the PageBitmaps property when you are finished with them.

If LoadWholeFax is False, referencing any element of the PageBitmaps array (other than the current page) causes the referenced page to be loaded into memory. This operation can take some time, depending on the size of the page.

The following example obtains bitmaps for all pages in a fax and processes them:

```
var
    I      : Integer;
    Bmp    : TBitmap;
...
for I := 1 to ApdFaxViewer1.NumPages do begin
    Bmp := ApdFaxViewer1.PageBitmaps[I];
    ...process the bitmap
    Bmp.Free;
end;
```

See also: LoadWholeFax

PageHeight

read-only, run-time property

```
property PageHeight : Cardinal
```

↳ The height (in pixels) of the currently viewed page.

When selecting regions of faxes and processing page bitmaps, it is useful to know the dimensions of the page currently being viewed. PageHeight is the height (in pixels) of the current page. If no fax is currently loaded into the viewer, the value of PageHeight is 0.

The following example uses the PageHeight property to select the top half of a fax page:

```
var
  R : TRect;
...
R.Top    := 0;
R.Left   := 0;
R.Bottom := (ApdFaxViewer1.PageHeight div 2) - 1;
R.Right  := ApdFaxViewer1.PageWidth - 1;
ApdFaxViewer1.SelectRegion(R);
```

See also: PageBitmaps, PageWidth

PageWidth

read-only, run-time property

```
property PageWidth : Cardinal
```

↳ The width (in pixels) of the currently viewed page.

When selecting regions of faxes and processing page bitmaps, it is useful to know the dimensions of the page currently being viewed. PageWidth is the width (in pixels) of the current page. If no fax is currently loaded into the viewer, the value of PageWidth is 0.

See also: PageHeight

PrevPage

method

```
procedure PrevPage;
```

↳ Displays the previous page in the fax.

The TApdFaxViewer displays one page of the fax at a time. Calling PrevPage changes the display to the previous page in the fax. If the current page is the first page in the fax, calling PrevPage has no effect.

If LoadWholeFax is False, calling this method causes the current page to be discarded and the new page to be loaded into memory. This operation can take some time, depending on the size of the page.

The following example demonstrates the use of `PrevPage`:

```
procedure Form1.ShowPreviousBtnClick(Sender : TObject);
begin
    {show previous page}
    ApdFaxViewer1.PrevPage;
end;
```

See also: `ActivePage`, `FirstPage`, `LastPage`, `LoadWholeFax`, `NextPage`

Rotation

property

```
property Rotation : TViewerRotation
TVviewerRotation = (vr0, vr90, vr180, vr270);
```

Default: `vr0`

↪ The angle at which a fax is viewed.

When you set `Rotation`, the bitmap image of the fax is rotated in memory and the rotated image is displayed on the screen. Rotation is performed in 90 degree increments. The possible values for `Rotation` are:

Value	Result
<code>vr0</code>	The fax is displayed as it was converted or received.
<code>vr90</code>	The fax is rotated 90 degrees to the right.
<code>vr180</code>	The fax is rotated 180 degrees to the right (i.e., turned upside down).
<code>vr270</code>	The fax is rotated 270 degrees to the right (or 90 degrees to the left, depending on your viewpoint).

By default, faxes are displayed as they were received (0 degree rotation, or `vr0`).

For more information see “Rotation” on page 651.

See also: `Scaling`

property Scaling : Boolean

Default: False

☞ Specifies whether image scaling is performed on viewed faxes.

If Scaling is True, faxes viewed in a TApdFaxViewer are scaled to a new size determined by the values of four properties. HorizMult and HorizDiv are combined to form a fraction (HorizMult/HorizDiv) that is then multiplied by the width of the fax to determine its new width. VertMult and VertDiv are combined to form a fraction (VertMult/VertDiv) that is multiplied by the height of the fax to determine its new height.

For example, assume that a fax is 1728 pixels wide and 2200 pixels tall. If HorizMult equals 1, HorizDiv equals 2, VertMult equals 4, and VertDiv equals 3, the fax would be scaled to 864 pixels wide ($1728 * 1 / 2$) and 2933 pixels high ($2200 * 4 / 3 = 2933$).

When Scaling is changed, the display is changed immediately to reflect the new settings. If you need to prevent this, the BeginUpdate and EndUpdate methods can be called to allow all scaling settings to be set before the display is updated.

See “Scaling” on page 649 for more information.

See also: BeginUpdate, EndUpdate, HorizDiv, HorizMult, VertDiv, VertMult

SelectImage

method

```
procedure SelectImage;
```

☞ Selects the entire current page.

SelectImage selects the entire page that is currently being viewed. When SelectImage is called, the screen is immediately updated to reflect the selection, causing the entire page to be shown in reverse video. The selection can be copied to the Windows clipboard by calling CopyToClipboard. See “Copying a fax to the clipboard” on page 653 for more information.

The following example demonstrates the use of SelectImage:

```
procedure Form1.CopyBtnClick(Sender : TObject);
begin
    ApdFaxViewer1.SelectImage;
    ApdFaxViewer1.CopyToClipboard;
end;
```

See also: CopyToClipboard, SelectRegion


```
procedure SelectRegion(const R : TRect);
```

✚ Selects the image bounded by the specified rectangle.

SelectRegion selects a portion of the page being viewed. R specifies the rectangle that is to be selected. When SelectRegion is called, the screen is immediately updated to reflect the selection, causing the rectangle specified by R to be displayed in reverse video. The selection can be copied to the Windows clipboard by calling CopyToClipboard. See “Copying a fax to the clipboard” on page 653 for more information.

The following example selects a rectangle of 10 pixels by 10 pixels in the upper left corner of the page:

```
var
    R : TRect;
...
R.Top      := 0;
R.Left     := 0;
R.Bottom   := 9;
R.Right    := 9;
ApdFaxViewer1.SelectRegion(R);
```

See also: CopyToClipboard, SelectImage

```
property VertDiv : Cardinal
```

Default: 1

✚ Determines the vertical divisor component for scaling.

Attempts to set the value of VertDiv to 0 are ignored.

For a detailed explanation of scaling, see “Scaling” on page 626.

See also: BeginUpdate, EndUpdate, HorizDiv, HorizMult, Scaling, VertMult

VertMult

property

`property VertMult : Cardinal`

Default: 1

↪ Determines the vertical multiplier component for scaling.

Attempts to set the value of VertMult to 0 are ignored.

For a detailed explanation of scaling, see “Scaling” on page 626.

See also: BeginUpdate, EndUpdate, HorizDiv, HorizMult, Scaling, VertDiv

VertScroll

property

`property VertScroll : Cardinal`

Default: 8

↪ Determines the number of pixels that are scrolled during vertical scrolling.

When the up or down arrow keys are pressed or the up or down arrows on the vertical scrollbar are clicked, the TApdFaxViewer scrolls the display up or down. VertScroll determines the number of pixels that are scrolled. The default VertScroll is 8 pixels.

See also: HorizScroll

WhitespaceCompression

property

`property WhitespaceCompression : Boolean`

Default: False

↪ Determines whether vertical white space is compressed.

When WhitespaceCompression is True, large blocks of vertical white space are replaced with smaller blocks of vertical white space.

If the viewer encounters WhitespaceFrom or more blank raster lines, they are replaced with WhitespaceTo blank lines. For example, if WhitespaceFrom is 20 and WhitespaceTo is 5 and the viewer encounters 100 blank lines, only 5 blank lines are displayed. The same thing happens if only 20 blank lines are encountered. If, however, the viewer encounters 19 blank lines, those lines are all displayed.

By default, compression of vertical white space is not enabled.

Compression of white space takes place only when a fax is loaded. If you need to change the white space compression settings, you must change the appropriate properties and then reload the fax from the file.

You can force the reload of a fax using the following code:

```
procedure Form1.ReloadFax;
var
    SavePage : Cardinal;
    SaveFile : string;
begin
    SavePage := ApdFaxViewer1.ActivePage;
    SaveFile := ApdFaxViewer1.FileName;

    {discard the current fax}
    ApdFaxViewer1.FileName := '';

    {reload the fax}
    ApdFaxViewer1.FileName := SaveFile;
    ApdFaxViewer1.ActivePage := SavePage;
end;
```

See also: `WhitespaceFrom`, `WhitespaceTo`

WhitespaceFrom

property

property `WhitespaceFrom` : Cardinal

Default: 0

↳ Specifies the number of consecutive blank lines that are compressed if white space compression is enabled.

When `WhitespaceCompression` is `True`, `WhitespaceFrom` is used to determine the number of consecutive lines of vertical white space that are compressed.

The value of `WhitespaceFrom` must be greater than the value of `WhitespaceTo`. If it is not, an `EBadArgument` exception is raised when an attempt is made to load a fax file into the viewer.

See also: `WhitespaceCompression`, `WhitespaceTo`

property WhitespaceTo : Cardinal

Default: 0

- ↪ Specifies the number of blank lines that are substituted for every occurrence of WhitespaceFrom or more consecutive blank lines.

When WhitespaceCompression is True, WhitespaceTo is used to determine the number of blank lines that should be displayed for occurrences of WhitespaceFrom consecutive blank lines.

The value of WhitespaceTo must be less than the value of WhitespaceFrom. If it is not, an EBadArgument exception is raised when an attempt is made to load a fax file into the viewer.

See also: WhitespaceCompression, WhitespaceFrom

TApdFaxPrinter Component

The TApdFaxPrinter provides services for printing fax files to any Windows printer. Header and Footer properties allow you to put text at the top or bottom of each page. You can scale each fax page to fit on the requested paper size and you can print multiple fax pages on a single sheet of paper.

The TApdFaxPrinterStatus component provides a standard display for monitoring the print progress, or you can use a status event handler to notify your program of status events. The TApdFaxPrinterLog component provides automatic logging of the success or failure of a print job, or you can intercept the log events with an event handler to provide your own logging.

Headers and footers

The TApdFaxPrinter can add headers and footers to each printed fax. This can be useful when the information on the received fax is not enough. For example, not all faxes provide the time sent/received or the station ID of the caller.

The headers and footers are configured separately, so you can use one, both, or neither. You can separately select the font for headers and footers. A class called TApdFaxPrintMargin publishes three properties of the header or footer: Caption, Enabled, and Font. Caption specifies the text to be printed. For example, to print “This is my fax header” at the top of each page, use:

```
ApdFaxPrinter1.FaxHeader.Caption := 'This is my fax header';
```

The Caption property supports replacement tags. A replacement tag is one of several characters prefixed with ‘\$’. When the header or footer is printed, the tags are replaced with appropriate text. A header or footer can consist of any mix of tags and normal text (be careful that your normal text doesn’t happen to contain tags, though). The available replacement tags are shown in Table 15.9.

Table 15.9: *Replacement tags*

Tag	Description
\$D	Today’s date in MM/DD/YY format, always 8 characters.
\$N	Total number of pages, variable length.
\$P	Current page number, variable length.
\$F	Fax file name, variable length.
\$T	Current time in HH:MMpm format, always 7 characters.

Note that some of the tags vary in length. For example, \$P would be replaced by '1' for the first page and "10" for the tenth page.

The default footer caption is:

```
DefFaxFooterCaption = 'PAGE: $P of $N';
```

This prints a footer on the first page of a six page fax as:

```
PAGE: 1 of 6
```

The Enabled property of headers and footers simply turns printing on or off. Enabled is checked for each page printed to see if a header or footer should be printed. You can use the OnNextPage event to turn the printing of headers and footers on and off on a page-by-page basis. For example, you could print headers only on even-numbered pages and footers only on odd-numbered pages.

The Font property is a TFont class and allows you to change the selected font. The default font is MS Sans Serif.

No size adjustments are made in the header or footer for captions that are larger than the page width. The header and footer text is printed exactly as specified in the Caption property.

Scaling

When a fax is received or created, it is usually stored in the same size as the original document. If you receive a fax that was originally on legal-sized paper, printing it on letter-sized paper causes three inches of each page to overflow to the next page. If you set the PrintScale property to psFitToPage, each page is scaled to fit on the requested paper size.

Fax printer events

The fax printer component generates these events.

OnFaxPrintLog

```
procedure(Sender: TObject; FaxPLCode: TFaxPLCode) of object;
```

Generated at the start and end of each printed fax. This provides an opportunity to log the status of each fax printed.

OnFaxPrintStatus

```
procedure(  
    Sender: TObject; StatusCode: TFaxPrintProgress) of object;
```

Generated at semi-regular intervals so that a program can display the progress of the printing fax.

OnNextPage

```
procedure(Sender: TObject; CP, LP: Word) of object;
```

Generated at the start of each page. Programs can intercept this event to change the settings of certain options on a per page basis.

Fax printer status

Printing a fax file can take from several seconds to several minutes, depending on numerous factors, including the size of the fax file and the number of pages to print. To allow you to give the user an indication of the progress of the print session, the fax printer frequently generates an OnFaxPrintStatus event.

The following example handles the OnFaxPrintStatus event:

```
TForm1 = class(TForm)
    ....
    File   : TLabel;
    Page   : TLabel;
    Status : TLabel;
    FP     : TApdFaxPrinter;
    ...
end;

procedure TForm1.ApFdFaxPrintStatus(
    Sender : TObject; StatusCode : TFaxPrintProgress);
const
    ProgressSt: array[TFaxPrintProgress] of string[10] =
        ('Idle', 'Converting', 'Composing',
         'Rendering', 'Submitting');
begin
    File.Caption := FP.FileName;
    Status.Caption := ProgressSt[StatusCode];
    Page.Caption := IntToStr(FP.CurrentPrintingPage);
end;
```

The method named TApdFaxPrinterStatus handles the OnFaxPrintStatus event by updating a form at each call. StatusCode specifies the current state of the print job.

Fax printer logging

It is often desirable to automate the printing of faxes. For example, a fax server might send and receive faxes during the night and automatically print received faxes at designated times. In this case it would be nice to keep a record of the faxes that were successfully printed and those that weren't. The printer logging feature provides the opportunity to log information about each printed fax.

To support logging, the fax printer generates an `OnFaxPrintLog` event at the start and end of each fax printed. The event passes a parameter that identifies the current log action. The following is an example of a simple log event:

```
procedure TForm1.ApdFaxPrintLog(  
    Sender : TObject; FaxPLCode : TFaxPLCode);  
begin  
    case FaxPLCode of  
        lcStart :  
            CurrentFile.Caption := ApdFaxPrinter1.FileName;  
        lcFinish:  
            PrintOK.Items.Add(ApdFaxPrinter1.FileName);  
        lcAborted:  
            PrintAborted.Items.Add(ApdFaxPrinter1.FileName);  
        lcFailed:  
            PrintFailed.Items.Add(ApdFaxPrinter1.FileName);  
    end;  
end;
```

This example shows every possible log value. It uses a `TLabel` component named `CurrentFile` to display the current file being printed and updates one of three `TListBox` components with the final status of the print session.

The printer logging routine is not limited to just writing status information. It can also be used to take care of cleanup duties after a fax is printed. For example, you could conserve disk space by automatically archiving a received fax after it is printed.

A supplied component can do automatic print logging for you. If you create an instance of a `TApdFaxPrinterLog` and assign it to `FaxPrinterLog`, logging is done automatically. For each `OnFaxPrintLog` event, the fax printer calls the `UpdateLog` method of `TApdFaxPrinterLog` to write the information to the log file. It then calls the `OnFaxPrintLog` event.

Example

This example shows how to construct and use a fax printer component. Create a new project, add the following components, and set the property values as indicated in Table 15.10.

Table 15.10: *Fax printer component example*

Component	Property	Value
TApdFaxPrinter		
TApdFaxPrinterStatus		
TOpenDialog		
TButton	Caption	Select File
	Name	FileNameButton
TButton	Caption	Print
	Name	PrintButton
TLabel	Caption	File Name
	Name	fnLabel
TEdit	Name	FileNameEdit

Double click on the FileNameEdit's OnChange event handler within the Object Inspector and modify the generated method to match the following code:

```
procedure TForm1.FileNameEditChange(Sender : TObject);
begin
    ApdFaxPrinter1.FileName := FileNameEdit.Text;
end;
```

This event handler updates the name of the file to be printed whenever the text in the edit control changes.

Double click on the FileNameButton's OnClick event handler and modify the generated method to match the following code:

```
procedure TForm1.FileNameButtonClick(Sender : TObject);
begin
    OpenDialog1.Filter := 'APF Files (*.APF)|*.APF';
    if OpenDialog1.Execute then
        FileNameEdit.Text := OpenDialog1.FileName;
end;
```

This event handler updates the file name edit box with the selected file from the OpenFileDialog call.

Double click on the PrintButton's OnClick event handler and change the generated method to match the following code:

```
procedure TForm1.PrintButtonClick(Sender : TObject);
begin
    ApdFaxPrinter1.PrintFax;
end;
```

This tells the TApdFaxPrinter to begin printing the fax file specified by the FileName.

Compile and run the example. Click Select File to select the fax file to print.

This example is in the EXFPRN1 project in the \ASYNCPRO\EXAMPLES directory.

Hierarchy

TComponent (VCL)

❶ TApdBaseComponent (OOMisc)	8
TApdCustomFaxPrinter (AdFaxPrn)	
TApdFaxPrinter (AdFaxPrn)	

Properties

Caption	FaxWidth	PrintScale
CurrentPrintingPage	FileName	StatusDisplay
FaxFooter	FirstPageToPrint	TotalFaxPages
FaxHeader	LastPageToPrint	❶ Version
FaxPrinterLog	MultiPage	
FaxResolution	PrintProgress	

Methods

PrintAbort	PrintFax	PrintSetup
------------	----------	------------

Events

OnFaxPrintLog	OnFaxPrintStatus	OnNextPage
---------------	------------------	------------

Reference Section

Caption

property

```
property Caption : string
```

Default: “APro Fax Printer”

↳ Used by Windows in the Print Manager and for network title pages.

Each document submitted to the Windows Print Manager has an associated name to identify it in the print queue. Caption allows you to specify the name.

CurrentPrintingPage

read-only, run-time property

```
property CurrentPrintingPage : Word
```

↳ The number of the pages currently being printed.

As each page is printed, CurrentPrintingPage is updated to reflect the current page number in the fax.

FaxFooter

property

```
property FaxFooter : TApdFaxPrinterMargin
```

```
TApdFaxPrinterMargin = class(TApdCustomFaxPrinterMargin)
```

```
published
```

```
    property Caption;
```

```
    property Enabled;
```

```
    property Font;
```

```
end;
```

Default: Caption: “PAGE: \$P of \$N”, Enabled: True

↳ Specifies the options for the fax page footer.

Caption specifies the text of the footer. It can consist of normal text and replacement tags. A replacement tag is one of several characters prefixed with ‘\$’. When the footer is printed, the tags are replaced with appropriate text. The available replacement tags are listed in “Headers and footers” on page 674.

Enabled indicates whether footers should be printed. Font is the font for the footer.

See “Headers and footers” on page 674 for more information.

See also: FaxHeader

```
property FaxHeader : TApdFaxPrinterMargin
TApdFaxPrinterMargin = class(TApdCustomFaxPrinterMargin)
published
    property Caption;
    property Enabled;
    property Font;
end;
```

Default: Caption: “FILE: \$F”, Enabled: True

↪ Specifies the options for the fax page header.

Caption specifies the text of the header. It can consist of normal text and replacement tags. A replacement tag is one of several characters prefixed with ‘\$’. When the header is printed, the tags are replaced with appropriate text. The available replacement tags are listed in “Headers and footers” on page 674.

Enabled indicates whether headers should be printed. Font is the font for the header.

See “Headers and footers” on page 674 for more information.

See also: FaxFooter

```
property FaxPrinterLog : TApdFaxPrinterLog
```

↪ An instance of a printer logging component.

If FaxPrinterLog is nil (the default), the fax printer does not perform automatic logging. You can install an OnFaxPrintLog event handler to perform logging in this case.

If you create an instance of (or a descendant of) a TApdFaxPrinterLog (see page 695) and assign it to FaxPrinterLog, logging is done automatically.

See “Fax printer logging” on page 677 for more information.

See also: OnFaxPrintLog

FaxResolution

read-only, run-time property

```
property FaxResolution : TFaxResolution
```

```
TFaxResolution = (frNormal, frHigh);
```

Default: frNormal

↳ Specifies the resolution of the fax.

FaxResolution is the resolution of the fax specified by FileName. The resolution of the fax is used internally to scale the fax for printing. If FaxResolution is frNormal, the resolution of the fax is 200x100. If FaxResolution is frHigh, the resolution of the fax is 200x200.

FaxWidth

read-only, run-time property

```
property FaxWidth : TFaxWidth
```

```
TFaxWidth = (fwNormal, fwWide);
```

Default: fwNormal

↳ Specifies the width of the fax.

FaxWidth is the width (in pixels) of the fax specified by FileName. It is used internally to scale the fax for printing. If FaxWidth is fwNormal, the width of the fax is 1728 pixels. If FaxWidth is fwWide, the width of the fax is 2048 pixels.

FileName

property

```
property FileName : string
```

↳ The name of the fax file to print.

The FileName property must be set to a valid APF file name before PrintSetup or PrintFax are called. If no FileName is specified and PrintFax is called, no printing occurs. If no FileName is specified and PrintSetup is called, you can select the printer device, but the page numbers to print will be invalid.

See also: PrintFax, PrintSetup

FirstPageToPrint

run-time property

property FirstPageToPrint : Word

↳ Specifies the first page to be printed in the fax.

FirstPageToPrint specifies the first page to print in the fax specified by FileName. This is usually set by the PrintSetup method by selecting the starting and ending pages to print.

See also: PrintSetup

LastPageToPrint

run-time property

property LastPageToPrint : Word

↳ Specifies the last page to be printed in the fax.

LastPageToPrint specifies the last page to print in the fax specified by FileName. This is usually set by the PrintSetup method by selecting the starting and ending pages to print.

MultiPage

property

property MultiPage : Boolean

Default: False

↳ Determines the number of fax pages that are printed on each printed page.

If MultiPage is False (the default), each fax page is printed on one printed page.

If MultiPage is True, multiple fax pages are printed on each printed page. If printing is in Portrait mode, 4 fax pages are printed on each printed page. If printing is in Landscape mode, 2 fax pages are printed on each printed page.

Any value of PrintScale can be used with MultiPage.

See also: PrintScale

```
property OnFaxPrintLog : TFaxPLevent

TFaxPLevent = procedure(
    Sender : TObject; FaxPLCode : TFaxPLCode) of object;

TFaxPLCode = (lcStart, lcFinish, lcAborted, lcFailed);
```

↳ Defines an event handler that is called at designated points during a fax printing session.

The primary purpose of OnFaxPrintLog is to give applications a chance to log statistical information about a fax print session.

Sender is the fax printer component to be logged. FaxPLCode indicates the state of the print job. The possible states are:

State	Meaning
lcStart	Printing started.
lcFinish	Printing finished successfully.
lcAborted	Printing aborted before completion.
lcFailed	Printing failed.

No other information is passed with the event. You can use fax printer status properties such as FileName and CurrentPrintingPage to get additional information about the print job.

See “Fax printer logging” on page 677 for more information.

See also: FaxPrinterLog

```
property OnFaxPrintStatus : TFaxPrintStatusEvent

TFaxPrintStatusEvent = procedure(
    Sender : TObject; StatusCode : TFaxPrintProgress) of object;

TFaxPrintProgress = (
    ppIdle, ppConverting, ppComposing, ppRendering, ppSubmitting);
```

↳ Defines an event handler that is called regularly during a printing session.

This event is generated for each action that the print component performs. You can use it to update a status display that informs the user about the fax printing progress.

Sender is the fax printer component that is in progress. StatusCode indicates the status of the print job. The possible values are:

Value	Meaning
ppIdle	Nothing is happening.
ppConverting	The current page is being converted from APF format.
ppComposing	The current page is being composed for printing.
ppRendering	The current page is being rendered.
ppSubmitting	The current page is being submitted to the printer.

No other information is passed with the event. You can use fax printer status properties such as FileName and CurrentPrintingPage to get additional information about the print job.

See also: StatusDisplay

OnNextPage

event

```
property OnNextPage : TFaxPrnNextPageEvent
TFaxPrnNextPageEvent = procedure(
    Sender : TObject; CP, TP : Word) of object;
```

↳ Defines an event handler that is called before each page is printed.

This event is generated for each page in the fax before it is printed. You can use OnNextPage, for example, to abort the printing of a fax by calling PrintAbort.

CP is the current page number of the fax that is printing. TP is the total number of pages to be printed.

See also: PrintAbort

PrintAbort

method

```
procedure PrintAbort;
```

↳ Cancels printing of a fax.

PrintAbort can be called whenever a fax is in the process of being rendered or submitted to the printer. It halts the sending of data to Print Manager. Any data already sent to Print Manager will still be printed.

Calls to PrintAbort during other states are ignored.

See also: PrintFax


```
procedure PrintFax;
```

↳ Prints the fax.

PrintFax prints the fax specified by FileName. It is usually called after PrintSetup is called to select the printer and the range of pages to print. Printing can be aborted by calling PrintAbort.

See also: PrintAbort, PrintSetup

```
property PrintProgress : TFaxPrintProgress
```

```
TFaxPrintProgress = (  
    ppIdle, ppConverting, ppComposing, ppRendering, ppSubmitting);
```

↳ Indicates the progress of the print job.

PrintProgress contains the current state of the TApdFaxPrinter during a print job. You can use this property to track the fax progress in your status routines. The possible values for PrintProgress are:

Value	Meaning
ppIdle	Nothing is happening.
ppConverting	The current page is being converted from APF format.
ppComposing	The current page is being composed for printing.
ppRendering	The current page is being rendered.
ppSubmitting	The current page is being submitted to the printer.

PrintScale

property

```
property PrintScale : TFaxPrintScale  
TFaxPrintScale = (psNone, psFitToPage);
```

Default: psFitToPage

↳ Specifies how each page of the fax is scaled for printing.

If PrintScale equals psFitToPage (the default), the fax is scaled so that each fax page fits on a single printed page. If PrintScale equals psNone, no scaling is done. The fax is printed as it appears in the fax document.

To print multiple fax pages on each printed page, use MultiPage.

See also: MultiPage

PrintSetup

method

```
procedure PrintSetup;
```

↳ Sets the options for the printer.

When PrintSetup is called, the Windows common printer setup dialog is displayed. You can then specify options such as the printer to use, the page range, and so on.

See also: PrintFax

StatusDisplay

property

```
property StatusDisplay : TApdAbstractFaxPrinterStatus
```

↳ An instance of a fax printer status window.

If StatusDisplay is nil (the default), the fax printer does not provide an automatic status window. You can install an OnFaxPrintStatus event handler to display the status in this case.

If you create an instance of a class derived from TApdAbstractFaxPrinterStatus or use the supplied TApdFaxPrinterStatus component (see page 693) and assign it to StatusDisplay, the status window is displayed and updated automatically.

See also: OnFaxPrintStatus

property TotalFaxPages : Word

↳ The total number of pages in the fax file.

TotalFaxPages is the total number of pages in the fax file specified by FileName. It is not the number of pages to be printed, since that is controlled by FirstPageToPrint and LastPageToPrint. The number of pages to print is calculated by:

$$\text{PagesToPrint} := (\text{LastPageToPrint} - \text{FirstPageToPrint}) + 1$$

TApdAbstractFaxPrinterStatus Class

TApdAbstractFaxPrinterStatus is an abstract class that defines the methods and properties needed by a component that automatically displays status while a TApdFaxPrinter component is printing a fax. You generally won't need to create a descendent class of your own, since Async Professional supplies one, the TApdFaxPrinterStatus component described on page 693.

However, TApdFaxPrinterStatus shows a particular set of information about a print job in a predefined format. If this format is not suitable for your needs, you can create your own descendant of TApdAbstractFaxPrinterStatus. The best way to start is to study the source code of TApdFaxPrinterStatus (in the AdFPStat unit) and its associated form, TStandardFaxPrinterStatusDisplay.

The TApdAbstractFaxPrinterStatus class contains an instance of a TForm that holds controls used to display the printing status. You design this form, create an instance, and assign the instance to the Display property of TApdAbstractFaxPrinterStatus.

TApdAbstractFaxPrinterStatus replaces the standard VCL properties Caption, Ctl3D, Position, and Visible and the standard VCL method Show. When these routines are used in the status component, the overridden versions perform the same actions on the associated Display form. Thus you can display the status form by calling Show, erase it by setting Visible to False, adjust its position by assigning to Position, and use 3D effects by setting Ctl3D to True.

Once you create an instance of your TApdAbstractFaxPrinterStatus descendant, you must assign it to the StatusDisplay property of your TApdFaxPrinter component. When the fax printer needs to update the status display, it calls the UpdateDisplay method of TApdAbstractFaxPrinterStatus, which you must override to update your status window.

Hierarchy

TComponent (VCL)

- TApdBaseComponent (OOMisc) 8
- TApdAbstractFaxPrinterStatus (AdFaxPrn)

Properties

Display	FaxPrinter	● Version
---------	------------	-----------

Methods

CreateDisplay	DestroyDisplay	UpdateDisplay
---------------	----------------	---------------

Reference Section

CreateDisplay

virtual abstract method

```
procedure CreateDisplay; virtual; abstract;
```

- ↳ An abstract method that creates a form to display fax printing status.

A descendant of `TApdAbstractFaxPrinterStatus` must override this method with a routine that creates a `TForm` component that contains various controls (typically of type `TLabel`) for displaying the fax printing status. The `TForm` should also contain a `TButton` control and associated `AbortClick` event handler that allows the user to abort the fax printing.

`CreateDisplay` must then assign the instance of this form to the `Display` property.

See also: `DestroyDisplay`, `Display`

DestroyDisplay

virtual abstract method

```
procedure DestroyDisplay; virtual; abstract;
```

- ↳ An abstract method that destroys the display form.

A descendant of `TApdAbstractFaxPrinterStatus` must override this method to destroy the `TForm` instance created by `CreateDisplay`.

See also: `CreateDisplay`, `Display`

Display

run-time property

```
property Display : TForm
```

- ↳ A reference to the form created by `CreateDisplay`.

`CreateDisplay` must assign a properly initialized instance of a `TForm` to this property. `UpdateDisplay` can refer to this property to update the status window.

See also: `CreateDisplay`, `UpdateDisplay`

FaxPrinter

read-only, run-time property

```
property FaxPrinter : TApdCustomFaxPrinter
```

- ↳ The fax printer component that is using the status component.

When you derive components from `TApdAbstractFaxPrinterStatus`, you will probably reference `TApdFaxPrinter` properties to display information about the progress of the print session. Use this property to do so. It is automatically initialized when you assign the status component to the `StatusDisplay` property of `TApdFaxPrinter`.

```
procedure UpdateDisplay(First, Last : Boolean); virtual; abstract;
```

↳ An abstract method that writes the contents of the status window.

A descendant of `TApdAbstractFaxPrinterStatus` must override this method to update the display form. The `TApdFaxPrinter` component calls this method regularly from its `OnFaxPrintStatus` event handler.

On the first call to `UpdateDisplay`, `First` equals `True` and `UpdateDisplay` should call the `Show` method of `Display` to draw the outline and background of the status form. On the last call to `UpdateDisplay`, `Last` equals `True` and `UpdateDisplay` should set the `Visible` property of `Display` to `False` to erase the status window.

For all other calls to `UpdateDisplay`, `First` and `Last` both equal `False`. During these calls, `UpdateDisplay` must update the various labels in the `Display` form. To get information about the fax printing status, read the values of the various fax printer properties (use `FaxPrinter` to find the fax printer) such as `FileName` and `CurrentPrintingPage`.

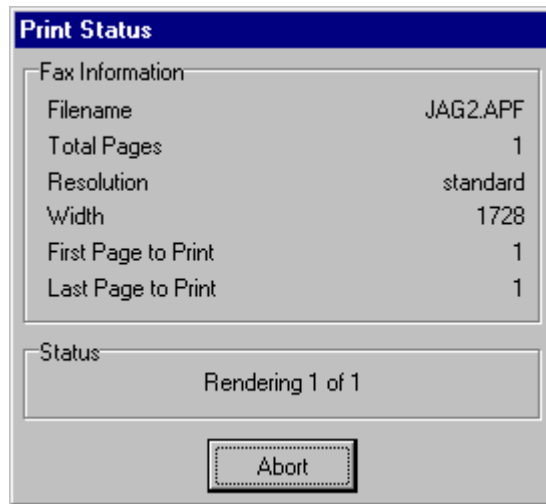
The `AbortClick` event handler, if provided, should call the `PrintAbort` method of `TApdFaxPrinter` to terminate fax printing.

TApdFaxPrinterStatus Component

TApdFaxPrinterStatus is a descendant of TApdAbstractFaxPrinterStatus that implements a standard printer status display. To use it, just create an instance and assign it to the StatusDisplay property of your TApdFaxPrinter component. TApdFaxPrinterStatus includes all of the most frequently used information about a print job and it provides an Abort button so that the user can stop the printing at any time.

TApdFaxPrinterStatus overrides all the abstract methods of TApdAbstractFaxPrinterStatus. TApdFaxPrinterStatus has no methods that you must call or properties that you must adjust. You might want to change the settings of the Ctl3D and Position properties to modify the appearance and placement of the window.

Figure 15.1 shows the TStandardFaxPrintStatusDisplay form that is associated with a TApdFaxPrinterStatus component.



The screenshot shows a window titled "Print Status" with a blue header bar. Inside the window, there are two main sections. The first section, labeled "Fax Information", contains a table with the following data:

Filename	JAG2.APF
Total Pages	1
Resolution	standard
Width	1728
First Page to Print	1
Last Page to Print	1

The second section, labeled "Status", contains the text "Rendering 1 of 1". At the bottom of the window, there is an "Abort" button with a dashed border.

Figure 15.1: TStandardFaxPrintStatusDisplay form.

For an example of using a TApdFaxPrinterStatus component, see the TApdFaxPrinter example on page 674.

Hierarchy

TComponent (VCL)

 TApdBaseComponent (OOMisc) 8

 TApdCustomFaxPrinterStatus (AdFaxPrn)

 TApdFaxPrinterStatus (AdFaxPrn)

TApdFaxPrinterLog Component

TApdFaxPrinterLog is a small component that can be associated with a TApdFaxPrinter component to provide automatic printer logging services. Just create an instance of TApdFaxPrinterLog and assign it to the FaxPrinterLog property of the TApdFaxPrinter component.

TApdFaxPrinterLog creates or appends to a text file whose name is given by the LogFileName property. Each time the OnFaxPrintLog event is generated, the associated TApdFaxPrinterLog instance opens the file, writes a new line to it, and closes the file.

Following is a sample of the text file created by TApdFaxPrinterLog:

```
Printing d:\changes.apf started at 4/17/96 3:38:58 PM
Printing d:\changes.apf finished at 4/17/96 3:39:24 PM

Printing d:\changes.apf started at 4/17/96 3:53:35 PM
Printing d:\changes.apf aborted at 4/17/96 3:53:41 PM
```

Hierarchy

TComponent (VCL)

- ① TApdBaseComponent (OOMisc) 8
 - TApdCustomFaxPrinterLog (AdFaxPrn)
 - TApdFaxPrinterLog (AdFaxPrn)

Properties

- FaxPrinter
- LogFileName
- ① Version

Methods

- UpdateLog

Reference Section

FaxPrinter

property

```
property FaxPrinter : TApdCustomFaxPrinter
```

↪ The fax component that is using the logging component.

When you derive components from TApdFaxPrinterLog, you will probably reference TApdFaxPrinter properties to display information about the progress of the print session. Use this property to do so. It is automatically initialized when you assign the status component to the StatusDisplay property of TApdFaxPrinter.

LogFileName

property

```
property LogFileName : string
```

Default: "FAXPRINT.LOG"

↪ Determines the name of the file used to store the fax printer log.

You should set the value of LogFileName before calling the PrintFax method of TApdFaxPrinter. However, because the log file is opened and closed for each update, you can change LogFileName at any time. If you set LogFileName to an empty string, automatic logging is disabled until you assign a non-empty string.

See also: TApdFaxPrinter.PrintFax

UpdateLog

method

```
procedure UpdateLog(const LogCode : TFaxPLCode); virtual;  
TFaxPLCode = (lcStart, lcFinish, lcAborted, lcFailed);
```

↪ Called for each fax printer logging event.

UpdateLog creates or appends to the log file, builds and writes a text string for each event, and closes the log file. LogCode can have the following values:

Value	Meaning
lcStart	Printing started.
lcFinish	Printing finished successfully.
lcAborted	Printing aborted before completion.
lcFailed	Printing failed.

TApdFaxPrinterLog contains a field named FaxPrinter that UpdateLog uses to obtain additional information (i.e., FileName, CurrentPrintingPage) about the print job.

See also: TApdFaxPrinter.OnFaxPrintLog

Sending and Receiving Faxes

Async Professional provides components that support send and receive services for Class 1, Class 1.0, Class 2, and Class 2.0 faxmodems. These faxmodems all provide similar capabilities. All can connect to any other Group 3 fax device, they use the same image transmission format, and all are capable of transferring data at the same speeds. The differences between them are in the area of how the PC fax software interacts with the faxmodem. When using a Class 1 or Class 1.0 faxmodem, the communication between the PC and the modem is at a fairly low level. They exchange information and commands through HDLC (High-level Data Link Control) packets instead of Hayes-type ‘AT’ sequences. The fax software is also responsible for negotiating the faxmodem-to-faxmodem data transfer rate by sending and receiving training sequences.

When using a Class 2 or Class 2.0 faxmodem, the communication between the PC and the modem is based on an extended set of Hayes-type “AT” commands and text responses. The faxmodem itself negotiates the faxmodem-to-faxmodem data transfer rate and reports the results back to the PC.

Because all faxmodems are used in such a similar fashion, they are supported with a single set of components. In general, you don’t need to be concerned with the class of the faxmodem is attached to the systems that your software supports. You call the same functions in either case.

The fax transfer process

This section describes the anatomy of a fax transfer in some detail. You don’t need to know all of this to use the Async Professional fax components. However, it may be helpful for you to understand the detailed differences between Class 1, Class 1.0, Class 2, and Class 2.0 and to know some of the limitations of Group 3 faxmodems in general. The discussion also specifies exactly when the various fax events are generated. The explanation is organized according to the fax transfer phases associated with these events. A list of these phases is described in Table 15.11.

Table 15.11: *Fax transfer phases*

Phase	Description
Phase A	Dial
Phase B	Pre-message
Phase C	Message
Phase D	Post-message
Phase E	Hang-up

The “phase name” is the term used by the TIA/EIA specification for each activity. Phase A is associated with dialing the phone number or preparing the modem to receive a call. The sender and receiver negotiate the parameters of the fax connection during Phase B. The actual fax image (the “message”) is transferred during Phase C. During Phase D, the sender and receiver decide whether a page must be resent, or whether more pages follow. During Phase E, the modem disconnects from the phone line in preparation for the next call.

All phases from A through E are processed in order unless an error occurs. An `OnFaxError` event can be generated in any phase. If a fax session is in progress, the `OnFaxError` event is followed by an `OnFaxLog` event.

Class 1/1.0 and Class 2/2.0 modems differ only in their processing of Phase B and Phase D. With Class 1/1.0 modems, the terminal software handles these two phases by transmitting and receiving HDLC packets. With Class 2/2.0 modems, the terminal sends an ‘AT’ sequence and waits for the modem to return a text response.

Phase A: Dial

The first step in Phase A is to initialize the modem for sending or receiving faxes. `TApdSendFax` continues Phase A by generating an `OnFaxNext` event to get the phone number and fax file name. If a fax is queued for sending, it dials the fax recipient’s phone number.

Phase B: Pre-message

Phase B starts as soon as the call is answered. The major task of Phase B is for the modems to agree on a set of capabilities to use while transferring the fax. The modems exchange their capabilities, then perform an iterative training procedure to find the fastest transmission parameters that are reliable.

During training, the sender starts at the fastest modulation rate that the receiver supports. It sends a known sequence of bytes, which the receiver attempts to read. If the receiver gets the right pattern, it informs the sender and the transfer moves on to Phase C. If the pattern is incorrect, the receiver informs the sender and the sender determines whether to retry at the same rate or to step down to the next lower modulation rate. The process is then repeated. Generally a reduction in transfer rate is required because the telephone line is too noisy. If no available modulation rate succeeds, the connection is terminated.

After a modulation rate is successfully negotiated, various events are generated. `TApdReceiveFax` generates an `OnFaxAccept` event to accept or reject this fax. If the fax is rejected, an `OnFaxLog` event is generated to log the rejection. If the fax is accepted, an `OnFaxName` event is generated to get a name for the received file, and an `OnFaxLog` event is generated to log the start of the receive. `TApdSendFax` generates an `OnFaxLog` event to log the start of the transmit session.

Phase C: Message

During Phase C, TApdSendFax sends page image data to the faxmodem, which transmits it to the receiving faxmodem. The receiving faxmodem sends it on to the receiver software. During this process, the sender must honor flow control, because the transmitting faxmodem could impose flow control to temporarily stop the data flow. The faxmodem releases flow control when it wants to resume.

Software or hardware flow control must also be enabled at the modem. When you use TAPI or the AWMODEM.INI database, the appropriate type of flow control is automatically enabled. Otherwise, you must ensure that flow control is enabled at the modem.

While in Phase C, the TApdSendFax component must continuously send data to the faxmodem, except when it is blocked by flow control. Any break in the stream generates a “data underflow” error. What happens after a data underflow depends on the faxmodem, but in most cases the fax session cannot be continued.

To avoid data underflow, the TApdComPort component must use a port baud rate that is higher than (not just equal to) the fax bit per second transfer rate. In almost all cases this should be 19200 baud. There’s no benefit to using higher baud rates and at least one of the modems we tested works only at 19200.

Another possible cause of data underflow is an ill-behaved Windows 3.X program that doesn’t yield CPU time or thread starvation caused by a Win32 program that raises its thread priority too high for too long. The best way to avoid this is to use a relatively large TApdComPort output buffer (for example, 16384 bytes).

OnFaxStatus events are generated regularly during Phase C. By default they are generated once every second (you can change the frequency by setting the StatusInterval property).

Phase D: Post-message

At the end of Phase C the transmitter sends an end-of-page sequence, which marks the start of Phase D. The receiver then tells the transmitter whether or not the page was received successfully. If it was not, the two modems can negotiate a retransfer. If it was, the transmitter tells the receiver whether any additional pages are coming.

If there are more pages, the process can loop back to either the middle of Phase B (to retrain the connection) or the beginning of Phase C (to keep the existing connection parameters). The receiving modem decides which approach to use. Async Professional’s receive routines loop to the beginning of Phase C. Other software packages and fax machines can choose either approach.

Phase E: Hang-up

Phase E disconnects or hangs up the modem, terminating the fax call. When transmitting faxes, the input file is closed and an OnFaxLog event is generated to indicate the end of this fax. TApdSendFax then loops back to Phase A to see whether another fax should be sent.

When receiving faxes, the output file is closed and an OnFaxLog event is generated. TApdReceiveFax then waits for another incoming call unless the OneFax property is True. If there are no more faxes to send or receive, an OnFaxFinish event is generated.

Fax send/receive events

The fax send/receive session takes place in the background and communicates with your application via four events:

OnFaxStatus

```
procedure(CP : TObject; First, Last : Boolean) of object;
```

Generated approximately once per second during the entire fax session so that programs can display the progress of the session. See “OnFaxStatus” on page 729. Also see “Fax status” on page 707.

OnFaxLog

```
procedure(CP : TObject; LogCode : TFaxLogCode) of object;
```

Generated at the start and end of each fax call. This provides the opportunity to log the status of the fax transfer. See “OnFaxLog” on page 729. Also see “Fax logging” on page 712.

OnFaxError

```
procedure(CP : TObject; ErrorCode : Integer) of object;
```

Generated when an unrecoverable error occurs. Recoverable errors do not generate this message because they are an expected part of fax transfers and, when possible, the failed operation is retried automatically. See the OnFaxError event on page 728. Also see “Error handling” on page 705.

OnFaxFinish

```
procedure(CP : TObject; ErrorCode : Integer) of object;
```

Generated after all faxes have been transmitted or received or after the fax session terminates due to an unrecoverable error. This event also sends the final result code of the fax session. See the OnFaxFinish event on page 728.

Fax sessions and the TApdComPort

Fax sessions require the following values for critical TApdComPort properties:

```
ApdComPort1.DataBits := 8;  
ApdComPort1.StopBits := 1;  
ApdComPort1.Parity := pNone;  
ApdComPort1.Baud := 19200;  
ApdComPort1.InSize := 8192;  
ApdComPort1.OutSize := 8192;  
ApdComPort1.HWFlowOptions := [hwfUseRTS, hwfRequireCTS];
```

When TApdSendFax and TApdReceiveFax first link to a TApdComPort component (i.e., when their ComPort properties are set) they immediately force the above properties to these values. You are then free to change these properties; the fax components won't try to change them back again. However, improper changes to these critical properties can result in unreliable fax operation.

Databits, Stopbits, and Parity must be set to 8,1,none since that is the proper setting for binary data transfer. Fax devices don't use parity checking.

Baud must be set to 19200 for a couple of reasons. First, the fax software must continuously transmit data to the faxmodem, so that it can continuously transmit data to the receiving fax device. A pause in a fax data stream is considered by the fax device to be a fatal error and usually results in an immediate abort of the fax session, or an abort at the end of the current page.

Because the highest fax bps rate is 14400 bits per second, a comport baud rate slightly higher than that, 19200, is sufficient to assure that short pauses in the data stream from the software do not result in pauses between the fax devices.

A baud rate higher than 19200 isn't necessary since the fax bps rate will never be greater than 14400. Selecting a baud rate higher than 19200 does little to improve pause tolerance but increases the possibility of line errors.

The second reason for forcing the baud rate to 19200 is that a few older faxmodems require that baud rate and won't operate at any other baud rate.

InSize is the size, in bytes, of the comport component's communication input buffer. It's forced to 8192 from the default 4096 to increase the fax software's tolerance of ill-behaved programs that don't yield often or that increase their thread priority for too long.

OutSize is the size, in bytes, of the comport component's communication output buffer. It's forced to 8192 from the default 4096 to better protect against data underflows. The fax component keeps the output buffer as full as possible. If the fax application is kept from running due to an ill-behaved program, the Windows communication driver continues to transmit the buffered data. As long as the fax application is allowed to run again before the output buffer drains, no data underflows occur.

HWFlowOptions is one of the TAPdComPort properties (the other is SWFlowOptions) that determines what type of flow control, if any, is used by the comport component. The faxmodem specifications dictate that all faxmodems support software flow control. Faxmodems can optionally support hardware flow control, and most do. Because hardware flow control is more reliable than software flow control, and because both the Async Professional modem database and the Win32 TAPI modem database force hardware flow control on by default, hardware flow control is forced on in the TAPdComPort.

Flow control is a critical issue for fax sessions and it must be properly enabled, both in fax application software and in the faxmodem, for reliable fax operation. Assuring hardware flow control is enabled in the software is only half the battle—you must also assure that hardware flow control is enabled at the faxmodem.

Fortunately, this is easy to do, although it isn't as automatic as the enabling of flow control within the TAPdComPort component. Fax applications must do one of the following:

1. Use TAPI to initialize the modem.
2. Use the TAdModem and modemcap database components to initialize the modem.
3. Manually send the appropriate modem commands to enable hardware flow control.

Number 1 is the preferred choice if it is available. The TAPI modem database is far more extensive than the Async Professional database. 32-bit Windows users expect your program to use TAPI rather than direct serial port access.

Number 2 is the preferred choice when TAPI isn't available. Although it's possible that the exact modem model isn't in the modemcap database, choosing the closest match from the same manufacturer is usually sufficient to assure that the modem is properly configured.

Number 3 is the most difficult approach since you must figure out the appropriate modem commands to send to the modem to enable hardware flow control. Use it only when neither TAPI or modemcap are available.

TAPI/Fax Integration

TAPI provides several features that supplement faxing. The most obvious is that TAPI permits selection of the modem by name instead of by port number. TAPI also pre-configures the modem to a state where APRO can re-initialize it for faxing. When sending a fax, TAPI can apply the dialing options to the phone number, for example to automatically add a '9' for an outside line. When receiving a fax, TAPI can passively monitor the line for incoming calls, which allows other processes to access the port.

TAPI integration with the faxing components is determined by the `TapiDevice` property of the `TApdAbstractFax` component (and the `TApdSendFax` and `TApdReceiveFax` descendants). If `TapiDevice` is not assigned, TAPI integration will not be used. If `TapiDevice` is assigned, that device will be used to select and configure the modem, provide phone number translations, and passively answer calls.

The level of TAPI integration depends on whether a fax is being sent or received. When the `StartTransmit` method of the `TApdSendFax` component is called, and the `TapiDevice` property is assigned, the `ConfigAndOpen` method of the `TapiDevice` will be called. Internal `OnTapiXxx` event handlers will be assigned to monitor the progress of the `TapiDevice`. When TAPI configures the modem and opens the port, the `PhoneNumber` property will be translated using the `TranslateAddress` method of the `TapiDevice`, and that translated number will be dialed. From there, the fax will be sent in the usual manner. Once the `OnFaxFinish` event is generated, the `CancelCall` method of the `TapiDevice` will be called to close the port.

When the `StartReceive` method of the `TApdReceiveFax` component is called, and the `TapiDevice` property is assigned, the `AnswerOnRings` property of the `TapiDevice` will be set to 100 to prevent TAPI from answering the call, and the `AutoAnswer` method of `TapiDevice` will be called. When TAPI detects `AnswerOnRings` ring signals, the `TapiDevice` will cancel itself, and the `TApdReceiveFax` will answer the call. When the `OnFaxFinish` event is generated, the `CancelCall` method of the `TapiDevice` will be called to close the port.

When the `TapiDevice` property is assigned, the event handlers of the `TApdTapiDevice` will be generated as usual. These events will be generated from within the internal `OnTapiXxx` event handler that the faxing components will assign. Any additional processing that your project does within these events must be complete before your project's `OnTapiXxx` events return.

To maintain backwards compatibility with previous versions of APRO that did not provide TAPI and fax integration, set the `TapiDevice` property to nil.

One benefit of TAPI integration is that you can now send a fax with the `TApdSendFax` component while a `TApdReceiveFax` component is waiting for faxes. To implement this, the `TApdSendFax` component must have a `TApdComPort` and `TApdTapiDevice` component

assigned, and the TAPdReceiveFax component must have a different TAPdComPort and TAPdTapiDevice component assigned. The SelectedDevice property of both TAPdTapiDevice components can point to the same TAPI device. Call the StartReceive method of the TAPdReceiveFax component to begin passively answering calls. When ready to transmit a fax, check the InProgress property of the TAPdReceiveFax to make sure a fax is not being actively received. If InProgress is False, then it is safe to call the StartTransmit method of the TAPdSendFax. When the fax has been sent, the TAPdReceiveFax will still be in passive answer mode.

Aborting a fax session

There will be times when you need to cancel a fax transfer while it is still in progress. This may be necessary if something goes wrong at the remote fax machine or if the user simply decides not to continue the transfer. The CancelFax method gives your program an opportunity to inform the fax routines that you want to stop the transfer.

Typically, a program checks for the user entering characters such as <Esc> or <CtrlX> to signal cancellation of the transfer. Because all fax transfers are made through modem connections, you might be able to use the modem's DCD (Data Carrier Detect) signal. You could check it within the abort function and abort the transfer quickly if the connection is broken, without waiting for the timeouts built into the fax transfer routines.

Unfortunately, a low DCD signal is not always a reliable indication of a broken connection. Faxmodems constantly drop and raise the physical carrier signal during initial handshaking and after each page. If the modem's DCD signal follows the physical carrier then you should not assume a low DCD indicates a broken connection. For modems that set DCD high during the entire fax session, you can assume that a low DCD indicates a broken connection and abort the fax session. Given the uncertainty of the behavior of the DCD signal, unless you are certain that the modem's DCD signal has the latter behavior, you should not use DCD, but wait for the modem to timeout at the end of the current page.

Error handling

All fax transfers are subject to errors like line errors, file not found errors, and other file I/O errors. The fax components handle errors internally by retrying the operation or requesting the remote fax device to retry. If, however, the situation is unrecoverable, an OnFaxError event is generated. Your application should include an handler for this event. Following is a simple example:

```
procedure Form1.ApSendFax1FaxError(  
    CP : TObject; ErrorCode : Integer);  
begin  
    ShowMessage('Fatal fax error: ' + ErrorMessage(ErrorCode));  
end;
```

This event handler's sole task is to display a message about the error. `ErrorMsg` is a function from the `AdExcept` unit that returns an English string for any Async Professional error code. See "Error Handling and Exception Classes" on page 900 for additional information about errors. Table 15.12 contains an annotated list of error codes that apply to fax calls.

Table 15.12: *Fax call error codes*

Error Code	Meaning
<code>ecFaxBadMachine</code>	The called fax device doesn't support the resolution or the width of the document to send.
<code>ecFaxBadModemResult</code>	The faxmodem sent an unexpected response string; no recovery is possible.
<code>ecFaxBusy</code>	The called fax device is busy. The call is automatically retried <code>DialAttempts</code> times. If the line is still busy, the fax session ends with the <code>ecFaxBusy</code> error code.
<code>ecFaxDataCall</code>	One end of the connection is a data call, not a fax call.
<code>ecFaxInitError</code>	An error occurred in the initialization process. Usually means that the modem returned "ERROR". This can occur if the local modem is not a Class 1, Class 1.0, Class 2, or Class 2.0 faxmodem.
<code>ecFaxNoCarrier</code>	An incoming call was not successfully answered. Usually means that the modems could not agree on a modulation rate.
<code>ecFaxNoDialTone</code>	The faxmodem returned a NO DIALTONE error message after a dial attempt.
<code>ecFaxNoFontFile</code>	The external font file <code>APFAX.FNT</code> could not be found. This file is used for converting header lines and cover pages to fax format.
<code>ecFaxPageError</code>	An unexpected response was received while waiting for an acknowledgement to the last transmitted page; no recovery is possible.

Table 15.12: *Fax call error codes (continued)*

Error Code	Meaning
ecFaxSessionError	The fax session failed. This error occurs when the remote fax device doesn't respond to handshaking requests or specifically requests an abort. Possible causes of this error are lack of flow control, very poor line quality, or an operator-requested abort at the remote fax device. For Class 2 and Class 2.0 modems, you can get a more exact error code by calling <code>GetHangupResult</code> . Unfortunately these codes are not usually helpful, because they describe activities controlled by the modem itself.
ecFaxTrainError	The training procedure used to select a modulation rate for a Class 1/1.0 faxmodem did not succeed for any of the possible rates.
ecFaxVoiceCall	One end of the connection is a voice call, not a fax call.

Fax status

A fax session can last a few seconds or several hours, depending on the size and speed of the transfer. The fax session proceeds in the background without intervention from your program. To inform you of what's happening as the transfer progresses, Async Professional provides an event for regular notification of the progress.

During a fax session, an `OnFaxStatus` event is generated regularly (the default is once per second). This gives your application the opportunity to monitor and display the progress of the transfer. The following code fragments show how to monitor the progress.

```
TForm = class(TForm)
...
  FN : TLabel;
  PG : TLabel;
  BT : TLabel;
...
end;
```

```

procedure TForm1.ApSendFax1FaxStatus(
  CP : TObject; First, Last : Boolean);
begin
  if First then
    ...do setup stuff
  else if Last then
    ...do cleanup stuff
  else begin
    {Show status}
    FN.Caption := ApSendFax1.FaxFile;
    PG.Caption := ApSendFax1.CurrentPage;
    BT.Caption := ApSendFax1.BytesTransferred;
  end;
end;

```

The `ApSendFax1FaxStatus` method handles the `OnFaxStatus` event by updating a form at each call. The `First` and `Last` flags passed to this routine indicate whether this is the first call to this event or the last call to this event. When transmitting faxes, it is clear when `OnFaxStatus` events should be generated—the first event should be generated just after `StartTransmit` is called, events should be generated regularly while sending faxes, and the last event should be generated when there are no more faxes to send.

It is less clear when `OnFaxStatus` events should be generated when receiving faxes. They could be started as soon as `StartReceive` is called and continued until the user cancels the receive session. This behavior is appropriate for a dedicated fax server application that is constantly receiving and printing faxes. Other applications, however, might prefer not to receive `OnFaxStatus` events until an incoming call is detected.

`TApdFaxReceive` provides for both situations by providing the `ConstantStatus` property. Set `ConstantStatus` to `False` (the default) to receive `OnFaxStatus` events only for the duration of a fax call. Set `ConstantStatus` to `True` to receive `OnFaxStatus` events from the time `StartReceive` is called until the user cancels the fax session.

Information about the progress of the fax session is obtained by reading the values of various `TApdAbstractFax`, `TApdSendFax` and `TApdReceiveFax` properties, including:

BytesTransferred: The number of bytes transmitted or received so far for the current page.

CurrentPage: The number of the current page. `CurrentPage` is zero when transmitting a cover page, one for the first page, two for the second page, and so on.

ElapsedTime: The elapsed time (in milliseconds) since the remote station ID was received. An indicator of the elapsed time for the current fax call.

FaxProgress: A code indicating the current state of the fax call. The Table 15.3 shows all of the possible values. The usual states are fpSendPage and fpGetPage, which mean the fax session is sending or receiving page data. Other status values indicate various start-up and handshaking states. Fatal errors are not represented by fax states because they are first reported via the OnFaxError event. However, it is possible that a final status message might be sent after a fatal error occurs.

Table 15.13: *Possible FaxProgress values*

Status Code	Transmit /Receive	Value	Explanation
fpInitModem	T	1	The faxmodem is being initialized for fax use. For Class 1 and 1.0 modems, this is nearly instantaneous; for Class 2 and 2.0 modems, it takes a couple of seconds.
fpDialing	T	2	The faxmodem is dialing and waiting for a response from the remote fax device.
fpBusyWait	T	3	The called number was busy. Another dial attempt will be made after DialRetryWait seconds.
fpSendPage	T	4	Page data is being transmitted. The status event handler can interrogate the CurrentPage and BytesTransferred properties to track the progress of sending/receiving this page.
fpSendPageStatus	T	5	All data for the current page has been transmitted. The remote fax device is being told whether there are more pages to follow.
fpPageError	T	6	The remote fax device did not successfully receive the page data and it should be sent again.
fpPageOK	T	7	The remote fax device received the page data successfully.
fpWaiting	R	20	The faxmodem is waiting for incoming fax calls.

Table 15.13: *Possible FaxProgress values (continued)*

Status Code	Transmit /Receive	Value	Explanation
fpNoConnect	R	21	The incoming call was not a fax call.
fpAnswer	R	22	The faxmodem is answering an incoming call.
fpIncoming	R	23	The incoming call has been validated as a fax call.
fpGetPage	R	24	Page data is currently being received. The status event handler can interrogate CurrentPage and BytesTransferred to find out which page is being received and how many bytes have been received so far.
fpGetPageResult	R	25	The faxmodem just reported whether it received a page successfully. The status procedure can interrogate LastPageStatus to find out the result.
fpCheckMorePages	R	26	The faxmodem is waiting for the remote fax device to indicate whether it has more pages to send.
fpGetHangup	R	27	The faxmodem is waiting for a response to a disconnect or hangup command.
fpGotHangup	R	28	The faxmodem has disconnected or hung up.
fpSessionParams	T/R	40	The local and remote fax devices completed negotiation of session parameters. The status event handler can interrogate the SessionBPS, SessionResolution, and SessionWidth properties for the negotiated parameters of the current session.

Table 15.13: *Possible FaxProgress values (continued)*

Status Code	Transmit /Receive	Value	Explanation
fpGotRemoteID	T/R	41	A fax connection was established and the remote fax has reported its station ID. The status event handler can now interrogate RemoteID to get the ID string.
fpCancel	T/R	42	The session was cancelled by the user or by the remote device.
fpFinished	T	43	The fax session is over.

FaxError: The code of the fatal error encountered in the fax session. See the error codes in “Error handling” on page 705 for more information.

HangupCode: The hangup code returned by a class 2 or class 2.0 faxmodem. The hangup code can sometimes provide additional error information for failed fax sessions.

ModemModel: The modem model identification string returned by the faxmodem.

ModemRevision: The modem revision identification string returned by the faxmodem.

ModemChip: The modem chip identification string returned by the faxmodem.

ModemBPS: The highest bps rate supported by the modem.

ModemECM: Indicates whether the modem supports error control mode.

PageLength: The length in bytes of the page currently being transmitted. When a fax is being received, PageLength is zero.

RemoteID: The 20 character identification string returned by the remote fax device.

SessionBPS: The negotiated bps rate (bytes per second) for the current fax session.

SessionResolution: The negotiated resolution (standard or high) for the current fax session.

SessionWidth: The negotiated width (1728 or 2048 pixels) for the current fax session.

SessionECM: Indicates whether the current fax session is using error control.

TotalPages: The total number of pages to be transmitted. When a fax is being received, TotalPages is zero.

Automatic fax status display

Async Professional includes a mechanism for providing an automatic fax status display without programming, through the `TApdAbstractFaxStatusDisplay` property:

```
property StatusDisplay : TApdAbstractFaxStatus
```

The `TApdAbstractFaxStatus` class is described in more detail beginning on page 822. If `StatusDisplay` is assigned, its `UpdateDisplay` method is called to update the display. Then the `OnFaxStatus` event handler is called, if one is implemented.

When a fax component is created, either dynamically or when dropped on a form, it searches the form for an `TApdAbstractFaxStatus` instance and updates the `StatusDisplay` property with the first one it finds. `StatusDisplay` is also filled in if a `TApdAbstractFaxStatus` component is added to the form later. You can also change `StatusDisplay` at design time or run time to point to a different `TApdAbstractFaxStatus` component.

Async Professional also provides an non-abstract implementation of `TApdAbstractFaxStatus` called the `TApdFaxStatus` component. If you drop one of these on your form, it automatically displays full status information during all fax sessions. See page 826 for more information.

Fax logging

It's often important to have a log of all incoming and outgoing activity on a particular fax machine. You can use the log for billing purposes, to recover the station ID or image of a fax whose printout was lost, or to determine which faxes were not successfully sent during an automated transfer.

The fax logging event provides a mechanism for your program to keep an activity log. Although the logging event is similar to the status event, it is not generated as frequently. The status event is generated for at least 20 different send and receive states, and it is also generated once per second during the transfer of page data. The logging event is generated once at the beginning and once at the end of each transfer.

The LogCode parameter is of an enumerated type that indicates the condition at the time the logging routine is called. The TLogFaxCode values are described in Table 15.14.

Table 15.14: *TLogFaxCode values*

Value	Explanation
lfaxTransmitStart	A fax transmission is starting.
lfaxTransmitOk	The fax was transmitted successfully.
lfaxTransmitFail	The fax was not transmitted successfully.
lfaxReceiveStart	Fax reception is starting.
lfaxReceiveOk	The fax was received successfully.
lfaxReceiveSkip	The incoming fax was rejected.
lfaxReceiveFail	The fax was not received successfully.

When LogCode is lfaxReceiveSkip, RemoteID contains the identification string of the remote station but FaxFile is an empty string because the fax session never progressed to the point of generating a fax file name.

Following is an example OnFaxLog event handler:

```

procedure TForm1.ApdReceiveFax1FaxLog(
    CP : TObject; LogCode : TLogFaxCode);
begin
    case LogCode of
        lfaxReceiveStart :
            CurrentFile.Caption := ApdReceiveFax1.FaxFile;
        lfaxReceiveOK :
            GoodList.Items.Add(ApdReceiveFax1.FaxFile);
        lfaxReceiveFail :
            BadList.Items.Add(ApdReceiveFax1.FaxFile);
        lfaxReceiveSkip :
            SkipList.Items.Add(ApdReceiveFax1.RemoteID);
    end;
end;

```

This example shows the logging values that could be received during a fax receive session. The example uses a TLabel control named CurrentFile to display the name of the current fax file. As faxes are received this method updates three TListBox components: GoodList for all successful transfers, BadList for all failed transfers, and a SkipList of RemoteIDs for all skipped files.

The logging routine isn't limited to just writing logging information. It can also take care of file-related startup and cleanup activities. One example of this might be to delete outgoing fax files (if successfully transmitted, of course) or moving them from the current directory to a "has been sent" directory.

Automatic fax logging

Async Professional includes a mechanism for providing automatic fax logging without programming, through the FaxLog property of TApdAbstractFax:

```
property FaxLog : TApdFaxLog
```

The TApdFaxLog class is described in more detail on page 828. For each OnFaxLog event, the fax component checks whether FaxLog is assigned. If it is, the fax calls the UpdateLog method of TApdFaxLog to write information to the log file. It then calls the OnFaxLog event, if one is implemented.

When a fax component is created, either dynamically or when dropped on a form, it searches the form for a TApdFaxLog component and updates the FaxLog property with the first one it finds. FaxLog is also filled in if a TApdFaxLog component is added to the form later. You can also change FaxLog at design time or run time to point to a different TApdFaxLog.

TApdAbstractFax Component

The fax send and receive components (TApdSendFax and TApdReceiveFax) are derived from TApdAbstractFax. This abstract fax component provides the set of properties, methods, and events that are common to both sending and receiving, such as the Station ID of the sender or receiver, and the status of the current fax session. TApdAbstractFax also defines the events that allow you to log fax calls, generate file names for incoming faxes, and report status during transmission.

If you already know how to use file transfer protocols within the Async Professional architecture, fax transfers will seem very familiar. Just like a file transfer, a fax transfer involves initializing a comport component, initializing a fax component, and calling a method to start the background fax session. The events for fax transfer are quite similar, and in some cases identical, to those for file transfer.

There are a few differences, though, and these differences are inherent to the nature of faxes. Unlike most file transfer protocols, where the name and size of the file are transmitted before the file's contents are sent, a fax receiver isn't given a filename in which to store an incoming fax, and it doesn't know the size of the fax until the transmission terminates. The fax protocol also provides no means for the receiver to detect errors in the incoming data, or to request retransmission in the event of bad data. Fortunately, the compression technique used for fax data is relatively tolerant of errors. Because of these differences, your program must take on certain responsibilities that are not required when using file transfer protocols, such as providing names for incoming faxes and being prepared for out-of-disk-space errors when receiving faxes.

Note that certain properties that are described in the following reference section are specific to either sending or receiving faxes, or behave differently when sending or receiving faxes. These differences are described in the reference section of affected properties.

Hierarchy

TComponent (VCL)

❶ TApdBaseComponent (OOMisc)

8

TApdCustomAbstractFax (AdFax)

TApdAbstractFax (AdFax)

Properties

AbortNoConnect	HangupCode	SessionECM
BytesTransferred	InitBaud	SessionResolution
ComPort	InProgress	SessionWidth
CurrentPage	ModemBPS	SoftwareFlow
DesiredBPS	ModemChip	StationID
DesiredECM	ModemECM	StatusDisplay
ElapsedTime	ModemInit	StatusInterval
ExitOnError	ModemModel	SupportedFaxClasses
FaxClass	ModemRevision	TapiDevice
FaxFile	NormalBaud	TotalPages
FaxFileExt	PageLength	❶ Version
FaxLog	RemoteID	
FaxProgress	SessionBPS	

Methods

CancelFax	ConvertToLowRes
ConvertToHighRes	StatusMsg

Events

OnFaxError	OnFaxLog
OnFaxFinish	OnFaxStatus

Reference Section

AbortNoConnect

property

`property AbortNoConnect : Boolean`

Default: False

- ↳ Defines what happens when the connection to a fax number cannot be made after the default number of retries.

If `AbortNoConnect` is `True`, the background fax process ends and generates an `OnFaxFinish` event, even if additional faxes are queued to be sent. If `AbortNoConnect` is `False`, the background fax process moves on to the next fax to be sent, as specified by the `OnFaxNext` event handler.

BytesTransferred

read-only, run-time property

`property BytesTransferred : Boolean`

- ↳ The number of bytes received or transmitted so far for the current page.

`BytesTransferred` can be used by an `OnFaxStatus` event handler to get the number of bytes received or transferred so far. The appropriate time to check `BytesTransferred` is when `FaxProgress` equals `fpSendPage` or `fpGetPage`. At other times, it is either zero or a value associated with the previous page.

See also: `CurrentPage`, `PageLength`, `TotalPages`

CancelFax

method

`procedure CancelFax;`

- ↳ Cancels the current fax session.

`CancelFax` cancels the fax session, regardless of its current state. When appropriate, a cancel command is sent to the local modem or the remote fax device. The fax component generates an `OnFaxFinish` event with the error code `ecCancelRequested`, then cleans up and terminates. It also attempts to put the faxmodem back “onhook” (i.e., ready for the next call).

The following example shows how to cancel a fax from a fax status dialog:

```
procedure TStandardDisplay.CancelClick(Sender : TObject);
begin
    ApdSendFax1.CancelFax;
end;
```

See also: OnFaxError, OnFaxFinish

ComPort

property

```
property ComPort : TApdCustomComPort
```

↳ Determines the serial port used by the fax component.

A properly initialized comports component must be assigned to this property before sending or receiving faxes.

When a TApdComPort is assigned to the ComPort property, the fax component forces the ComPort to the property values shown below:

```
ApdComPort1.DataBits := 8;
ApdComPort1.StopBits := 1;
ApdComPort1.Parity := pNone;
ApdComPort1.Baud := 19200;
ApdComPort1.InSize := 8192;
ApdComPort1.OutSize := 8192;
ApdComPort1.HWFlowOptions := [hwfUseRTS, hwfRequireCTS];
```

These values are essential for proper and reliable fax operation and should be changed only if you are certain of the impact of your changes. See “Fax sessions and the TApdComPort” on page 702 for more information.

ConvertToHighRes

method

```
procedure ConvertToHighRes(const FileName : string);
```

↳ Converts a fax file to high resolution.

ConvertToHighRes converts an existing APF fax file into high resolution. If the APF contains multiple pages, ConvertToHighRes converts all pages to high resolution. Each fax page is converted to a bitmap, converted back to a single-high resolution page APF in the Windows temp folder, and then all pages are concatenated to recreate the original APF. If the page is already high resolution, an exception is not raised.

See also: ConvertToLowRes

```
procedure ConvertToLowRes(const FileName : string);
```

- ↪ Converts a fax file to low resolution.

ConvertToLowRes converts an existing APF fax file into low resolution. If the APF contains multiple pages, ConvertToLowRes converts all pages to low resolution. Each fax page is converted to a bitmap, converted back to a single low resolution pages APF in the Windows temp folder, and then all pages are concatenated to recreate the original APF. If the page is already low resolution, and exception is not raised.

See also: ConvertToHighRes

```
property CurrentPage : Word
```

- ↪ The page number of the page currently being received or transmitted.

CurrentPage can be used by an OnFaxStatus event handler to get the number of the page currently being received or transmitted. The appropriate time to check CurrentPage is when FaxProgress equals fpSendPage or fpGetPage. At other times, it is either zero or a value associated with the previous page.

See also: BytesTransferred, PageLength, TotalPages

```
property DesiredBPS : Word
```

Default: 9600

- ↪ Determines the highest fax bps rate to negotiate for the next fax session.

DesiredBPS limits the fax bps rate for subsequent fax sessions. Although many faxmodems support higher bps rates (12000 and 14400), DesiredBPS defaults to 9600 for more reliable fax sessions and higher quality faxes because the slightly lower baud rate makes lines errors less likely.

Changing DesiredBPS during a fax session has no effect on the current session.

See also: ModemBPS, SessionBPS

DesiredECM

property

property DesiredECM : Boolean

Default: False

↳ Determines whether fax sessions attempt to use error control.

The fax protocol contains an optional error control facility that allows modems to detect and correct some transmission errors. Since very few faxmodems support fax error control, DesiredECM defaults to False, meaning the faxmodems do not attempt to negotiate error control.

See also: ModemECM, SessionECM

ElapsedTime

read-only, run-time property

property ElapsedTime : DWORD

↳ Indicates the elapsed time for the fax call.

ElapsedTime is the number of milliseconds that have elapsed since fax call has started. The fax call, in this context, is considered started when the remote station ID has been received. This will usually be less than a second or two after the call is actually answered.

ExitOnError

property

property ExitOnError : Boolean

Default: False

↳ Determines what happens when an error occurs during a fax transmit or receive.

If ExitOnError is True, no more faxes are transmitted or received. If ExitOnError is False (the default), the background fax process continues on with the next fax in the transmit queue, or resets the modem and waits for a new incoming fax.

See also: AbortNoConnect

FaxClass

property

```
property FaxClass : TFaxClass
```

```
TFaxClass = (  
    fcUnknown, fcDetect, fcClass1, fcClass1_0, fcClass2, fcClass2_0);
```

Default: fcDetect

↪ Indicates whether the faxmodem is used as Class 1, Class 1_0, Class 2, or Class 2.0.

If FaxClass is fcDetect (the default), TApdAbstractFax determines what classes the modem supports and enables the highest class. If you set FaxClass to a specific class, no attempt is made to determine if the class you request is supported by the faxmodem.

See also: SupportedFaxClasses

FaxFile

property

```
property FaxFile : string
```

↪ The name of the fax file currently being transmitted or received.

If you are sending a single fax, set FaxFile to the name of the file. If you are sending multiple fax files, you must implement an OnFaxNext event handler. FaxFile is automatically set to the fax file returned by your event handler.

FaxFile can be used with status and logging routines to return the name of the fax file currently being transmitted or received.

See also: TApdSendFax.CoverFile, TApdSendFax.OnFaxNext,
TApdSendFax.PhoneNumber

FaxFileExt

property

```
property FaxFileExt : string
```

Default: "APF"

↪ The default extension assigned to incoming fax files.

By default, all incoming fax files created by the two built-in methods of naming faxes use a file extension of APF. You can change the extension assigned to incoming files by setting FaxFileExt to the new desired extension.

See "Naming incoming fax files" on page 754 for more information.

See also : TApdReceiveFax.FaxNameMode

```
property FaxLog : TApdFaxLog
```

↳ An instance of a fax logging component.

If FaxLog is nil (the default), the fax component does not perform automatic logging. You can install an OnFaxLog event handler to perform logging in this case.

If you create an instance of (or a descendant of) a TApdFaxLog class (see page 828), and assign it to FaxLog, the fax component will call the log component's UpdateLog method automatically.

FaxProgress

read-only, run-time property

```
property FaxProgress : Word
```

↳ Returns a code that indicates the current state of the fax session.

This property is most useful within an OnFaxStatus event handler. See “Fax status” on page 707 for more information.

See also: OnFaxError, OnFaxStatus

HangupCode

read-only, run-time property

```
property HangupCode : Word
```

↳ The hangup code for a Class 2 or 2.0 fax transfer.

When a Class 2 or 2.0 faxmodem session terminates abnormally, it returns a “hangup code” to help explain what went wrong. Although these codes refer to low-level portions of the faxmodem link over which you have no control, sometimes they can point out a programming error that you can correct.

The following table shows the codes that can be returned (in hexadecimal), with a brief description of each one. The codes are grouped according to the transfer phase in which they can occur. Some of the terms in the table are defined only in the Class 2 and 2.0 specification. Refer to that specification for more information.

Hangup Code	Description
Call placement and termination	
00	Normal end of connection
01	Ring detect without successful handshake
02	Call aborted from +FKS or <Can>
03	No loop current

Hangup Code	Description
04	Ringback detected, no answer timeout
05	Ringback detected, answer without CED
Transmit phase A	
10	Unspecified phase A error
11	No answer
Transmit phase B	
20	Unspecified phase B error
21	Remote cannot receive or send
22	COMREC error in transmit phase B
23	COMREC invalid command received
24	RSPREC error
25	DCS sent three times without response
26	DIS/DTC received three times; DCS not recognized
27	Failure to train at 2400 bps or +FMS error
28	RSPREC invalid response received
Transmit phase C	
40	Unspecified transmit phase C error
41	Unspecified image format error
42	Image conversion error
43	DTE to DCE data underflow
44	Unrecognized transparent data command
45	Image error, line length wrong
46	Image error, page length wrong
47	Image error, wrong compression code
Transmit phase D	
50	Unspecified transmit phase D error
51	RSPREC error
52	MPS sent three times without response
53	Invalid response to MPS
54	EOP sent three times without response
55	Invalid response to EOP
56	EOM sent three times without response
57	Invalid response to EOM

Hangup Code	Description
58	Unable to continue after PIN or PIP
Receive phase B	
70	Unspecified receive phase B error
71	RSPREC error
72	COMREC error
73	T.30 T2 timeout, expected page not received
74	T.30 T1 timeout after EOM received
Receive phase C	
90	Unspecified receive phase C error
91	Missing EOL after 5 seconds
92	Bad CRC or frame (ECM mode)
93	DCE to DTE buffer overflow
Receive phase D	
100	Unspecified receive phase D error
101	RSPREC invalid response received
102	COMREC invalid response received
103	Unable to continue after PIN or PIP

InitBaud	property
-----------------	-----------------

property InitBaud : Integer

Default: 0

↳ Determines the initialization baud rate for modems that require different baud rates for initialization and fax operations.

Some older 24/96 faxmodems (2400 data, 9600 fax), require that the initialization commands be sent at 2400 baud, but that all fax commands and fax data be sent and received at 19200. The fax software must constantly adjust the current baud rate depending on the operation it is performing.

Since most faxmodems do not require a special initialization baud rate, InitBaud defaults to zero, which means that no baud rate switches are performed. If you encounter an older modem that requires this behavior, set InitBaud to 2400.

NormalBaud is a companion property to InitBaud. When InitBaud is non-zero, the fax components switch to the specified baud rate when sending initialization commands and switch back to the normal baud rate, 19200, when sending fax commands or fax data. If you encounter a case where the normal baud rate should be something other than 19200, you must change NormalBaud.

See also: NormalBaud

InProgress

read-only, run-time property

property InProgress : Boolean

↳ Indicates whether a fax is actively being transmitted or received.

InProgress will be True immediately after StartTransmit is called, and when a fax is being received following a call to StartReceive. Use this property to determine whether the fax component is actively transferring a fax or not.

ModemBPS

read-only, run-time property

property ModemBPS : LongInt

↳ Returns the highest bps rate supported by the faxmodem.

When you reference ModemBPS, commands are sent to the modem to determine its highest bps rate. This works only for Class 2 and 2.0 modems; a Class 1 modem cannot report this information until a fax connection has been established.

ModemBPS works by attempting to enable the most capable modem features and stepping down if the modem returns "ERROR." It starts at a 14400 bps transfer rate, then tries 12000, 9600, 7200, 4800, and 2400.

The technique used by ModemBPS works on most Class 2 and 2.0 faxmodems. One low-cost, no-name-clone faxmodem tested wouldn't return "ERROR" no matter what it was asked to do, even though it supported only 9600 bps with no error correction.

See also: ModemECM

ModemChip

read-only, run-time property

```
property ModemChip : string
```

↳ Returns the type of chip for a Class 2 or 2.0 faxmodem.

When you reference `ModemChip`, commands are sent to the modem to determine the type of chip. This works only for Class 2 and 2.0 modems. `ModemChip` is an empty string for a Class 1/1.0 modem.

See also: `ModemModel`, `ModemRevision`

ModemECM

read-only, run-time property

```
property ModemECM : Boolean
```

↳ Indicates whether the faxmodem supports error correction.

When you reference `ModemECM`, commands are sent to the modem to determine whether it supports error correction. This works only for Class 2 and 2.0 modems; a Class 1/1.0 modem cannot report this information until a fax connection has been established.

The technique used by `ModemECM` works on most Class 2 and 2.0 faxmodems. One low-cost, no-name-clone faxmodem that was tested wouldn't return "ERROR" no matter what was tried, even though it supported only 9600 bps with no error correction.

See also: `ModemBPS`

ModemInit

property

```
property ModemInit : TModemString
```

```
TModemString = string[40];
```

↳ A custom modem initialization string.

If a custom modem initialization string is assigned to `ModemInit`, Async Professional always sends this string to the modem just before it sends its own `DefInit` string ("ATE0Q0V1X4S0=0"). This occurs whenever `StartTransmit`, `StartReceive`, or `InitModemForFaxReceive` is called.

Note that the `DefInit` string may override certain actions of the `ModemInit` string. This is necessary for proper operation of the Async Professional fax routines.

The string should not contain an "AT" prefix or a trailing carriage return.

ModemModel

read-only, run-time property

```
property ModemModel : string
```

↳ Returns the model for a Class 2 or 2.0 faxmodem.

When ModemModel is referenced, commands are sent to the modem to determine the model. This works only for Class 2 and 2.0 modems. ModemModel is an empty string for a Class 1 modem.

See also: ModemChip, ModemRevision

ModemRevision

read-only, run-time property

```
property ModemRevision : string
```

↳ Returns the revision for a Class 2 or 2.0 faxmodem.

When ModemRevision is referenced, commands are sent to the modem to determine the revision. This works only for Class 2 and 2.0 modems. ModemRevision is an empty string for a Class 1/1.0 modem.

See also: ModemChip, ModemModel

NormalBaud

property

```
property NormalBaud : Integer
```

Default: 0

↳ Determines the normal baud to use for modems that require different baud rates for initialization and fax operations.

NormalBaud isn't needed unless the faxmodem requires separate baud rates for initialization commands and the baud rate required for normal fax operations is not 19200. See InitBaud for a complete description of the operation of InitBaud and NormalBaud.

See also: InitBaud

```
property OnFaxError : TFaxErrorEvent  
  
TFaxErrorEvent = procedure(  
    CP : TObject; ErrorCode : Integer) of object;
```

↳ Defines an event handler that is called when an unrecoverable fax error occurs.

This event is generated only for unrecoverable errors. Most fax errors caused by line noise are handled automatically by the fax devices and are not reported to this event handler.

CP is the fax component that generated the error. ErrorCode is a number indicating the type of error. See “Error handling” on page 705 for a list of the error codes.

The OnFaxFinish event is generated soon after the OnFaxError event and passes the same error code. The OnFaxFinish event is generated for both successful and failed transfers, so you may want to use it instead of an OnFaxError handler.

See also: OnFaxFinish

OnFaxFinish**event**

```
property OnFaxFinish : TFaxFinishEvent  
  
TFaxFinishEvent = procedure(  
    CP : TObject; ErrorCode : Integer) of object;
```

↳ Defines an event handler that is called when a fax session ends.

This event is generated at the end of each fax session, successful or not. If the session ends successfully, ErrorCode is zero. Otherwise, ErrorCode is a number indicating the type of error. For a list of the error codes, see “Error handling” on page 705. CP is the fax component whose session just finished.

It's important to note that a fax session may consist of more than one fax call. For example, if you implement an OnFaxNext event to send several faxes with one call to StartTransmit, all of those faxes are considered to be in the same fax session.

This handler could be used to display a completion dialog box (needed only if a fax status event handler is not used) or to allow scheduling another fax transfer.

The following example displays a message when a fax session finishes:

```
procedure TForm1.ApdSendFax1FaxFinish(  
    CP : TObject; ErrorCode : Integer);  
begin  
    ShowMessage('Fax finished: ' + ErrorMessage(ErrorCode));  
end;
```

```
property OnFaxLog : TFaxLogEvent

TFaxLogEvent = procedure(
    CP : TObject; LogCode : TFaxLogCode) of object;

TFaxLogCode = (lfaxNone, lfaxTransmitStart, lfaxTransmitOk,
    lfaxTransmitFail, lfaxReceiveStart, lfaxReceiveOk,
    lfaxReceiveSkip, lfaxReceiveFail);
```

↳ Defines an event handler that is called at designated points during a fax transfer.

The primary purpose of this event is to allow the logging of statistical information about fax transfers. For example, the transfer time and whether the transfer succeeded or failed could be logged. This event could also be used for startup and cleanup activities.

CP is the fax component to be logged. LogCode is a code that indicates the state of the fax transfer. The possible states are listed in “Fax logging” on page 712. No other information is passed with this event, but the fax status properties, such as FaxFile and PhoneNumber, can be used to get additional information about the fax session.

See also: FaxLog

OnFaxStatus

```
property OnFaxStatus : TFaxStatusEvent

TFaxStatusEvent = procedure(
    CP : TObject; First, Last : Boolean) of object;
```

↳ Defines an event handler that is called regularly during a file transfer.

This event is generated once per second during the entire fax session and after the completion of each major operation (e.g., incoming ring detected, remote station ID received). It can be used to update a status display that informs the user about the fax progress.

CP is the fax component that is in progress. A number of the properties of this component can be read to establish the status of the session. For a list of the properties, see “Fax status” on page 707. First is True on the first call to the handler, False otherwise. Last is True on the last call to the handler, False otherwise.

A predefined status component called TApdFaxStatus is supplied with Async Professional. If you don't want to write an OnFaxStatus event handler, you can use this standard fax status window. Just create an instance of TApdFaxStatus and assign it to the StatusDisplay property of the TApdSendFax or TApdReceiveFax component.

See also: StatusDisplay

PageLength

read-only, run-time property

```
property PageLength : LongInt
```

↳ The total number of bytes in the current page.

PageLength is valid only when you are sending a fax. When receiving a fax, the total size of the page is not known in advance, so PageLength is zero.

PageLength can be used by an OnFaxStatus event handler to get the total number of bytes in the current page. The appropriate time to check PageLength is when FaxProgress equals fpSendPage or fpGetPage. At other times, it is either zero or a value associated with the previous page.

See also: BytesTransferred, CurrentPage, TotalPages

RemoteID

read-only, run-time property

```
property RemoteID : TStationID
```

```
TStationID = string[20];
```

↳ The station ID of the remote fax machine.

RemoteID can be used by an OnFaxStatus event handler to get the station ID of the remote fax machine. The appropriate time to check RemoteID is when FaxProgress equals fpGotRemoteID. Before that, it returns an empty string.

See also: StationID

SessionBPS

read-only, run-time property

```
property SessionBPS : Word
```

↳ The negotiated transfer rate in bits per second.

SessionBPS can take on the values 14400, 12000, 9600, 7200, 4800, and 2400. Most faxmodems now support 9600 or higher. The fax connection process attempts to negotiate the highest possible rate unless you have set DesiredBPS to limit the highest rate.

SessionBPS can be used by an OnFaxStatus event handler to get the negotiated transfer rate. The appropriate time to check SessionBPS is when FaxProgress equals fpSessionParams. Before that, it is undefined.

Session parameters can change more than once during a single session. Be sure that your OnFaxStatus event handler updates its parameter display each time FaxProgress returns the value fpSessionParams.

See also: DesiredBPS, SessionECM, SessionResolution, SessionWidth

```
property SessionECM : Boolean
```

↳ Indicates whether automatic error correction is enabled.

SessionECM is True if automatic error correction is enabled for this transfer, or False if it isn't. Error correction is enabled if both modems support it and DesiredECM is True.

SessionECM can be used by an OnFaxStatus event handler to check for automatic error correction. The appropriate time to check SessionECM is when FaxProgress equals fpSessionParams. Before that, it is undefined.

Session parameters can change more than once during a single session. Be sure that your OnFaxStatus event handler updates its parameter display each time FaxProgress returns the value fpSessionParams.

See also: DesiredECM, SessionBPS, SessionResolution, SessionWidth

SessionResolution**read-only, run-time property**

```
property SessionResolution : Boolean
```

↳ Indicates whether the fax is high resolution or standard resolution.

SessionResolution is True for a high resolution fax transfer, or False for a standard resolution transfer. Async Professional automatically enables high resolution if it is sending an APF file that contains high resolution data, or if it is receiving a high resolution fax from a remote partner.

SessionResolution can be used by an OnFaxStatus event handler to check for the fax resolution. The appropriate time to check SessionResolution is when FaxProgress equals fpSessionParams. Before that, it is undefined.

Session parameters can change more than once during a single session. Be sure that your OnFaxStatus event handler updates its parameter display each time FaxProgress returns the value fpSessionParams.

See also: SessionBPS, SessionECM, SessionWidth

property SessionWidth : Boolean

Default: True

↳ Indicates whether the fax is normal or wide width.

If SessionWidth is True (the default), the fax is a standard width of 1728 pixels (about 8.5 inches). If SessionWidth is False, the fax width is 2048 pixels (about 10 inches).

SessionWidth can be used by an OnFaxStatus event handler to check the fax width. The appropriate time to check SessionWidth is when FaxProgress equals fpSessionParams. Before that, it is undefined.

Session parameters can change more than once during a single session. Be sure that your OnFaxStatus event handler updates its parameter display each time FaxProgress returns the value fpSessionParams.

See also: SessionBPS, SessionECM, SessionResolution

SoftwareFlow**property**

property SoftwareFlow : Boolean

Default: False

↳ Determines whether the fax components enable or disable software flow control during the fax session.

When using software flow control during a fax session, the flow control must be enabled and disabled at various points in the session. Because hardware flow control is more reliable, it is used by default and the fax components do not enable or disable software flow control. If you need to use software flow control, you must set SoftwareFlow to True.

For more information regarding flow control see “Fax sessions and the TAPdComPort” on page 702”.

```
property StationID : TStationID
```

```
TStationID = string[20];
```

↪ The station ID of the faxmodem.

A fax device can identify itself to another fax device with a 20 character name, called the station ID. The Class 1/1.0, Class 2, and Class 2.0 specifications indicate that the station ID should contain just a phone number; therefore, they limit it to just the digits 0 through 9 and space. However, the station ID is frequently used to store an alphabetic name. Most faxmodems support this convention by allowing upper and lower case letters, as well as other special characters in the station ID. This can cause problems for some fax machines, though, since they cannot print these characters.

Async Professional does not filter the characters stored in the station ID. If your software must be compatible with the broadest possible range of fax hardware, you might want to limit the characters stored in StationID.

This station ID is used on both incoming and outgoing calls.

A fax file stored in APF format also contains a station ID in the file header. This station ID is stored when a document is converted to APF format. For more information, see “TApdFaxConverter Component” on page 594.

See also: `TApdFaxConverter.StationID`

```
property StatusDisplay : TApdAbstractFaxStatus
```

↪ An instance of a fax status window.

If StatusDisplay is nil (the default), the fax does not provide an automatic status window. An OnFaxStatus event handler can be installed to display status in this case.

If you create an instance of a class derived from TApdAbstractFaxStatus or use the supplied TApdFaxStatus component (see page 826) and assign it to StatusDisplay, the status window is displayed and updated automatically.

See also: `OnFaxStatus`


```
property StatusInterval : Word
```

Default: 1

↳ The maximum number of seconds between OnFaxStatus events.

The OnFaxStatus event is generated for each major fax session event (connected, got station ID, and so on) and at intervals of StatusInterval seconds.

This property also determines how frequently the StatusDisplay window is updated.

See also: OnFaxStatus, StatusDisplay

```
function StatusMsg(const Status : Word) : string;
```

↳ Returns an English string for a fax status code.

This routine is intended primarily for use in fax status routines. It returns a status string from the string table resource linked into your EXE. The string ID numbers correspond to the values of the fpXxx constants (see “Fax status” on page 707). If the string table doesn't contain a string resource with the requested ID, an empty string is returned.

The returned string is never longer than MaxMessageLen (80) characters.

```
property SupportedFaxClasses : TFaxClassSet
```

```
TFaxClassSet = set of TFaxClass;
```

```
TFaxClass = (
    fcUnknown, fcDetect, fcClass1, fcClass1_0, fcClass2, fcClass2_0);
```

↳ The set of fax classes supported by the faxmodem.

SupportedFaxClasses is available only at run time because it sends commands to the faxmodem to determine what baud rates are supported (when it equals fcDetect).

Initially FaxClass is fcDetect, so that the first reference to it causes the faxmodem interrogation. Thereafter, references to SupportedFaxClasses return the known set of supported fax classes. The re-interrogation of the faxmodem can be forced by setting FaxClass to fcDetect.

Generally, applications should use the highest supported class: fcClass2_0, then fcClass2, and finally fcClass1.

See also: FaxClass

```
property TapiDevice : TApdCustomTapiDevice
```

↪ Determines an instance of a TAPI device.

This refers to a TAPI device that we may create internally when sending or receiving faxes. We check for an Assigned TapiDevice and will use it if there is an ApdTapiDevice on the form. EnableVoice property should be false by default and left that way.

See also: ComPort

```
property TotalPages : Word
```

↪ The total number of pages in the current fax.

TotalPages is valid only when you are sending a fax. When receiving a fax, the total number of pages is not known in advance, so TotalPages is zero.

TotalPages can be used by an OnFaxStatus event handler to get the total number of bytes in the current page. The appropriate time to check TotalPages is when FaxProgress equals fpSendPage or fpGetPage. Before that, it is either zero or a value associated with the previous page.

See also: BytesTransferred, CurrentPage, PageLength

TApdSendFax Component

The TApdSendFax component is used to send faxes. It builds on the services of TApdAbstractFax and implements the properties, events, and methods required to transmit faxes.

Specifying a fax to send

A *fax to send* consists of three items: an APF file (or list of APF files), a phone number, and an optional cover file. TApdSendFax provides a few mechanisms for specifying this information. When sending only one fax the simplest approach is to set the FaxFile, PhoneNumber, and CoverFile properties, then call StartTransmit. TApdSendFax dials the specified number, transmits the fax, and generates an OnFaxFinish event to indicate that it is finished.

When you need to send multiple fax files to the same location, you can fill the FaxFileList property (a stringlist) with the names of the files, set the PhoneNumber and CoverFile properties, then call StartTransmit. TApdSendFax concatenates the APF files into a single temporary APF, dials the specified number, transmits the fax, deletes the temporary APF, and generates an OnFaxFinish event to indicate that it is finished.

If you need to send a single fax file to multiple locations, you can use the OnFaxNext event to specify the files and phone numbers. If an OnFaxNext event handler is specified, TApdSendFax calls it instead of using FaxFile/FaxFileList, PhoneNumber, and CoverFile. The event handler is expected to return the fax file, phone number, and cover file of the next fax to send. If there are no more fax files to send, the event handler should return empty strings. The following code fragments illustrate how:

```
TForm1 = class(TForm)
...
private
    FaxList : TStringList;
    FaxIndex : Word;
end;
...
```

```

procedure TForm1.AddFilesClick(Sender : TObject);
begin
    FaxList.Add('260-7151^FILE1.APF');
    FaxList.Add('555-1212^FILE2.APF');
    FaxIndex := 0;
end;

procedure TForm1.ApdSendFax1FaxNext(
    CP : TObject; var APhoneNumber, AFaxFile,
    ACoverFile : OpenString);
var
    S : string;
    CaretPos : Byte;
begin
    try
        S := FaxList[FaxIndex];
        CaretPos := Pos('^', S);
        APhoneNumber := Copy(S, 1, CaretPos-1);
        AFaxFile := Copy(S, CaretPos+1, 255);
        ACoverFile := '';
        Inc(FaxIndex);
    except
        APhoneNumber := '';
        AFaxFile := '';
        ACoverFile := '';
    end;
end;

```

This example sends two fax files to two different locations: it sends FILE1.APF to the fax device at 260-7151 and sends FILE2.APF to 555-1212. No cover files are used.

When AddFilesClick is called (presumably from clicking the Add Files button), the two fax files and phone numbers are added to a TStringList named FaxList. FaxIndex, which is used to keep track of which fax file to send, is initialized to zero.

When TApdSendFax generates the OnFaxNext event, ApdSendFax1NextFax uses the next string from FaxList to extract the fax file name and phone number, returning them in the APhoneNumber and AFaxFile parameters. When there are no more faxes to send, FaxList generates an EListError exception. The except block then sets the passed parameters to empty strings, which tells TApdSendFax that there are no more faxes to send. An OnFaxFinish event is fired at the end of the fax session (after all faxes have been sent).

Cover pages

Fax transmissions often include a cover page, which provides basic information about the fax, usually consisting of the sender's name, the recipient's name, the total number of pages, and perhaps the date and time. Async Professional provides several options for sending this information. First, you can avoid a cover page altogether and put the same information on a fax page header. See the `HeaderLine` property on page 748 for details.

If you want to send a cover page, you have three options. First, you can build the cover page right into the main APF file for the document. This option applies to text documents only. Simply store the cover page text at the beginning of the document, insert a form feed character, and continue with the document itself. Then use the `TApdFaxConverter` component to generate the APF file.

Second, you can create a separate APF file that contains the cover page. Take this approach if the cover page contains special graphics or text that doesn't change between fax transmissions. Create the APF file and set the `CoverFile` property of `TApdSendFax` to the fully qualified name of the file or return the fully qualified file name in the `ACoverFile` parameter of the `OnFaxNext` event.

The third and most flexible approach is to put the cover page in a separate text file. If the cover file name does not have an extension of APF, it is assumed to be a text file. `TApdSendFax` automatically converts this to fax format (using either the built-in standard font or a Windows font) and transmits it to the remote machine.

What makes this approach more flexible is the fact that the text file can use replacement tags. A replacement tag is one of several characters prefixed with '\$'. As each line of the cover file is converted, the tags are replaced with appropriate text. A line in the cover file can consist of any mix of tags and normal text (be careful that your normal text doesn't happen to contain tags, though). Blank lines and spaces can be used to format the cover page. The available replacement tags are shown in Table 15.15.

Table 15.15: *Available Replacement Tags*

Tag	Description
\$D	Today's date in MM/DD/YY format, always 8 characters
\$I	Station ID, variable length
\$N	Total number of pages, variable length
\$P	Current page number, variable length
\$R	Recipient's name as contained in <code>HeaderRecipient</code> , variable length

Table 15.15: *Available Replacement Tags (continued)*

Tag	Description
\$F	Sender's name as contained in HeaderSender, variable length
\$S	Title as contained in HeaderTitle, variable length
\$T	Current time in HH:MMpm format, always 7 characters

Note that some of the tags vary in length. For example, \$P would be replaced by '1' for the first page and "10" for the tenth page.

By taking advantage of replacement tags, you can create a single cover page file that is used for every fax transfer. Below is a sample cover file and sample code that sets the replacement tags:

COVER.TXT:

Olympic Training Center
Colorado Springs, Colorado

DATE : \$D
TIME : \$T
FROM : \$F
TO : \$R

(form feed)

Program:

```
ApdSendFax.HeaderSender := 'Picabo';  
ApdSendFax.HeaderRecipient := 'Elizabeth';  
ApdSendFax.FaxFile := 'FILE1.APF';  
ApdSendFax.CoverFile := 'COVER.TXT';  
ApdSendFax.PhoneNumber := '260-7151';  
ApdSendFax.StartTransmit;
```

The file FILE1.APF is sent to 260-7151 using the cover page file COVER.TXT. The date and time tags are replaced with the current date and time. The sender's name is set to "Picabo" and the recipient's name is set to "Elizabeth."

Send fax events

OnFaxNext

```
procedure(  
    CP : TObject; var APhoneNumber : string; var AFaxFile : string;  
    var ACoverFile : string) of object;
```

Generated when it is time to transmit a fax (just after StartTransmit is called and after each fax is sent). When transmitting multiple faxes, your program must implement an OnFaxNext event handler to return, in sequence, each fax file to send. See EXFAXL for an example that shows how to use a TStringList component to maintain a list of fax file names and phone numbers.

☛ **Caution:** An OnFaxFinish event will not be fired until after empty strings are returned in your OnFaxNext event to signal the end of the fax session (all faxes have been sent).

Example

This example shows how to construct and use a fax send component. This example includes a TApdFaxStatus component (see page 826) so that you can see the progress of the fax session.

Create a new project, add the following components, and set the property values as indicated in Table 15.16.

Table 15.16: *Example components and property values*

Component	Property	Value
TApdComPort	ComNumber	<set as needed for your PC>
	OutSize	8192
TApdSendFax	FaxFile	<name of the APF file to send>
	PhoneNumber	<fax number to dial>
TApdFaxStatus		
TButton	Name	Send

The output buffer size of the comport component (the OutSize property of TApdComPort) is raised from its default value of 4096 to 8192. TApdSendFax requires an output buffer of at least 8192 bytes to ensure there is always enough room to add the next raster line to send, plus keep the output buffer full enough to avoid data underflow errors.

Double-click on the Send button's OnClick event handler within the Object Inspector and modify the generated method to match this:

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

This method starts a background fax transmit session, which dials the specified fax phone number and attempts to send the specified fax file.

The form includes a TApdFaxStatus component, which is automatically displayed by the fax and periodically updated to show the progress of the fax transfer.

This example is in the EXFAXS project in the \ASYNCPRO\EXAMPLES directory.

Hierarchy

TComponent (VCL)

❶ TApdBaseComponent (OOMisc)	8
TApdCustomAbstractFax (AdFax)	
❷ TApdAbstractFax (AdFax)	715
TApdCustomSendFax (AdFax)	
TApdSendFax (AdFax)	

Properties

❷ AbortNoConnect	DialRetryWait	❷ HangupCode
BlindDial	DialWait	HeaderLine
BufferMinimum	❷ ElapsedTime	HeaderRecipient
❷ BytesTransferred	EnhFont	HeaderSender
❷ ComPort	EnhHeaderFont	HeaderTitle
CoverFile	EnhTextEnabled	❷ InitBaud
❷ CurrentPage	❷ ExitOnError	MaxSendCount
❷ DesiredBPS	❷ FaxClass	❷ ModemBPS
❷ DesiredECM	❷ FaxFile	❷ ModemChip
Detect Busy	❷ FaxFileExt	❷ ModemECM
DialAttempt	FaxFileList	❷ ModemInit
DialAttempts	❷ FaxLog	❷ ModemModel
DialPrefix	❷ FaxProgress	❷ ModemRevision

- ② NormalBaud
- NoSoftwareFlow
- ② PageLength
- PhoneNumber
- ② RemoteID
- SafeMode
- ② SessionBPS
- ② SessionECM
- ② SessionResolution
- ② SessionWidth
- ② SoftwareFlow
- ② StationID
- ② StatusDisplay
- ② StatusInterval
- ② StatusMsg
- ② SupportedFaxClasses
- ToneDial
- ② TotalPages
- ① Version

Methods

- ② CancelFax
- ConcatFaxes
- ConvertCover
- StartManualTransmit
- StartTransmit
- ② StatusMsg

Events

- ② OnFaxError
- ② OnFaxFinish
- ② OnFaxLog
- OnFaxNext
- ② OnFaxStatus

Reference Section

BlindDial

property

property BlindDial : Boolean

Default: False

↳ Allows a fax to be sent regardless of whether the modem detects a dial tone.

If BlindDial is True, a different initialization sequence is sent to the modem before a fax is sent (ATX3 is sent instead of ATX4). This initialization sequence allows the modem to use a phone line, even if it can't detect a dial tone.

BufferMinimum

property

property BufferMinimum : Word

Default: 1000

↳ Defines the minimum number of bytes that must be in the output buffer before TApdSendFax yields control.

Once started, a fax transmit session must have a constant supply of data to transmit. Lack of data to transmit is referred to as a data underflow condition and usually results in a failed fax session. Because Windows is a multi-tasking environment, TApdSendFax tries to fill the output buffer as full as possible before yielding control. BufferMinimum is the minimum number of output bytes that must be in the output buffer before TApdSendFax yields.

If your program is operating among ill-behaved programs or other conditions that might result in long periods where it isn't given a chance to run, increasing BufferMinimum decreases the chance of a data underflow error.

An attempt to set BufferMinimum to more than the OutSize property of TApdComPort is ignored.

See also: MaxSendCount, SafeMode

```
procedure ConcatFaxes(FileName : ShortString);
```

↳ Combines multiple APF files into a single APF file.

This method can be used in conjunction with the FaxFileList property to combine multiple APF files into a single APF file. Simply the APF files you wish to combine to the FaxFileList property, and call ConcatFaxes with FileName set to the desired output filename. If FileName exists prior to calling ConcatFaxes, it will be overwritten without warning.

Note that this method is not required to send a list of files to a single destination, since APF files in the FaxFileList property will automatically be combined when StartTransmit is called (assuming OnFaxNext is not implemented).

This method can be useful if you need to send a concatenated fax to multiple phone numbers (using the OnFaxNext event) or if you want to retain the concatenated fax following the transmission.

See also: FaxFileList, OnFaxNext

ConvertCover**method**

```
procedure ConvertCover(const InCover, OutCover : string);
```

↳ Converts a cover file to APF using replaceable tags.

The ConvertCover method converts an ASCII text file into an APF file suitable for faxing. This method supports the replaceable tags supported by the TApdFaxConverter. See “Cover pages” on page 738 for a description of the replaceable tags.

InCover is the pathname of the cover file to convert, OutCover is the pathname of the APF to create.

CoverFile**property**

```
property CoverFile : string
```

↳ The name of the cover file to send.

If you are sending a single fax with a cover sheet, set CoverFile to the name of the text or APF file to be used as the cover sheet. If you are sending multiple fax files, you must implement an OnFaxNext event handler. CoverFile is automatically set to the cover file name returned by your event handler.

If the extension of the cover file is “APF”, the file is assumed to be an APF file and is transmitted as-is, before FaxFile is transmitted. Otherwise the cover file is assumed to be a text file. Each text line is read from the file, replacement tags are replaced with appropriate text, and the line is converted to a series of raster lines and transmitted to the remote fax. After the cover file is converted and transmitted, the FaxFile is transmitted.

CoverFile can be used by status and logging routines to return the name of the current cover file.

See “Cover pages” on page 738 for more information.

See also: OnFaxNext, PhoneNumber, TAPdAbstractFax.FaxFile

DetectBusy property

property DetectBusy : Boolean

Default: DefDetectBusy (True)

↳ Enables or disables busy signal detection.

If DetectOnBusy is True, the busy signal is detected if the receiving fax is already off hook. An ErrorCode of ecFaxBusy will be passed in the OnFaxError event. If False, a different initialization sequence is sent to the modem before a fax is sent (ATX2 is sent instead of ATX4). This initialization sequence disables the modem’s capability to detect a busy signal.

See also: OnFaxError

DialAttempt read-only, run-time property

property DialAttempt : Word

↳ Indicates the number of times the current fax number has been dialed.

If the dialed fax number is busy, TAPdSendFax waits briefly and calls the number again. It tries up to DialAttempts times. The DialAttempt property returns the number of the current attempt. DialAttempt is incremented immediately upon encountering a busy line.

See also: DialAttempts, DialRetryWait

DialAttempts

property

```
property DialAttempts : Word
```

Default: 3

↳ Determines the number of times TApdSendFax automatically dials a fax number.

This is the number of times a fax session is attempted, it is not the number of retries. When DialAttempts is one, for example, the fax number is dialed only once. If the line is busy, it is not tried again. When DialAttempts is three, the fax number is dialed a maximum of three times.

See also: DialAttempt, DialRetryWait

DialPrefix

property

```
property DialPrefix : TModemString
```

```
TModemString = string[40];
```

↳ The optional dial prefix.

DialPrefix specifies an optional dial prefix that is inserted in the dial command between “ATDT” and the number to dial. If your telephone system requires special numbers or codes when dialing out, you can specify them once here rather than in every fax number.

Do not include “ATD” or a ‘T’ or ‘P’ tone/pulse modifier in the dial prefix. “ATD” is automatically prefixed by StartTransmit and the ‘T’ or ‘P’ is controlled by ToneDial.

See also: ToneDial

DialRetryWait

property

```
property DialRetryWait : Word
```

Default: 60

↳ The number of seconds to wait after a busy signal before trying the number again.

After encountering a busy signal, TApdSendFax checks to see if it should try this number again by comparing DialAttempts to DialAttempt. If more attempts are required, it first waits DialRetryWait seconds before dialing again to give the dialed fax machine time to complete the current session.

If no more dialing attempts are required, TApdSendFax does not wait, but immediately progresses to the next state, which is logging the current fax session as failed.

See also: DialAttempt, DialAttempts

DialWait

property

property DialWait : Word

Default: 60

↪ The number of seconds to wait for a connection after dialing the number.

This property determines how many seconds to wait after dialing the receiver's phone number. If the receiver does not answer within this time, the OnFaxFinish event will fire and return an error code of ecFaxNoAnswer.

See also: OnFaxServerFinish

EnhFont

property

property EnhFont : TFont

↪ Determines the font used to convert cover pages.

If EnhTextEnabled is True, the font specified by EnhFont is used by TApdSendFax to convert the cover page. Any font available to Windows can be used (double click on the property to invoke the font dialog and see a list of the fonts). Only one font can be used for a document (i.e., font sizes and types cannot be mixed within a single cover page).

There is an upper limit on the size of the font, but this limit is not typically reached unless a very large font is used (e.g., greater than 72 pt). If the limit is exceeded, an ecEnhFontTooBig error occurs during the conversion process.

See also: EnhTextEnabled

EnhHeaderFont

property

property EnhHeaderFont : TFont

↪ Determines the font used to convert the fax header.

If EnhTextEnabled is True, the font specified by EnhHeaderFont is used by TApdSendFax to convert the fax header. Any font available to Windows can be used (double click on the property to invoke the font dialog and see a list of the fonts).

There is an upper limit on the size of the font, but this limit is not typically reached unless a very large font is used (e.g., greater than 72 pt). If the limit is exceeded, an ecEnhFontTooBig error occurs during the conversion process.

See also: EnhTextEnabled

EnhTextEnabled

property

```
property EnhTextEnabled : Boolean
```

Default: False

↳ Determines whether TAPdSendFax uses the default font.

If `EnhTextEnabled` is `True`, the enhanced text-to-fax converter is used by `ApdSendFax` when converting fax headers and text cover pages. This means that the font specified by `EnhFont` is used to convert the cover page and the font specified by `EnhHeaderFont` is used to convert the fax header.

The converter makes no attempt to keep all text on the page when the size of the font is changed. You must ensure that the cover page line length and document length fit on the page in the desired font.

See also: `EnhFont`, `EnhHeaderFont`

FaxFileList

property

```
property FaxFileList : TStringList
```

↳ Defines a list of APF files to concatenate into a single APF file.

If there is more than one APF file in `FaxFileList` when `StartTransmit` is called, the files in the list will be combined together into a single, temporary file for transmission. The temporary APF is deleted when the fax session is completed—whether the session was successful or not.

As with the `FaxFile` property, this property is ignored if an `OnFaxNext` event is implemented.

See also: `ConcatFaxes`, `OnFaxFinish`, `StartTransmit`

HeaderLine

property

```
property HeaderLine : string
```

↳ The optional line of text that is sent at the top of each fax page.

A header line consists of normal text and replacement tags. A replacement tag is one of several characters prefixed with ‘\$’. When the header line is transmitted, the tags are replaced with appropriate text. The available replacement tags are listed in “Cover pages” on page 738.

No check is made to make sure your header line fits on a page. If your header line does not fit, it is truncated when it is transmitted. Using the default fonts, you can fit approximately 144 characters on a standard width page. If `EnhTextEnabled` is `True`, you will have to experiment to see how many characters fit on the width of the page.

- ☛ **Caution:** Recently passed United States legislation makes it unlawful to send faxes within the United States without showing certain sender information on the fax. The new requirement states, in part: “It shall be unlawful for any person within the United States to use any computer or other electronic device to send any message via facsimile machine unless such message clearly contains, in a margin at the top or bottom of each transmitted page or on the first page of the transmission, the date and time it is sent and an identification of the business, other entity, or individual sending the message and the telephone number of the sending machine of such business, other entity, or individual.”

See also: `EnhHeaderFont`, `EnhTextEnabled`, `HeaderRecipient`, `HeaderSender`, `HeaderTitle`

HeaderRecipient **property**

```
property HeaderRecipient : string
```

- ☛ The fax recipient's name.

This string replaces the `$R` replacement tag in a cover page text file or a header line.

See “Cover pages” on page 738 for more information and examples.

See also: `HeaderLine`, `HeaderSender`, `HeaderTitle`

HeaderSender **property**

```
property HeaderSender : string
```

- ☛ The fax sender's name.

This string replaces the `$F` replacement tag in a cover page text file or a header line.

See “Cover pages” on page 738 for more information and examples.

See also: `HeaderLine`, `HeaderRecipient`, `HeaderTitle`

```
property HeaderTitle : string
```

↳ The fax title.

This string replaces the \$\$ replacement tag in a cover page text file or a header line.

See “Cover pages” on page 738 for more information and examples.

See also: HeaderLine, HeaderRecipient, HeaderSender

```
property MaxSendCount : Word
```

Default: 50

↳ Determines the maximum number of raster lines TApdSendFax sends before yielding control.

MaxSendCount prevents TApdSendFax from completely taking over the CPU. It provides a balance to BufferMinimum back towards sharing the CPU among all tasks. MaxSendCount overrides BufferMinimum and forces TApdSendFax to yield after sending MaxSendCount raster lines, even if the output buffer contains less than BufferMinimum bytes. The risk, of course, is that yielding too soon may result in a data underflow error.

The default values for BufferMinimum and MaxSendCount provide the best combination of cooperative multitasking and avoidance of data underflow. You should not alter these values unless fax sessions are failing due to data underflow errors.

See also: BufferMinimum, SafeMode

```
property OnFaxNext : TFaxNextEvent

TFaxNextEvent = procedure(CP : TObject;
  var APhoneNumber : TPassString; var AFaxFile : TPassString;
  var ACoverFile : TPassString) of object;

TPassString = string[255];
```

↳ Defines an event handler that returns the phone number, fax file, and cover file for the next fax.

If no handler is installed for this event, TApdSendFax dials the number specified by the PhoneNumber property and sends the fax(es) specified in the FaxFile or FaxFileList property. If the OnFaxNext event handler is installed, that event will be generated once the faxmodem has been initialized. The PhoneNumber, FaxFile and FaxFileList properties are ignored.

CP is the fax component that is transmitting. The event handler should return the next number to dial in APhoneNumber, the next fax file to send in AFaxFile, and an optional cover file name in ACoverFile. If a cover file isn't used, ACoverFile should be set to an empty string.

The event handler should return empty strings for APhoneNumber, AFaxFile, and ACoverFile when there are no more faxes to send.

An OnFaxFinish event will be fired at the end of the fax session (all faxes have been sent—and you have returned empty strings in your OnFaxNext handler to signal the end of the session).

See “Specifying a fax to send” on page 736 for more information.

PhoneNumber

property

```
property PhoneNumber : string
```

↳ Specifies the number to dial.

If you are sending a single fax, set `PhoneNumber` to the number to dial. If you are sending multiple fax files, you must implement an `OnFaxNext` event handler. `PhoneNumber` is automatically set to the phone number returned by your event handler.

If the phone system requires prefix codes (like '9'), the codes must be specified in `PhoneNumber` or in `DialPrefix`.

`PhoneNumber` can be used with status and logging routines to return the phone number dialed for the current fax session.

See also: `CoverFile`, `DialPrefix`, `OnFaxNext`, `TApdAbstractFax.FaxFile`

SafeMode

property

```
property SafeMode : Boolean
```

Default: True

↳ Determines whether `TApdSendFax` should yield during time-critical handshaking periods.

At the beginning and end of every fax page, `TApdSendFax` performs time-critical handshaking with the receiving fax device, where tolerance for delays is very low. Delays as small as a few hundred milliseconds can cause the receiver to believe the page transfer failed.

When `SafeMode` is True (the default), `TApdSendFax` does not yield during these periods. While this is the safest possible mode of operation for `TApdSendFax`, it will result in apparent brief periods of unresponsiveness to the user. That is, they won't be able to switch to another application or cancel the fax transfer until the critical handshaking is over. Fortunately, the time-critical period lasts only 1-3 seconds, so this should rarely be a problem.

StartManualTransmit

method

```
procedure TApdSendFax.StartManualTransmit;
```

↳ Begins transmitting a fax over an existing call.

`StartManualTransmit` is called during an existing call to send a fax.

See `ExFaxOD` and `ExFoDs` for examples.

See also: `StartTransmit`

```
procedure StartTransmit;
```

↳ Starts transmitting faxes in the background.

The steps leading up to calling StartTransmit are:

1. Create a port component.
2. Create a fax component.
3. Set PhoneNumber and FaxFile or provide an OnFaxNext event handler to return this information.
4. Write other event handlers for fax events.
5. Call StartTransmit.

StartTransmit returns immediately and transmits fax files in the background, occasionally generating events to keep you apprised of the progress. When the fax is finished, either successfully or with a fatal error, it generates an OnFaxFinish event.

The TAPI/Fax integration with receiving faxes will wait for incoming faxes using a passive answering mode. If TapiDevice is assigned, the associated TApdTapiDevice will be placed in AutoAnswer mode. When AnswerOnRing ring signals are detected, the TApdReceiveFax will answer the call. What this means for you is that you can be waiting for incoming faxes then go ahead and send a fax over the same device. After the fax is finished sending, TAPI will go back to passively waiting for an incoming fax.

See also: OnFaxNext, TApdAbstractFax.TapiDevice, TApdAbstractFax.OnFaxFinish

ToneDial**property**

```
property ToneDial : Boolean
```

Default: True

↳ Determines whether tone or pulse dialing is used for fax transmissions.

If ToneDial is True (the default), tone dialing is used. Otherwise, pulse dialing is used. Setting ToneDial does not immediately issue a modem command, but determines whether 'T' or 'P' is added to the dial command later.

TApdReceiveFax Component

The `TApdReceiveFax` component is used to receive faxes. It builds on the services of `TApdAbstractFax` and implements the properties, events and methods required to receive faxes.

Accepting fax files

The `OnFaxAccept` event gives your program the opportunity to accept or reject an incoming fax. The `OnFaxAccept` event is generated as soon as `TApdReceiveFax` establishes a fax session and determines the station ID of the sender.

There aren't many reasons to refuse an incoming fax, but there might be some that are important to your application. For example, if your automated fax receive system knows that there is not enough disk space to hold even a small incoming fax, you can save the sender's money by refusing the fax instead of accepting part of it and failing later.

Another possibility is to reject junk faxes automatically. You could maintain a list of known-good station IDs whose faxes you always accept, or a list of known-bad station IDs whose faxes you always reject.

If you don't implement an `OnFaxAccept` event then all faxes are accepted.

The following event handler accepts faxes only from station ID "719-260-7151":

```
procedure TForm1.ApdReceiveFax1FaxAccept(  
    CP : TObject; var Accept : Boolean);  
begin  
    Accept := ApdReceiveFax1.RemoteID = '719-260-7151';  
end;
```

Naming incoming fax files

When a fax is received, a file name must be chosen for storing the incoming image. Unlike a file transfer protocol, the fax sender does not provide any name.

`TApdReceiveFax` provides two methods for generating fax file names. The `FaxNameMode` property specifies how the name should be automatically generated. The `OnFaxName` event gives your program the opportunity to return a file name.

`FaxNameMode` provides the following choices shown in Table 15.17.

Table 15.17: *FaxNameMode* options

Option	Result
fnNone	Names the file "NONAME.APF".
fnMonthDay	Names the file mmddnnnn.APF, where mm is the current month, dd is the current day, and nnnn is a sequential number (starting at 0001) for the number of files received this day.
fnCount	Names the file FAXnnnn.APF, where nnnn is a sequential number (starting at 0001) that is the first free number for the current directory.

For both `fnMonthDay` and `fnCount`, the time required to find a usable file name depends on the number of existing files with the same name format. The time is probably not an issue as long as there are fewer than 100 matching files. If more than 9999 files are received, the name "NONAME.APF" is used.

To avoid the delay of finding the next available file name or if to use a different algorithm for naming the fax files, write an event handler for the `OnFaxName` event. This event is generated by `TApdReceiveFax` during a fax session as soon as the incoming fax is accepted in the `OnFaxAccept` event.

The following event handler generates sequence numbers internally rather than scanning the output directory to obtain a sequence number:

```
const
  LastNumber : Word = 0;

procedure TForm1.ApdReceiveFax1FaxName(
  CP : TObject; var Name : OpenString);
begin
  if LastNumber < 10000 then begin
    Inc(LastNumber);
    Str(LastNumber, Name);
    while Length(Name) < 4 do
      Name := '0'+Name;
    Name := 'FAX'+Name+'.'+ApdReceiveFax1.FaxFileExt;
  end else
    Name := 'NONAME.APF';
end;
```

This example keeps track of sequence numbers internally, using the typed constant `LastNumber`. Of course, this overwrites existing files each time the program is restarted. `FaxFileExt` is used to change the file extension from the default of "APF". See page 736 for more information.

Example

This example shows how to construct and use a fax receive component. This example includes a TApdFaxStatus component (see on page 826) so that you can see the progress of the fax session.

Create a new project, add the following components, and set the property values as indicated in Table 15.18.

Table 15.18: *Fax receive component example*

Component	Property	Value
TApdComPort	ComNumber	<set as needed for your PC>
	InSize	8192
TApdReceiveFax		
TApdFaxStatus		
TButton	Name	Receive

The input buffer size of the comport component (the InSize property of TApdComPort) is raised from its default value of 4096 to 8192. Although this is not essential, it provides a buffer against programs that don't yield frequently.

Double-click on the Receive button's OnClick event handler within the Object Inspector and modify the generated method to match this:

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

This method starts a background fax receive session, which initializes the modem and prepares it to receive faxes, then calls StartReceive to start waiting for incoming faxes.

When a fax call arrives, TApdReceiveFax answers the phone, validates the call as a fax call, and then begins receiving the fax data. The form includes a TApdFaxStatus component, which is automatically displayed by the background fax process and periodically updated to show the progress of the fax transfer.

This example is in the EXFAXR project in the \ASYNCPRO\EXAMPLES directory.

Hierarchy

TComponent (VCL)

- 1 TapdBaseComponent (OOMisc) 8
 - TapdCustomAbstractFax (AdFax)
 - 2 TapdAbstractFax (AdFax) 715
 - TapdCustomReceiveFax (AdFax)
 - TapdReceiveFax (AdFax)

Properties

2 AbortNoConnect	2 FaxFileExt	2 PageLength
AnswerOnRing	2 FaxLog	2 RemoteID
2 BytesTransferred	FaxNameMode	2 SessionBPS
2 ComPort	2 FaxProgress	2 SessionECM
ConstantStatus	2 HangupCode	2 SessionResolution
2 CurrentPage	2 InitBaud	2 SessionWidth
2 DesiredBPS	2 ModemBPS	2 SoftwareFlow
2 DesiredECM	2 ModemChip	2 StationID
DestinationDir	2 ModemECM	2 StatusDisplay
2 ElapsedTime	2 ModemInit	2 StatusInterval
2 ExitOnError	2 ModemModel	2 SupportedFaxClasses
FaxAndData	2 ModemRevision	2 TotalPages
2 FaxClass	2 NormalBaud	
2 FaxFile	OneFax	

Methods

2 CancelFax	PrepareConnectInProgress	StartReceive
InitModemForFaxReceive	StartManualReceive	2 StatusMsg

Events

OnFaxAccept	2 OnFaxFinish	OnFaxName
2 OnFaxError	2 OnFaxLog	2 OnFaxStatus

Reference Section

AnswerOnRing

property

```
property AnswerOnRing : Word
```

Default: 1

↳ Determines the number of rings before a call is answered.

AnswerOnRing is the number of “RING” responses allowed before the incoming call is answered. The default is one ring. Values less than or equal to zero are treated the same as one ring.

ConstantStatus

property

```
property ConstantStatus : Boolean
```

Default: False

↳ Determines whether status events are generated as soon as StartReceive is called.

When transmitting faxes, the time to display status events is clear. While there are faxes to transmit, status should be displayed; when there are no more faxes to transmit, the fax session is over. The issue is less clear when receiving faxes because TApdReceiveFax is often waiting for faxes with no real status information to display.

If ConstantStatus is False (the default), the first status event is generated when an incoming ring is detected. The last status event is generated at the conclusion of the receipt of the fax. While TApdReceiveFax is waiting for the next incoming call, no status events are generated. Status events are generated again when the next incoming ring is detected. This continues until CancelFax is called or a fatal error occurs.

If ConstantStatus is True, the first status event is generated as soon as StartReceive is called. The last status event is generated only after CancelFax is called or a fatal error occurs.

See also: StartReceive, TApdAbstractFax.OnFaxStatus

DestinationDir

property

property DestinationDir : string

Default: Empty string

↪ Determines the directory for incoming fax files.

There are two ways to indicate the directory for storing incoming fax files. If an OnFaxName event handler is used, it can return the fully qualified name (i.e., including drive and directory). If an OnFaxName event handler does not return the fully qualified name, or one of the automatic fax naming methods is used, DestinationDir can be set to the name of the desired directory.

If DestinationDir is empty and the OnFaxName event handler doesn't return a fully qualified file name, incoming files are stored in the current directory.

See also: OnFaxName

FaxAndData

property

property FaxAndData : Boolean

Default: False

↪ Specify whether a compatible faxmodem will answer data calls.

If FaxAndData is True, the faxmodem is configured to answer both fax and data calls. If it is False, only fax calls are answered.

Not all faxmodems support this feature and there is no broad standard for enabling it. It is available only in Class 2 and 2.0 faxmodems. The modem must claim "adaptive answer" or a similar term in its list of features. It must accept the "AT+FAA=1" sequence that Async Professional sends to enable the feature.

To use this feature, you should first set FaxAndData to True. Then call InitModemForFaxReceive to initialize the modem for receiving either fax or data calls. Do not call StartReceive. Instead, wait for incoming calls in your program. When a call arrives and you issue the "ATA" command, the faxmodem will detect whether the call is data or fax.

If the call is data, the modem responds with "CONNECT." If you detect this string, your program should proceed as it normally does for a data call.

If the call is fax, the modem responds with “CED”, “FAX”, or “+FCON”. If you detect any of these strings, call `PrepareConnectInProgress` to tell `TApdReceiveFax` that the call has already been answered, then call `StartReceive`.

This area of faxmodem behavior is not standardized. You will need to experiment with your brand of faxmodem to find a method that works for you.

See the EXADAPT example program for an example.

See also: `PrepareConnectInProgress`

FaxNameMode property

```
property FaxNameMode : TFaxNameMode
TFaxNameMode = (fnNone, fnCount, fnMonthDay);
```

Default: `fnCount`

- ↳ Determines how an incoming fax is named if you don't assign an `OnFaxName` event handler.

`TApdReceiveFax` must assign a file name to an incoming fax. If you do not assign an `OnFaxName` event handler, `TApdReceiveFax` generates a name based on the value of `FaxNameMode`. The modes are described in “Naming incoming fax files” on page 754.

InitModemForFaxReceive method

```
procedure InitModemForFaxReceive;
```

- ↳ Initializes a faxmodem for receiving faxes.

This method sends appropriate “AT” commands to the faxmodem to prepare it for receiving faxes. It first sends any string you have specified using `ModemInit`. Next it sends the `DefInit` string (“ATE0Q0V1X4S0=0”). Then it enables Class 1, Class 1.0, Class 2, or Class 2.0 operation, either by autodetecting the highest class supported by the modem or using the last class specified by `FaxClass`.

For Class 2 and 2.0 modems `InitModemForFaxReceive` also sends strings for setting the station ID (see the `StationID` property on page 621), transfer rate (see the `DesiredBPS` property on page 719), error correction options (see the `DesiredECM` property on page 720), and “fax and data” option (see the `FaxAndData` property on page 759). Hence, these properties must be set before calling `InitModemForFaxReceive`. For Class 1 and Class 1.0 modems, these features are set while negotiating the connection, so you can set the relevant properties any time before calling `StartReceive`.

The critical portions of `InitModemForFaxReceive` are performed automatically whenever you call `StartReceive`, so you don't need to call it yourself in most situations. However, it must be called again whenever the modem has been used for any purpose besides faxing. For example, you should call it whenever the modem has been used as a data modem.

`InitModemForFaxReceive` is also necessary when you are answering both data and fax calls on the same line. See `FaxAndData` for details.

See also: `FaxAndData`, `TApdAbstractFax.DesiredBPS`, `TApdAbstractFax.DesiredECM`, `TApdAbstractFax.FaxClass`, `TApdAbstractFax.ModemInit`, `TApdAbstractFax.StationID`

OneFax **property**

`property OneFax : Boolean`

Default: False

↳ Enables or disables “one fax” receive behavior.

If `OneFax` is `True`, the background fax process stops after receiving one fax. If `OneFax` is `False` (the default), the background fax process waits for faxes until a fatal error occurs or `CancelFax` is called.

The most likely use of “one fax” behavior is in conjunction with `FaxAndData` and `PrepareConnectInProgress`. These are used when your program handles incoming calls itself and passes control to the fax routines only after a fax call is detected.

See also: `FaxAndData`, `PrepareConnectInProgress`, `TApdAbstractFax.CancelFax`

OnFaxAccept **property**

`property OnFaxAccept : TFaxAcceptEvent`

`TFaxAcceptEvent = procedure(
 CP : TObject; var Accept : Boolean) of object;`

↳ Defines an event handler that is called at the beginning of the receive fax session after the station ID of the caller is received.

This event provides an opportunity to accept or reject an incoming fax. If an `OnFaxAccept` event handler is not provided, all faxes are accepted.

When this event is generated, the only information known about the incoming fax is the station ID. So, the typical application for this event is to accept or reject faxes based on the station ID. For example, you can avoid receiving junk faxes by accepting only faxes coming from known station IDs.

See “Accepting fax files” on page 754 for more information.

```
property OnFaxName : TFaxNameEvent

TFaxNameEvent = procedure(
    CP : TObject; var Name : TPassString) of object;

TPassString = string[255];
```

↳ Defines an event handler that is called to return a file name for a fax.

The fax protocol doesn't include file name information, so the receiving software must generate file names (see FaxNameMode) or you can generate the file name in an OnFaxName event handler.

See "Naming incoming fax files" on page 754 for more information.

PrepareConnectInProgress**method**

```
procedure PrepareConnectInProgress;
```

↳ Forces TApdReceiveFax to pick up a connection in progress.

PrepareConnectInProgress is intended to be used when your program answers calls that can be either fax or data. When your program detects an incoming fax, it should call PrepareConnectInProgress and then call StartReceive, which will start processing at the point where an incoming call has already been answered. This approach can be used only if your faxmodem supports "adaptive answer." This term is used by modem manufacturers to indicate the ability to discriminate between fax and data calls.

See also: FaxAndData

StartManualReceive**method**

```
procedure TApdCustomReceiveFax.StartManualReceive(
    SendATAToModem : Boolean);
```

↳ Begins receiving a fax immediately.

StartManualReceive is called to begin receiving a fax over an existing call or over a new call that has not been answered.

SendATAToModem determines if the "ATA" answer command is sent to the modem to begin receiving the file. If SendATAToModem is True, a fax will be received over a call that has not yet been answered. If SendATAToModem is False, a fax will be received over an existing call.

Refer to ExFasOD for an example.

See also: StartReceive

```
procedure StartReceive;
```

↳ Starts waiting for and receiving faxes in the background.

The steps leading up to calling StartReceive look something like this:

1. Create a port component.
2. Create a fax component.
3. Write appropriate event handles for fax events.
4. Call InitModemForFaxReceive.
5. Call StartReceive.

StartReceive returns immediately. TApdReceiveFax waits for and receives faxes in the background, occasionally generating the various OnFaxXxx events. When the receive fax session is finished, either successfully or with a fatal error, it generates an OnFaxFinish event.

The TAPI/Fax integration with receiving faxes will wait for incoming faxes using a passive answering mode. If TapiDevice is assigned, the associated TApdTapiDevice will be placed in AutoAnswer mode. When AnswerOnRing ring signals are detected, the TApdReceiveFax will answer the call. What this means for you is that you can be waiting for incoming faxes then go ahead and send a fax over the same device. After the fax is finished sending, TAPI will go back to passively waiting for an incoming fax.

See also: TapiDevice, TApdAbstractFax.OnFaxXxx

Fax Server Components

The purpose of the Fax Server Components is to provide flexible, integrated fax reception, transmission, and queuing functionality.

The TApdFaxServer component provides the faxing engine and interfaces with the fax modem. This component can be used by itself to receive faxes, or with a TApdFaxServerManager to send faxes. The TApdFaxServer reproduces much of the functionality of the TApdSendFax and TApdReceiveFax components, so many of the properties are similar.

The TApdFaxServerManager component provides the TApdFaxServer component with fax jobs when requested. This component manages the fax jobs submitted by a TApdFaxServer or TApdFaxClient component by monitoring a specified directory for new fax jobs. This component also handles scheduling of fax jobs by inspecting the scheduled time of each fax job when a job is requested by the TApdFaxServer component.

The TApdFaxClient component provides the TApdFaxServerManager component with fax jobs. This component creates a fax job by creating a job file. The job file contains a header, entries for a single recipient or multiple recipients, cover page text, and the actual APF-formatted image data. The TApdFaxClient can be installed on a remote station to create fax jobs, then the job file just needs to be placed in the directory being monitored by the TApdFaxServerManager to enter the fax queue.

The fax server process

A fax server is a process that can receive and send faxes, often with the ability to schedule outbound faxes so they can be sent at specific times. Fax servers also provide a way to create fax jobs to be submitted to the faxing engine. The Async Professional Fax Server Components are divided into four components to handle these steps in a fax server:

- The TApdFaxJobHandler component handles fax job files.
- The TApdFaxServer component physically send and receives faxes.
- The TApdFaxServerManager component schedules fax jobs and manages the fax queue.
- The TApdFaxClient component creates fax jobs.

A TApdFaxServer component is required for each faxmodem that is to be used. A TApdFaxServerManager is required for each directory that is to be monitored for fax jobs. A TApdFaxClient is used for each station that submits fax jobs to the TApdFaxServerManager. Figure 15.2 illustrates one possible configuration (the dotted boxes represent optional components).

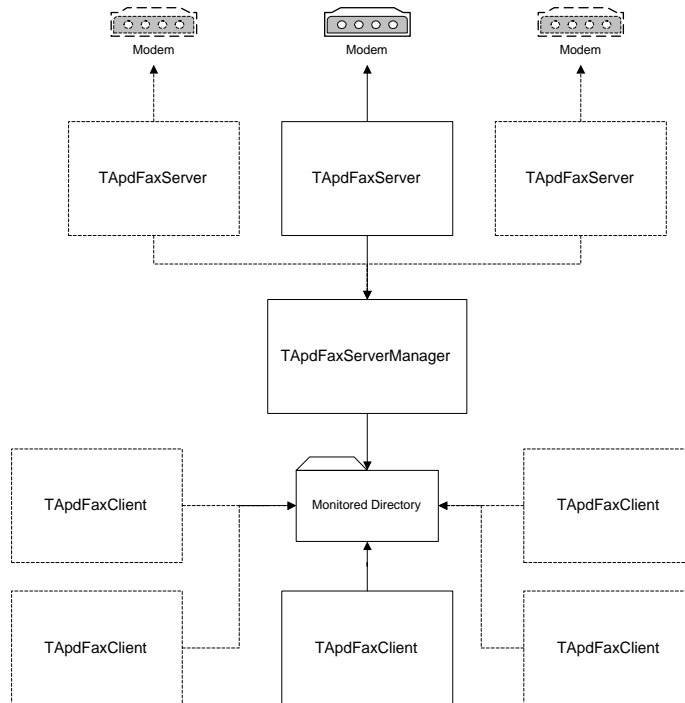


Figure 15.2: TApdFaxServerManger configuration.

In this scenario, there is one TApdFaxServer component for each modem to be used, one TApdFaxServerManager component monitoring a single directory, and several TApdFaxClient components feeding fax jobs to the monitored directory. The TApdFaxClients will create fax jobs, and copy them to the monitored directory. The TApdFaxServer will query the TApdFaxServerManager for fax jobs, the TApdFaxServerManager will find the jobs submitted by the TApdFaxClients, and will pass them to the TApdFaxServer. The TApdFaxServer will then process that job.

This scenario can be expanded upon in several ways to provide more flexible processing. The only limitation is that a directory can only be monitored by a single TApdFaxServerManager component.

Async Professional Job File format

The Fax Server Components cannot use the normal Async Professional Fax (APF) file format because APF does not provide enough flexibility for queuing or scheduling of multiple recipients. In order to support multiple recipients, scheduling, and ease of submission, the Fax Server Components use the Async Professional Job (APJ) file format. The APJ format contains a TFaxJobHeaderRec record, a TFaxRecipientRec record for each recipient, cover page text, and the APF data.

An APJ file is formatted as shown in Table 15.19.

Table 15.19: *APJ file format*

Fax job header
Recipient info
...
Recipient info
Cover page text
Regular APF format

The details of this format are most likely not needed for most applications, but the format is documented here in case you need to write APJ manipulation routines that are not provided with Async Professional.

The APJ file begins with a fax job header, which contains information that applies to the fax job as a whole. The TFaxJobHeaderRec adds 128 bytes to the size of the fax job file. Table 15.20 shows the fields of the TFaxJobHeaderRec.

Table 15.20: *TFaxJobHeaderRec fields*

Field	Purpose
ID	APRO fax job signature.
Status	Status of the fax job file.
JobName	Friendly name of the fax job.
Sender	Name of the sender.
SchedDT	TDateTime of the next scheduled job.
NumJobs	Number of recipient jobs in the APJ file.
NextJob	The index of the next job to send.

Table 15.20: *TFaxJobHeaderRec fields (continued)*

Field	Purpose
CoverOfs	The offset, in bytes, of the cover page text.
FaxHdrOfs	The offset, in bytes, of the TFaxHeaderRec.
Padding	42 bytes of extra data, leaving room for future expansion and forcing the size of the fax job header to 128 bytes.

The TFaxRecipientRec record follows the TFaxJobHeaderRec in the APJ file. Since there can be one, or several, recipients for a given APJ file, the space used will vary. 256 bytes are added to the size of the APJ for each recipient. Table 15.21 shows the fields of the TFaxRecipientRec.

Table 15.21: *TFaxRecipientRec fields*

Field	Purpose
Status	Status of this recipient info.
JobID	Unique ID for this job (usually the index number).
SchedDT	TDateTime this job is scheduled for.
AttemptNum	Number of times this fax has been attempted.
LastResult	The ErrorCode from the last attempt.
PhoneNumber	Phone number to dial for this fax job.
HeaderLine	Optional line of text sent at the top of each page.
HeaderRecipient	Recipient's name (\$R replaceable tag).
HeaderTitle	Fax title (\$S replaceable tag).
Padding	28 bytes of extra data, leaving room for future expansion and forcing the size of the fax job info to 256 bytes.

Following the TFaxRecipientRec structures in the APJ file is the cover page text data. Replaceable tags are supported, and are defined by the respective fields in the TFaxJobHeaderRec and TFaxRecipientRec records. Cover pages in APF format are not directly supported, but you can add an APF formatted cover page to the beginning of the fax file to have the same effect.

The APF file to be sent follows the cover page text.

Integration with other components

The `ApdFaxViewer`, `ApdSendFax`, `ApdReceiveFax`, `ApdFaxPrinter`, and other Async Professional fax components do not understand the APJ format, so the information must be extracted. The `TApdFaxServerManager` and `TApdFaxClient` descend from the `TApdFaxJobHandler` class, which contains methods to extract the APF data and cover file data from the APJ file. Use the `ExtractAPF` method to extract the APF fax data to a file, and the `ExtractCoverFile` method to extract the cover file data. The following example demonstrates how to use these two methods to display the APF data in a `TApdFaxViewer` and the cover text in a `TMemo`:

```
var
    FaxJobHandler : TApdFaxJobHandler;
begin
    FaxJobHandler := TApdFaxJobHandler.Create(nil);
    FaxJobHandler..ExtractAPF('C:\TEST.APJ', 'C:\TESTFAX.APF');
    ApdFaxViewer1.FileName := 'C:\TESTFAX.APF';
    if FaxJobHandler.ExtractCoverFile('C:\TEST.APJ',
        'C:\TESTCOVR.TXT') then
        Memo1.Lines.LoadFromFile('C:\TESTCOVR.TXT');
    FaxJobHandler.Free;
end;
```

TApdFaxJobHandler Component

The TApdFaxJobHandler component is the ancestor of the TApdFaxServerManager and TApdFaxClient components. This component provides methods to manipulate the Async Professional Job file format.

Hierarchy

TComponent (VCL)

❶ TApdBaseComponent (OOMisc)	8
TApdFaxJobHandler (AdFaxSrv)	

Properties

- ❶ Version

Methods

AddRecipient	GetJobHeader	ResetAPIStatus
CancelRecipient	GetRecipient	ResetRecipientStatus
ConcatFaxes	GetRecipientStatus	RescheduleJob
ExtractAPF	MakeJob	ShowFaxJobInfoDialog
ExtractCoverFile	ResetAPJPartials	

Reference Section

AddRecipient

method

```
procedure AddRecipient(  
    JobFileName : ShortString; RecipientInfo : TFaxRecipientRec);  
  
TFaxRecipientRec = packed record  
    Status : Byte;  
    JobID : Byte;  
    SchedDT : TDateTime;  
    AttemptNum : Byte;  
    LastResult : Word;  
    PhoneNumber : String[50];  
    HeaderLine : String[100];  
    HeaderRecipient : String[30];  
    HeaderTitle : String[30];  
    Padding : Array[228..256] of Byte;  
end;
```

↳ Adds new recipient information to an existing job file.

AddRecipient is used to add another recipient to an existing fax job file. JobFileName is the path and filename of the existing fax job file. RecipientInfo is the TFaxRecipientRec containing the information about this recipient. The following example will add a new recipient to an existing fax job file:

```
var  
    Recipient : TFaxRecipientRec;  
    FaxJobHandler : TApdFaxJobHandler;  
begin  
    Recipient.SchedDT := Now;  
    Recipient.PhoneNumber := '1 719 260 7151';  
    Recipient.HeaderLine := 'Fax to $R $D $T';  
    Recipient.HeaderRecipient := 'TurboPower';  
    Recipient.HeaderTitle := 'Info to TurboPower';  
    FaxJobHandler := TApdFaxJobHandler.Create(Self);  
    FaxJobHandler.AddRecipient('C:\FAXES\MyFax.APJ', Recipient);  
    FaxJobHandler.Free;  
end;
```

CancelRecipient

method

```
procedure CancelRecipient(JobFileName: ShortString; JobNum : Byte);
```

↪ CancelRecipient cancels a single recipient in a fax job file.

Use CancelRecipient to cancel a specific recipient in the fax job file. JobFileName is the name of the fax job file to modify. JobNum is the index of the recipient to cancel.

ConcatFaxes

method

```
procedure ConcatFaxes(  
    DestFaxFile: ShortString; FaxFiles: array of ShortString);
```

↪ Combines two APF files into a single APF file.

This method is used to concatenate multiple APF files into a single APF file. DestFaxFile is the file name that will contain the concatenated fax files. FaxFiles is an array containing the fax file names of the APF files to concatenate. The first fax file in the array will be at the beginning of the concatenated file, the second fax file in the array will follow the first, etc. If DestFileName already exists, it will be overwritten without warning.

The following example demonstrates how to use ConcatFaxes to concatenate several faxes (a cover page in APF format, DOC1..DOC5) into a new file (FAXFILE.APF):

```
var  
    ApdFaxJobHandler : TApdFaxJobHandler;  
begin  
    ApdFaxJobHandler := TApdFaxJobHandler.Create(Self);  
    try  
        ApdFaxJobHandler.ConcatFaxes(  
            'C:\FAXFILE.APF', [ 'C:\COVER.APF', 'C:\DOC1.APF',  
                                'C:\DOC2.APF', 'C:\DOC3.APF', 'C:\DOC4.APF',  
                                'C:\DOC5.APF' ] );  
    finally  
        ApdFaxJobHandler.Free;  
    end;  
end;
```

In C++Builder, the syntax used is usually not apparent, the following example shows how to do it:

```
{
    ShortString FaxFiles[6] = {
        "C:\\\\COVER.APF", "C:\\\\DOC1.APF", "C:\\\\DOC2.APF",
        "C:\\\\DOC3.APF", "C:\\\\DOC4.APF", "C:\\\\DOC5.APF"};
    ShortString DestFile = "C:\\\\FAXFILE.APF";
    TApdFaxJobHandler* ApdFaxJobHandler =
        new TApdFaxJobHandler(this);
    // the array index is 0-based, the array declaration is 1-based
    ApdFaxJobHandlerConcatFaxes(DestFile, FaxFiles, 5);
    delete ApdFaxJobHandler;
}
```

ExtractAPF

method

```
procedure ExtractAPF(JobFileName, FaxName : ShortString);
```

↳ Extracts the APF data from an APJ file.

Call the ExtractAPF method to extract the embedded APF data from within an APJ file. JobFileName is the file name of the APJ file that contains the APF. FaxName is the filename of the extracted APF file. This method is useful when you want to extract the APF data from an APJ file for use with another Async Professional fax component, such as the ApdFaxPrinter or ApdFaxViewer. The following example demonstrates how to extract the APF data and display it in a TApdFaxViewer:

```
var
    ApdFaxJobHandler : TApdFaxJobHandler;
begin
    ApdFaxJobHandler:= TApdFaxJobHandler.Create(nil);
    ApdFaxJobHandler.ExtractAPF(MyJob, APFFile);
    ApdFaxViewer.FileName := APFFile;
    ApdFaxJobHandler.Free;
end;
```

```
function ExtractCoverFile(  
    JobFileName, CoverName : ShortString) : Boolean;
```

↳ Extracts the cover file text from an APJ file.

Call the `ExtractCoverFile` method to extract the embedded cover page text from within an APJ file. `JobFileName` is the file name of the APJ file that contains the cover file text. `CoverName` is the file name of the extracted cover page text. If cover page data is not present in the APJ, this method will return `False`; if the APJ contains cover page text this method will return `True` and a new file (named `CoverName`) will be created. The following example demonstrates how to extract the cover page text and display it in a `TMemo`:

```
var  
    ApdFaxJobHandler : TApdFaxJobHandler;  
begin  
    ApdFaxJobHandler := TApdFaxJobHandler.Create(nil);  
    if ApdFaxJobHandler.ExtractCoverFile(MyJob, CoverFile) then  
        Memo.Lines.LoadFromFile(CoverFile);  
    ApdFaxJobHandler.Free;  
end;
```

```
procedure GetJobHeader(  
    JobFileName : ShortString; var JobHeader : TFaxJobHeaderRec);
```

↳ Returns the `TFaxJobHeaderRec` for the specified APJ file.

`GetJobHeader` returns the fax job header record for the APJ file. `JobFileName` is the file name of the APJ file. `JobHeader` is the `TFaxJobHeaderRec` record contained in `JobFileName`. The following example demonstrates how to extract the job header from an APJ file:

```
var  
    ApdFaxJobHandler : TApdFaxJobHandler;  
    JobHeader : TFaxJobHeaderRec;  
begin  
    ApdFaxJobHandler := TApdFaxJobHandler.Create(nil);  
    ApdFaxJobHandler.GetJobHeader(MyJob, JobHeader);  
    Labell.Caption := 'There are ' + IntToStr(JobHeader.NumJobs) +  
        ' recipients in this file';  
    ApdFaxJobHandler.Free;  
end;
```



```
function GetRecipient(JobFileName : ShortString;
  Index : Integer; var Recipient : TFaxRecipientRec) : Integer;
```

↳ Provides a TFaxRecipientRec for the specified APJ file.

Use GetRecipient to retrieve a TFaxRecipientRec record from an APJ file.

Since an APJ file can contain several TFaxRecipientRec records, you must specify the index of the record you want. JobFileName is the file name of the APJ file that contains the record you want. Index is the index of the recipient information to retrieve. Recipient will contain the recipient information. The return value will be ecOK if successful, or a non-zero value if unsuccessful. Use the ErrorMsg method in AdExcept.pas to convert the return value to a text string describing the error.

The following example will return the next scheduled job in an APJ and write some information about the job to a TListBox:

```
var
  ApdFaxJobHandler : TApdFaxJobHandler;
  Recipient : TFaxRecipientRec;
  JobHeader : TFaxJobHeaderRec;
  Count : Integer;
begin
  ApdFaxJobHandler := TApdFaxJobHandler.Create(nil);
  ApdFaxJobHandler.GetJobHeader(MyJob, JobHeader);
  Count := JobHeader.NextJob;
  ApdFaxJobHandler.GetRecipient('C:\TEST.APJ', Count, Recipient);
  ListBox.Items.Add('Job#' + IntToStr(Count));
  ListBox.Items.Add(' Scheduled for ' +
    DateTimeToStr(Recipient.SchedDT));
  ListBox.Items.Add(' To : ' + Recipient.HeaderRecipient);
  ListBox.Items.Add(' Fax# : ' + Recipient.PhoneNumber);
  ApdFaxJobHandler.Free;
end;
```

```
function GetRecipientStatus(
    JobFileName: ShortString; JobNum : Integer) : Integer;
```

↪ Returns the status flag for a specific recipient.

Use GetRecipientStatus to retrieve the status flag for a specific recipient. JobFileName is the name of the APJ to query. JobNum is the index of the recipient. The return value is one of the following values:

Name	Value	Description
stNone	0	Fax job not sent
stPartial	1	Fax job being sent
stComplete	2	Fax job sent
stPaused	3	This job is paused

MakeJob

method

```
procedure MakeJob(FaxFileName, CoverFile, JobName, Sender,
    JobFileName : ShortString; RecipientInfo : TFaxRecipientRec);
```

```
TFaxRecipientRec = packed record
    Status : Byte
    JobID : Byte;
    SchedDT : TDateTime;
    AttemptNum : Byte;
    LastResult : Word;
    PhoneNumber : String[50];
    HeaderLine : String[100];
    HeaderRecipient : String[30];
    HeaderTitle : String[30];
    Padding : Array[228..256] of Byte;
end;
```

↪ Creates a fax job file containing a single recipient.

Call MakeJob to create a file in the Async Professional Job format. FaxFileName is the name of the fax file in APF format. CoverFile is the name of the cover page file in text format. JobName is the friendly name of the fax job. Sender is the name of the sender.

JobFileName is the name of the resulting APJ file. RecipientInfo is the information about the recipient for this fax job. The following example demonstrates how to use MakeJob to create a fax file ready for submission to a TApdFaxServerManager:

```
var
  ApdFaxJobHandler : TApdFaxJobHandler;
  Recipient : TFaxRecipientRec;
begin
  FillChar(Recipient, SizeOf(TFaxRecipientRec), #0);
  Recipient.SchedDT := Now;
  Recipient.PhoneNumber := '1 719 471 9091';
  Recipient.HeaderLine := 'Fax to $R $D $T';
  Recipient.HeaderRecipient := 'TurboPower';
  Recipient.HeaderTitle := 'Info to TurboPower';
  ApdFaxJobHandler := TApdFaxJobHandler.Create(nil);
  ApdFaxJobHandler.MakeJob(
    'C:\DEFAULT.APF', 'C:\COVER.TXT', 'Info to TurboPower',
    'John Doe', 'C:\FAXES\INFO1.APJ', Recipient);
  ApdFaxJobHandler.Free;
end;
```

ResetAPJPartials

method

```
procedure ResetAPJPartials(JobFileName: ShortString);
```

↪ Resets all stPartial status flags to stNone.

This method is used to reset all recipients that have a status flag of stPartial. All recipient flags that are stPartial are changed to stNone. JobFileName is the name of the APJ to modify.

ResetAPJStatus

method

```
procedure ResetAPJStatus(JobFileName: ShortString);
```

↪ Resets all status flags in an APJ to stNone

This method is used to reset all recipient status flags to stNone, regardless of their current value. JobFileName is the name of the APJ to modify.

ResetRecipientStatus

method

```
procedure ResetRecipientStatus(  
    JobFileName: ShortString; JobNum, NewStatus: Byte);
```

↪ Resets individual recipient with new status, updates job header

This method is used to reset a specific recipient status flag to a new value. JobFileName is the name of the APJ to modify. JobNumb is the index of the recipient whose status flag should be changed. NewStatus is the value to which the status flag should be changed. Status flags can be one of the following values:

Name	Value	Description
stNone	0	Fax job not sent
stPartial	1	Fax job being sent
stComplete	2	Fax job sent
stPaused	3	This job is paused

RescheduleJob

method

```
procedure RescheduleJob(JobFileName : ShortString;  
    JobNum : Integer; NewSchedDT : TDateTime; ResetStatus: Boolean);
```

↪ Reschedules a recipient and updates the job header

The RescheduleJob method reschedules a specific recipient in an APJ to a new scheduled time. JobFileName is the name of the APJ to modify. JobNum is the index of the recipient to modify. NewSchedDT is the TDateTime representing the new scheduled date/time. ResetStatus determines whether the recipient's status flag is reset to stNone or not.

```
function ShowFaxJobInfoDialog(  
    JobFileName : ShortString) : TModalResult;
```

↳ Provides a dialog displaying fax job information.

Call the ShowFaxJobInfoDialog method to display a dialog containing information about the specified fax job file. JobFileName is the filename of the APJ file to show. The return value will be mrOK if OK was pressed. If the dialog was closed in some other manner, the return value will be mrCancel.

The TFaxJobHeaderRec information is displayed at the top of the dialog, and the TFaxRecipientRec information is displayed at the bottom. Since several TFaxRecipientRec structures can be in a single APJ file, navigation buttons are provided. Changes made to the editable controls can be applied without closing the dialog by clicking Apply. Accept the changes by clicking the OK, and cancel any changes by clicking Cancel. Note that changes that have been applied will be in effect even though the dialog box has been cancelled.

TApdFaxServer Component

The TApdFaxServer component is the faxing engine for the Fax Server Components. It 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. See “Receiving and Sending Faxes with TApdFaxServer” in the Developer’s Guide for step-by-step instructions.

Hierarchy

TComponent (VCL)	
● TApdBaseComponent (OOMisc)	8
TApdCustomAbstractFax (AdFax)	
TApdCustomFaxServer (AdFaxSrv)	
TApdFaxServer (AdFaxSrv)	

Properties

AnswerOnRing	EnhFont	ModemModel
BlindDial	EnhHeaderFont	ModemRevision
BufferMinimum	EnhTextEnabled	Monitoring
BytesTransferred	ExitOnError	NormalBaud
ComPort	FaxClass	PageLength
ConstantStatus	FaxFile	PrintOnReceive
CurrentPage	FaxFileExt	RemoteID
CurrentRecipient	FaxLog	SafeMode
CurrentJobFileName	FaxNameMode	SendQueryInterval
CurrentJobNumber	FaxPrinter	ServerManager
DelayBetweenSends	FaxProgress	SessionBPS
DesiredBPS	HangupCode	SessionECM
DesiredECM	InitBaud	SessionResolution
DestinationDir	MaxSendCount	SessionWidth
DialAttempt	ModemBPS	SoftwareFlow
DialAttempts	ModemChip	StationID
DialPrefix	ModemECM	StatusDisplay
DialWait	ModemInit	StatusInterval

SupportedFaxClasses
TapiDevice

ToneDial
TotalPages

❶ Version

Methods

CancelFax

ForceSendQuery

StatusMsg

Events

OnFaxServerAccept
OnFaxServerFatalError
OnFaxServerFinish

OnFaxServerName
OnFaxServerPortOpenClose
OnFaxServerLog

OnFaxServerStatus

Reference Section

AnswerOnRing

property

property AnswerOnRing : Word

Default: 1

- ↪ Determines the number of rings before a call is answered.

AnswerOnRing is the number of “RING” responses allowed before the incoming call is answered. The default is one ring. Values less than or equal to zero are treated the same as one ring.

BlindDial

property

property BlindDial : Boolean

Default: False

- ↪ Allows a fax to be sent regardless of whether the modem detects a dial tone.

If BlindDial is True, a different initialization sequence is sent to the modem before a fax is sent (ATX3 is sent instead of ATX4). This initialization sequence allows the modem to use a phone line, even if it can't detect a dial tone.

BufferMinimum

property

property BufferMinimum : Word

Default: 1000

- ↪ Defines the minimum number of bytes that must be in the output buffer before TApdFaxServer yields control.

Once started, a fax transmit session must have a constant supply of data to transmit. Lack of data to transmit is referred to as a data underflow condition and usually results in a failed fax session. Because Windows is a multi-tasking environment, the server tries to fill the output buffer as full as possible before yielding control. BufferMinimum is the minimum number of output bytes that must be in the output buffer before the server yields.

If your program is operating among programs that aren't functioning properly or other conditions that might result in long periods where it isn't given a chance to run, increasing `BufferMinimum` decreases the chance of a data underflow error.

An attempt to set `BufferMinimum` to more than the `OutSize` property of `TApdComPort` is ignored.

See also: `MaxSendCount`, `SafeMode`

BytesTransferred

read-only, run-time property

```
property BytesTransferred : Boolean
```

↳ The number of bytes received or transmitted so far for the current page.

`BytesTransferred` can be used by an `OnFaxStatus` event handler to get the number of bytes received or transferred so far. The appropriate time to check `BytesTransferred` is when `FaxProgress` equals `fpSendPage` or `fpGetPage`. At other times, it is either zero or a value associated with the previous page.

See also: `CurrentPage`, `PageLength`, `TotalPages`

CancelFax

method

```
procedure CancelFax;
```

↳ Cancels the current fax session.

`CancelFax` cancels the fax session, regardless of its current state. When appropriate, a cancel command is sent to the local modem or the remote fax device. The fax component generates an `OnFaxFinish` event with the error code `ecCancelRequested`, then cleans up and terminates. It also attempts to put the faxmodem back “onhook” (i.e., ready for the next call).

The following example shows how to cancel a fax from a fax status dialog:

```
procedure TStandardDisplay.CancelClick(Sender : TObject);
begin
    ApdSendFax1.CancelFax;
end;
```

See also: `OnFaxError`, `OnFaxFinish`

ComPort

property

```
property ComPort : TApdCustomComPort
```

- ↪ Determines the port to be used by the fax engine.

ComPort is the ApdComPort to be used by the fax engine. If the TapiMode property of the ApdComPort is tmOn, and a TApdTapiDevice is assigned to the TapiDevice property, TAPI will be used to configure and open the modem. If the TapiMode is tmOff, the port will be opened directly. See the section titled “Fax sessions and the TApdComPort” on page 702 for information regarding TApdComPort property settings required for faxing.

ConstantStatus

property

```
property ConstantStatus : Boolean
```

Default: False

- ↪ Determines whether status events are generated as soon as Monitoring is enabled.

When transmitting faxes, the time to display status events is clear. While there are faxes to transmit, status should be displayed; when there are no more faxes to transmit, the fax session is over. The issue is less clear when receiving faxes because TApdFaxServer is often waiting for faxes with no real status information to display.

If ConstantStatus is False (the default), the first status event is generated when an incoming ring is detected. The last status event is generated at the conclusion of the receipt of the fax. While TApdFaxServer is waiting for the next incoming call, no status events are generated. Status events are generated again when the next incoming ring is detected. This continues until CancelFax is called or a fatal error occurs.

If ConstantStatus is True, the first status event is generated as soon as StartReceive is called. The last status event is generated only after CancelFax is called or a fatal error occurs.

See also: Monitoring, OnFaxServerStatus

CurrentJobFileName

read-only, run-time property

```
property CurrentJobFileName : ShortString
```

- ↪ Contains the file name of the fax job file being sent.

CurrentJobFileName contains the file name of the fax job file currently being sent. You can use this value for status or logging purposes. The value of this property will be maintained until another fax is sent.

CurrentJobNumber

read-only, run-time property

property CurrentJobNumber : Integer

- ↪ Contains the index into the fax job file of the recipient being processed.

CurrentJobNumber contains the 0-based index into the fax job file (from the CurrentJobName property). For single-recipient fax jobs, this value will be 0. You can use this value for status or logging purposes. The value of this property will be maintained until another fax is sent.

CurrentPage

read-only, run-time property

property CurrentPage : Word

- ↪ The page number of the page currently being received or transmitted.

CurrentPage can be used by an OnFaxStatus event handler to get the number of the page currently being received or transmitted. The appropriate time to check CurrentPage is when FaxProgress equals fpSendPage or fpGetPage. At other times, it is either zero or a value associated with the previous page.

See also: BytesTransferred, PageLength, TotalPages

CurrentRecipient

read-only, run-time property

property CurrentRecipient : TFaxRecipientRec

```
TFaxRecipientRec = packed record
    Status : Byte;
    JobID : Byte;
    SchedDT : TDateTime;
    AttemptNum : Byte;
    LastResult : Word;
    PhoneNumber : String[50];
    HeaderLine : String[100];
    HeaderRecipient : String[30];
    HeaderTitle : String[30];
    Padding : Array[228..256] of Byte;
end;
```

- ↪ The TFaxRecipientRec containing recipient information about the current outbound fax.

CurrentRecipient contains information about the fax currently being sent. You can use the fields of this record for status or logging purposes. The TFaxRecipientRec of the last job will be maintained in this property until another fax is sent.

DelayBetweenSends

property

property DelayBetweenSends : Word

↳ Allows a minimum delay between fax transmissions.

DelayBetweenSends is the number of clock ticks to delay before beginning another fax transmission. If the SendQueryInterval property is non-zero, the ApdFaxServerManager will be queried for the next fax job at the end of each fax transmission. Some phone systems require several seconds between calls to reset the line. Set this property to the tick-count required by your phone system.

DesiredBPS

property

property DesiredBPS : Word

Default: 9600

↳ Determines the highest fax bps rate to negotiate for the next fax session.

DesiredBPS limits the fax bps rate for subsequent fax sessions. Although many faxmodems support higher bps rates (12000 and 14400), DesiredBPS defaults to 9600 for more reliable fax sessions and higher quality faxes because the slightly lower baud rate makes lines errors less likely.

Changing DesiredBPS during a fax session has no effect on the current session.

See also: ModemBPS, SessionBPS

DesiredECM

property

property DesiredECM : Boolean

Default: False

↳ Determines whether fax sessions attempt to use error control.

The fax protocol contains an optional error control facility that allows modems to detect and correct some transmission errors. Since very few faxmodems support fax error control, DesiredECM defaults to False, meaning the faxmodems do not attempt to negotiate error control.

See also: ModemECM, SessionECM

DestinationDir

property

```
property DestinationDir : string
```

Default: Empty string

↪ Determines the directory for incoming fax files.

There are two ways to indicate the directory for storing incoming fax files. If an `OnFaxServerName` event handler is used, it can return the fully qualified name (i.e., including drive and directory). If an `OnFaxServerName` event handler does not return the fully qualified name, or one of the automatic fax naming methods is used, `DestinationDir` can be set to the name of the desired directory.

If `DestinationDir` is empty and the `OnFaxServerName` event handler doesn't return a fully qualified file name, incoming files are stored in the current directory.

See also: `OnFaxServerName`

DialAttempt

read-only, run-time property

```
property DialAttempt : Word
```

↪ Indicates the number of times the current fax number has been dialed.

If the dialed fax number is busy, `TAPdSendFax` waits briefly and calls the number again. It tries up to `DialAttempts` times. The `DialAttempt` property returns the number of the current attempt. `DialAttempt` is incremented immediately upon encountering a busy line.

See also: `DialAttempts`, `DialRetryWait`

DialAttempts

property

```
property DialAttempts : Word
```

↪ Determines the number of times the fax number is dialed.

This is the number of times a fax call is attempted; it is not the number of retries. When `DialAttempts` dial operations are attempted without connection or successful fax transmission, the fax is considered a failure, and the fax job file is marked as such. The `OnFaxServerFinish` event will fire to provide notification of this, but the `OnFaxFatalError` event will not fire.

DialPrefix

property

```
property DialPrefix : TModemString  
TModemString = string[40];
```

↪ The optional dial prefix.

DialPrefix specifies an optional dial prefix that is inserted in the dial command between “ATDT” and the number to dial. If your telephone system requires special numbers or codes when dialing out, you can specify them once here rather than in every fax number.

Do not include “ATD” or a ‘T’ or ‘P’ tone/pulse modifier in the dial prefix. “ATD” is automatically prefixed by StartTransmit and the ‘T’ or ‘P’ is controlled by ToneDial.

See also: ToneDial

DialWait

property

```
property DialWait : Word  
Default: 60
```

↪ The number of seconds to wait for a connection after dialing the number.

This property determines how many seconds to wait after dialing the receiver's phone number. If the receiver does not answer within this time, the OnFaxServerFinish event will fire and return an error code of ecFaxNoAnswer.

See also: OnFaxServerFinish

EnhFont

property

```
property EnhFont : TFont
```

↪ Determines the font used to convert cover pages.

If EnhTextEnabled is True, the font specified by EnhFont is used by TApdFaxServer to convert the cover page. Any font available to Windows can be used (double click on the property to invoke the font dialog and see a list of the fonts). Only one font can be used for a document (i.e., font sizes and types cannot be mixed within a single cover page).

There is an upper limit on the size of the font, but this limit is not typically reached unless a very large font is used (e.g., greater than 72 pt). If the limit is exceeded, an ecEnhFontTooBig error occurs during the conversion process.

See also: EnhTextEnabled

EnhHeaderFont

property

```
property EnhHeaderFont : TFont
```

↳ Determines the font used to convert the fax header.

If EnhTextEnabled is True, the font specified by EnhHeaderFont is used by TApdFaxServer to convert the fax header. Any font available to Windows can be used (double click on the property to invoke the font dialog and see a list of the fonts).

There is an upper limit on the size of the font, but this limit is not typically reached unless a very large font is used (e.g., greater than 72 pt). If the limit is exceeded, an ecEnhFontTooBig error occurs during the conversion process.

See also: EnhTextEnabled

EnhTextEnabled

property

```
property EnhTextEnabled : Boolean
```

Default: False

↳ Determines whether TApdFaxServer uses the default font.

If EnhTextEnabled is True, the enhanced text-to-fax converter is used by ApdFaxSever when converting fax headers and text cover pages. This means that the font specified by EnhFont is used to convert the cover page and the font specified by EnhHeaderFont is used to convert the fax header.

The converter makes no attempt to keep all text on the page when the size of the font is changed. Ensure that the cover page line length and document length fit on the page in the desired font.

See also: EnhFont, EnhHeaderFont

ExitOnError

property

```
property ExitOnError : Boolean
```

↳ Determines what happens when an error occurs during a fax transmit or receive.

If ExitOnError is True, all fax operations are stopped if any error occurs while sending or receiving. Monitoring for incoming faxes and automatic querying for faxes to send is halted until explicitly re-started. If ExitOnError is False, faxing operations will continue if non-fatal errors occur. Fatal errors, such as ecBadResponse, will still stop all fax operations and cause the OnFaxServerFatalError event to fire.

FaxClass

property

```
property FaxClass : TFaxClass
```

```
TFaxClass = (  
    fcUnknown, fcDetect, fcClass1, fcClass1_0, fcClass2, fcClass2_0);
```

Default: fcDetect

↳ Indicates whether the faxmodem is used as Class 1, Class 1.0, Class 2, or Class 2.0.

If FaxClass is fcDetect (the default), TApdFaxServer determines what classes the modem supports and enables the highest class. If you set FaxClass to a specific class, no attempt is made to determine if the class you request is supported by the faxmodem.

See also: SupportedFaxClasses

FaxFile

read-only, run-time property

```
property FaxFile : string
```

↳ The name of the fax file currently being transmitted or received.

If you are sending a single fax, set FaxFile to the name of the file. If you are sending multiple fax files, you must implement an OnFaxNext event handler. FaxFile is automatically set to the fax file returned by your event handler.

FaxFile can be used with status and logging routines to return the name of the fax file currently being transmitted or received.

See also: TApdSendFax.CoverFile, TApdSendFax.OnFaxNext,
TApdSendFax.PhoneNumber

FaxFileExt

property

```
property FaxFileExt : string
```

Default: "APF"

↳ The default extension assigned to incoming fax files.

By default, all incoming fax files created by the two built-in methods of naming faxes use a file extension of APF. You can change the extension assigned to incoming files by setting FaxFileExt to the new desired extension.

See "Naming incoming fax files" on page 754 for more information.

See also: FaxNameMode

FaxLog

property

```
property FaxLog : TApdFaxLog
```

↳ An instance of a fax logging component.

If FaxLog is nil (the default), the fax component does not perform automatic logging. You can install an OnFaxServerLog event handler to perform logging in this case.

If you create an instance of (or a descendant of) a TApdFaxLog class (see page 828), and assign it to FaxLog, the fax component will call the log component's UpdateLog method automatically.

FaxNameMode

property

```
property FaxNameMode : TFaxNameMode
```

```
TFaxNameMode = (fnNone, fnCount, fnMonthDay);
```

Default: fnCount

↳ Determines how an incoming fax is named.

TApdFaxServer must assign a file name to incoming fax files. This property differs from the TApdReceiveFax component's property somewhat. If FaxNameMode is fnNone, you must supply an OnFaxServerName event to provide a file name for the incoming file; otherwise, the incoming fax file will not be saved. If FaxNameMode is fnCount or fnMonthDay, the fax name will follow the naming conventions of the TApdReceiveFax. OnFaxServerName event will be generated only if FaxNameMode is fnNone.

See also: TApdReceiveFax.FaxNameMode

FaxPrinter

property

```
property FaxPrinter : TApdCustomFaxPrinter
```

↳ The TApdCustomFaxPrinter to use for automatic fax printing.

If PrintOnReceive is True, this instance of a TApdCustomFaxPrinter will be used to automatically print received faxes.

See also: PrintOnReceive

FaxProgress

read-only, run-time property

```
property FaxProgress : Word
```

↳ Returns a code that indicates the current state of the fax session.

This property is most useful within an OnFaxServerStatus event handler. See “Fax status” on page 707 for more information.

See also: OnFaxServerStatus

ForceSendQuery

method

```
procedure ForceSendQuery;
```

↳ Provides a mechanism for manually querying for scheduled fax jobs.

The ForceSendQuery method allows you to manually request fax jobs from the TApdFaxServerManager component. The TApdFaxServerManager provides outbound faxes automatically every SendQueryInterval seconds, or on demand with the ForceSendQuery method. If a fax job is scheduled to be sent when this method is called, it will be processed and sent.

See also: SendQueryInterval

HangupCode

read-only, run-time property

```
property HangupCode : Word
```

↳ The hangup code for a Class 2 or 2.0 fax transfer.

When a Class 2 or 2.0 faxmodem session terminates abnormally, it returns a hangup code to help explain what went wrong. Although these codes refer to low-level portions of the faxmodem link over which you have no control, sometimes they can point out a programming error that you can correct.

The following table shows the codes that can be returned (in hexadecimal), with a brief description of each one. The codes are grouped according to the transfer phase in which they can occur. Some of the terms in the table are defined only in the Class 2 and 2.0 specification. Refer to that specification for more information.

Hangup Code	Description
Call placement and termination	
00	Normal end of connection
01	Ring detect without successful handshake
02	Call aborted from +FKS or <Can>
03	No loop current

Hangup Code	Description
04	Ringback detected, no answer timeout
05	Ringback detected, answer without CED
Transmit phase A	
10	Unspecified phase A error
11	No answer
Transmit phase B	
20	Unspecified phase B error
21	Remote cannot receive or send
22	COMREC error in transmit phase B
23	COMREC invalid command received
24	RSPREC error
25	DCS sent three times without response
26	DIS/DTC received three times; DCS not recognized
27	Failure to train at 2400 bps or +FMS error
28	RSPREC invalid response received
Transmit phase C	
40	Unspecified transmit phase C error
41	Unspecified image format error
42	Image conversion error
43	DTE to DCE data underflow
44	Unrecognized transparent data command
45	Image error, line length wrong
46	Image error, page length wrong
47	Image error, wrong compression code
Transmit phase D	
50	Unspecified transmit phase D error
51	RSPREC error
52	MPS sent three times without response
53	Invalid response to MPS
54	EOP sent three times without response
55	Invalid response to EOP
56	EOM sent three times without response
57	Invalid response to EOM

Hangup Code	Description
58	Unable to continue after PIN or PIP
Receive phase B	
70	Unspecified receive phase B error
71	RSPREC error
72	COMREC error
73	T.30 T2 timeout, expected page not received
74	T.30 T1 timeout after EOM received
Receive phase C	
90	Unspecified receive phase C error
91	Missing EOL after 5 seconds
92	Bad CRC or frame (ECM mode)
93	DCE to DTE buffer overflow
Receive phase D	
100	Unspecified receive phase D error
101	RSPREC invalid response received
102	COMREC invalid response received
103	Unable to continue after PIN or PIP

InitBaud

property

property InitBaud : Integer

Default: 0

↪ Determines the initialization baud rate for modems that require different baud rates for initialization and fax operations.

Some older 24/96 faxmodems (2400 data, 9600 fax) require that the initialization commands be sent at 2400 baud, but that all fax commands and fax data be sent and received at 19200. The fax software must constantly adjust the current baud rate depending on the operation it is performing.

Since most faxmodems do not require a special initialization baud rate, `InitBaud` defaults to zero, which means that no baud rate switches are performed. If you encounter an older modem that requires this behavior, set `InitBaud` to 2400.

`NormalBaud` is a companion property to `InitBaud`. When `InitBaud` is non-zero, the fax components switch to the specified baud rate when sending initialization commands and switch back to the normal baud rate, 19200, when sending fax commands or fax data. If you encounter a case where the normal baud rate should be something other than 19200, you must change `NormalBaud`.

See also: `NormalBaud`

MaxSendCount

property

property `MaxSendCount` : Word

Default: 50

- ↳ Determines the maximum number of raster lines `TApdFaxServer` sends before yielding control.

`MaxSendCount` prevents `TApdFaxServer` from completely taking over the CPU. It provides a balance to `BufferMinimum` back towards sharing the CPU among all tasks.

`MaxSendCount` overrides `BufferMinimum` and forces `TApdFaxServer` to yield after sending `MaxSendCount` raster lines, even if the output buffer contains less than `BufferMinimum` bytes. The risk, of course, is that yielding too soon may result in a data underflow error.

The default values for `BufferMinimum` and `MaxSendCount` provide the best combination of cooperative multi-tasking and avoidance of data underflow. You should not alter these values unless fax sessions are failing due to data underflow errors.

See also: `BufferMinimum`, `SafeMode`

```
property ModemBPS : LongInt
```

↳ Returns the highest bps rate supported by the faxmodem.

When you reference ModemBPS, commands are sent to the modem to determine its highest bps rate. This works only for Class 2 and 2.0 modems; a Class 1 and Class 1.0 modems cannot report this information until a fax connection has been established.

ModemBPS works by attempting to enable the most capable modem features and stepping down if the modem returns “ERROR.” It starts at a 14400 bps transfer rate, then tries 12000, 9600, 7200, 4800, and 2400.

The technique used by ModemBPS works on most Class 2 and 2.0 faxmodems. One low-cost, no-name-clone faxmodem we tested wouldn’t return “ERROR” no matter what was asked of it, even though it supported only 9600 bps with no error correction.

See also: ModemECM

```
property ModemChip : string
```

↳ Returns the type of chip for a Class 2 or 2.0 faxmodem.

When you reference ModemChip, commands are sent to the modem to determine the type of chip. This works only for Class 2 and 2.0 modems. ModemChip is an empty string for a Class 1 and Class 1.0 modems.

See also: ModemModel, ModemRevision

```
property ModemECM : Boolean
```

↳ Indicates whether the faxmodem supports error correction.

When you reference ModemECM, commands are sent to the modem to determine whether it supports error correction. This works only for Class 2 and 2.0 modems; a Class 1 and Class 1.0 modem cannot report this information until a fax connection has been established.

The technique used by ModemECM works on most Class 2 and 2.0 faxmodems. But be advised, one low-cost, no-name-clone faxmodem tested wouldn’t return “ERROR” no matter what was asked of it, even though it supported only 9600 bps with no error correction.

See also: ModemBPS

ModemInit

property

```
property ModemInit : TModemString
```

```
TModemString = string[40];
```

↳ A custom modem initialization string.

If you assign a custom modem initialization string to `ModemInit`, Async Professional always sends this string to the modem just before it sends its own `DefInit` string (“ATE0Q0V1X4S0=0”). This occurs whenever you call `StartTransmit`, `StartReceive`, or `InitModemForFaxReceive`.

Note that the `DefInit` string may override certain actions of the `ModemInit` string. This is necessary for proper operation of the Async Professional fax routines.

ModemModel

read-only, run-time property

```
property ModemModel : string
```

↳ Returns the model for a Class 2 or 2.0 faxmodem.

When you reference `ModemModel`, commands are sent to the modem to determine the model. This works only for Class 2 and 2.0 modems. `ModemModel` is an empty string for a Class 1 and Class 1.0 modems.

See also: `ModemChip`, `ModemRevision`

ModemRevision

read-only, run-time property

```
property ModemRevision : string
```

↳ Returns the revision for a Class 2 or 2.0 faxmodem.

When you reference `ModemRevision`, commands are sent to the modem to determine the revision. This works only for Class 2 and 2.0 modems. `ModemRevision` is an empty string for a Class 1 and Class 1.0 modems.

See also: `ModemChip`, `ModemModel`

Monitoring

property

property Monitoring : Boolean

Default: False

- ↳ Determines whether incoming calls are answered.

Set Monitoring to True to begin listening for incoming fax calls. Set Monitoring to False to stop listening for incoming calls. Monitoring will cause the TApdFaxServer to wait for faxes in the background, occasionally generating OnFaxServerStatus events. Monitoring will remain in effect until Monitoring is set to False, or a fatal error occurs and the OnFaxServerFatalError event fires.

NormalBaud

property

property NormalBaud : Integer

Default: 0

- ↳ Determines the normal baud to use for modems that require different baud rates for initialization and fax operations.

NormalBaud isn't needed unless the faxmodem requires separate baud rates for initialization commands and the baud rate required for normal fax operations is not 19200. See the InitBaud property on page 724 for a complete description of the operation of InitBaud and NormalBaud.

See also: InitBaud

OnFaxServerAccept

event

property OnFaxServerAccept : TFaxAcceptEvent

```
TFaxAcceptEvent = procedure(  
    CP : TObject; var Accept : Boolean) of object;
```

- ↳ Defines an event handler that is called at the beginning of the receive fax session after the station ID of the caller is received.

This event provides an opportunity to accept or reject an incoming fax. If an `OnFaxServerAccept` event handler is not provided, all faxes are accepted. CP is the `TApdFaxServer` that is answering the call. Set `Accept` to `True` to accept the fax, `False` to reject it. When this event is generated, the only information known about the incoming fax is the station ID. So, the typical application for this event is to accept or reject faxes based on the station ID. For example, you can filter junk faxes by accepting only faxes coming from known station IDs.

See “Accepting fax files” on page 754 for more information.

OnFaxServerFatalError

event

```
property OnFaxServerFatalError : TFaxServerFatalErrorEvent

TFaxServerFatalErrorEvent = procedure(
    CP : TObject; FaxMode : TFaxServerMode; ErrorCode,
    HangupCode : Integer) of object;

TFaxServerMode = (fsmIdle, fsmSend, fsmReceive);
```

↳ Defines an event handler that is called when a fatal error occurs during fax transmission or reception.

This event is generated only for unrecoverable errors. Most fax errors caused by line noise are handled automatically by the fax devices and are not reported through this event handler.

CP is the `TApdFaxServer` that generated the event. `FaxMode` is the direction of the fax. `ErrorCode` is a number indicating the type of error reported by the component. `HangupCode` is the number indicating the reason for hang-up as reported by a Class2 or Class 2.0 faxmodem. Class 1 and Class 1.0 modems will always return 0 for `HangupCode`.

Non-fatal errors, such as `ecOK`, `ecFaxBusy`, or `ecCancelRequested` will not cause this event to be generated.

If `ExitOnError` is `True`, `Monitoring` is set to `False` suspending further reception, and `SendQueryInterval` is set to 0 suspending further querying for new fax jobs. If this event fires, some form of intervention is required, such as re-initializing the modem.

```
property OnFaxServerFinish : TFaxServerFinishEvent

TFaxServerFinishEvent = procedure(
    CP : TObject; FaxMode : TFaxServerMode;
    ErrorCode : Integer) of object;

TFaxServerMode = (fsmIdle, fsmSend, fsmReceive);
```

↳ Defines an event handler that is called when a fax call ends.

This event is generated at the end of each successful fax transmission or reception. CP is the TApdFaxServer that generated the event. FaxMode is the direction of the fax. ErrorCode is a number indicating the result of the fax, which could be ecOK, ecCancelRequested, or ecFaxBusy. Further fax send and receive operations will continue based on the values of Monitoring and SendQueryInterval.

OnFaxServerLog**event**

```
property OnFaxServerLog : TFaxLogEvent

TFaxLogEvent = procedure(
    CP : TObject; LogCode : TFaxLogCode) of object;

TFaxLogCode = (lfaxNone, lfaxTransmitStart, lfaxTransmitOK,
    lfaxTransmitFail, lfaxReceiveStart, lfaxReceiveOK,
    lfaxReceiveSkip, lfaxReceiveFail);
```

↳ Defines an event handler that is called at designated points during a fax transfer.

The primary purpose of this event is to allow you to log statistical information about fax transfers. For example, you could log the transfer time and whether the transfer succeeded or failed. You can also use this event for start-up and cleanup activities.

CP is the fax component to be logged. LogCode is a code that indicates the state of the fax transfer. The possible states are listed in “Fax logging” on page 712. No other information is passed with this event, but you can use the fax status properties such as CurrentRecipient to get additional information about the fax session.

OnFaxServerName

event

```
property OnFaxServerName : TFaxNameEvent

TFaxNameEvent = procedure(
    CP : TObject; var Name : TpassString) of object;
```

↳ Defines an event handler that is called to return a file name for an incoming fax.

The fax receive protocol does not include file name information, so the receiving software must generate file names (see FaxNameMode) or you can generate the file name in an OnFaxServerName event handler. If the FaxNameMode property is fnCount or fnMonthDay, this event will not be generated. If FaxNameMode is fnNone this event will be generated. If this event is not defined and FaxNameMode is fnNone, the fax will not be saved.

OnFaxServerPortOpenClose

event

```
property OnFaxServerPortOpenClose : TFaxServerPortOpenCloseEvent

TFaxServerPortOpenCloseEvent = procedure(
    CP : Tobject; Opening : Boolean) of object;
```

↳ Defines an event handler that is called when the physical communications port is opened and closed.

This event is generated when the physical communications port is opened or closed. The primary purpose of this event is notification, but it can also be used to access the port prior to the fax being generated. The fax will not continue until this event exits. If a TapiDevice is defined, this event will fire when TAPI hands an open communications port to the application.

OnFaxServerStatus

event

```
property OnFaxServerStatus : TFaxServerStatusEvent

TFaxServerStatusEvent = procedure(CP : TObject;
    FaxMode : TFaxServerMode; First, Last : Boolean;
    Status : Word) of object;
```

↳ Defines an event handler that is called regularly during a file transfer.

This event is generated at StatusInterval intervals during the entire fax session and after the completion of each major operation (e.g. incoming ring detected, remote station ID received).

You can use this event to update a status display that informs the user about the fax progress.

CP is the TApdFaxServer component that generated the event. FaxMode is the direction of the fax. First is True on the first call to the handler for the given FaxMode, False otherwise. Last is True on the last call to the handler for the given FaxMode, False otherwise. Status is the value of the FaxProgress property at the time the event was generated.

A predefined status component called TApdFaxStatus is supplied with Async Professional. If you don't want to write an OnFaxServerStatus event handler, you can use this standard fax status window. Just create an instance of TApdFaxStatus and assign it to the StatusDisplay property of the TApdFaxServer.

See also: StatusDisplay, StatusInterval

PageLength

read-only, run-time property

property PageLength : LongInt

↳ The total number of bytes in the current page.

PageLength is valid only when you are sending a fax. When receiving a fax, the total size of the page is not known in advance, so PageLength is zero.

PageLength can be used by an OnFaxStatus event handler to get the total number of bytes in the current page. The appropriate time to check PageLength is when FaxProgress equals fpSendPage or fpGetPage. At other times, it is either zero or a value associated with the previous page.

See also: BytesTransferred, CurrentPage, TotalPages

PrintOnReceive

property

property PrintOnReceive : Boolean

Default: False

↳ Determines whether incoming faxes are automatically printed upon completion.

If PrintOnReceive is True, received faxes are automatically printed. This property can only be True if a TApdCustomFaxPrinter is assigned to the FaxPrinter property.

RemoteID

read-only, run-time property

```
property RemoteID : TStationID
```

```
TStationID = string[20];
```

↳ The station ID of the remote fax machine.

RemoteID can be used by an OnFaxServerStatus event handler to get the station ID of the remote fax machine. The appropriate time to check RemoteID is when FaxProgress equals fpGotRemoteID. Before that, it returns an empty string.

See also: StationID

SafeMode

property

```
property SafeMode : Boolean
```

Default: True

↳ Determines whether TApdFaxServer should yield during time-critical handshaking periods.

At the beginning and end of every fax page, TApdFaxServer performs time-critical handshaking with the receiving fax device, where tolerance for delays is very low. Delays as small as a few hundred milliseconds can cause the receiver to believe the page transfer failed.

When SafeMode is True (the default), TApdFaxServer does not yield during these periods. While this is the safest possible mode of operation, it will result in apparent brief periods of unresponsiveness to the user. That is, they won't be able to switch to another application or cancel the fax transfer until the critical handshaking is over. Fortunately, the time-critical period lasts only 1-3 seconds, so this should rarely be a problem.

SendQueryInterval

property

```
property SendQueryInterval : Word
```

Default: 30

↳ Determines the number of seconds between querying the TApdFaxServerManager for new fax jobs.

When a fax is not being received or transmitted, the TApdFaxServerManager component will be queried for fax transmission jobs that are scheduled, or in the queue to be scheduled. Set this property to 0 to disable automatic querying. When a send or receive operation is complete, the TApdFaxServerManager will be queried, and the interval will be reset.

```
property ServerManager : TApdFaxServerManager
```

↳ The TApdFaxServerManager that supplies fax jobs.

Assign a TApdFaxServerManager component to this property to allow the TApdFaxServer to automatically query for scheduled fax jobs.

SessionBPS**read-only, run-time property**

```
property SessionBPS : Word
```

↳ The negotiated transfer rate in bits per second.

SessionBPS can take on the values 14400, 12000, 9600, 7200, 4800, and 2400. Most faxmodems now support 9600 or higher. The fax connection process attempts to negotiate the highest possible rate unless you have set DesiredBPS to limit the highest rate.

SessionBPS can be used by an OnFaxServerStatus event handler to get the negotiated transfer rate. The appropriate time to check SessionBPS is when FaxProgress equals fpSessionParams. Before that, it is undefined.

Session parameters can change more than once during a single session. Be sure that your OnFaxServerStatus event handler updates its parameter display each time FaxProgress returns the value fpSessionParams.

See also: DesiredBPS, SessionECM, SessionResolution, SessionWidth

SessionECM**read-only, run-time property**

```
property SessionECM : Boolean
```

↳ Indicates whether automatic error correction is enabled.

SessionECM is True if automatic error correction is enabled for this transfer, or False if it isn't. Error correction is enabled if both modems support it and DesiredECM is True.

SessionECM can be used by an OnFaxServerStatus event handler to check for automatic error correction. The appropriate time to check SessionECM is when FaxProgress equals fpSessionParams. Before that, it is undefined.

Session parameters can change more than once during a single session. Be sure that your OnFaxServerStatus event handler updates its parameter display each time FaxProgress returns the value fpSessionParams.

See also: DesiredECM, SessionBPS, SessionResolution, SessionWidth

SessionResolution

read-only, run-time property

```
property SessionResolution : Boolean
```

↳ Indicates whether the fax is high resolution or standard resolution.

SessionResolution is True for a high resolution fax transfer, or False for a standard resolution transfer. Async Professional automatically enables high resolution if it is sending an APF file that contains high resolution data, or if it is receiving a high resolution fax from a remote partner.

SessionResolution can be used by an OnFaxStatus event handler to check for the fax resolution. The appropriate time to check SessionResolution is when FaxProgress equals fpSessionParams. Before that, it is undefined.

Session parameters can change more than once during a single session. Be sure that your OnFaxServerStatus event handler updates its parameter display each time FaxProgress returns the value fpSessionParams.

See also: SessionBPS, SessionECM, SessionWidth

SessionWidth

read-only, run-time property

```
property SessionWidth : Boolean
```

↳ Indicates whether the fax is normal or wide width.

If SessionWidth is True (the default), the fax is a standard width of 1728 pixels (about 8.5 inches). If SessionWidth is False, the fax width is 2048 pixels (about 10 inches).

SessionWidth can be used by an OnFaxServerStatus event handler to check the fax width. The appropriate time to check SessionWidth is when FaxProgress equals fpSessionParams. Before that, it is undefined.

Session parameters can change more than once during a single session. Be sure that your OnFaxServerStatus event handler updates its parameter display each time FaxProgress returns the value fpSessionParams.

See also: SessionBPS, SessionECM, SessionResolution

```
property SoftwareFlow : Boolean
```

Default: False

- ↳ Determines whether the fax components enable/disable software flow control during the fax session.

When using software flow control during a fax session, the flow control must be enabled and disabled at various points in the session. Because hardware flow control is more reliable, it is used by default and the fax components do not enable/disable software flow control. If you need to use software flow control, SoftwareFlow must be set to True.

For more information regarding flow control see “Fax sessions and the TApdComPort” on page 702.

StationID**property**

```
property StationID : TStationID
```

```
TStationID = string[20];
```

- ↳ The station ID of the faxmodem.

A fax device can identify itself to another fax device with a 20 character name, called the station ID. The Class 1, Class 1.0, Class 2, and Class 2.0 specifications indicate that the station ID should contain just a phone number; therefore they limit it to just the digits 0 through 9 and space. However, the station ID is frequently used to store an alphabetic name. Most faxmodems support this convention by allowing upper and lower case letters, as well as other special characters in the station ID. This can cause problems for some fax machines, though, since they cannot print these characters.

Async Professional does not filter the characters stored in the station ID. If your software must be compatible with the broadest possible range of fax hardware, you may want to limit the characters stored in StationID.

This station ID is used on both incoming and outgoing calls.

A fax file stored in APF format also contains a station ID in the file header. This station ID is stored when a document is converted to APF format. For more information, see “TApdFaxConverter Component” on page 594.

See also: TApdFaxConverter.StationID

StatusDisplay

property

```
property StatusDisplay : TApdAbstractFaxStatus
```

↳ An instance of a fax status window.

If `StatusDisplay` is nil (the default), the fax does not provide an automatic status window. You can install an `OnFaxServerStatus` event handler to display status in this case.

If you create an instance of a class derived from `TApdAbstractFaxStatus` or use the supplied `TApdFaxStatus` component (see page 826) and assign it to `StatusDisplay`, the status window is displayed and updated automatically.

See also: `OnFaxServerStatus`

StatusInterval

property

```
property StatusInterval : Word
```

Default: 1

↳ The maximum number of seconds between `OnFaxServerStatus` events.

The `OnFaxServerStatus` event is generated for each major fax session event (connected, got station ID, and so on) and at intervals of `StatusInterval` seconds.

This property also determines how frequently the `StatusDisplay` window is updated.

See also: `OnFaxServerStatus`, `StatusDisplay`

StatusMsg

method

```
function StatusMsg(const Status : Word) : string;
```

↳ Returns an English string for a fax status code.

This routine is intended primarily for use in fax status routines. It returns a status string from the string table resource linked into your EXE. The string ID numbers correspond to the values of the `fpXxx` constants (see “Fax status” on page 707). If the string table doesn’t contain a string resource with the requested ID, an empty string is returned.

The returned string is never longer than `MaxMessageLen` (80) characters.

SupportedFaxClasses

read-only, run-time property

```
property SupportedFaxClasses : TFaxClassSet  
TFaxClassSet = set of TFaxClass;  
TFaxClass = (  
    fcUnknown, fcDetect, fcClass1, fcClass1_0, fcClass2, fcClass2_0);
```

↪ The set of fax classes supported by the faxmodem.

SupportedFaxClasses is available only at run time because it sends commands to the faxmodem to determine what baud rates are supported (when it equals fcDetect).

Initially FaxClass is fcDetect, so that the first reference to it causes the faxmodem interrogation. Thereafter, references to SupportedFaxClasses return the known set of supported fax classes. You can force the re-interrogation of the faxmodem by setting FaxClass to fcDetect.

Generally, applications should use the highest supported class: fcClass2_0, then fcClass2, then fcClass1_0, and finally fcClass1.

See also: FaxClass

TapiDevice

property

```
property TapiDevice: TApdCustomTapiDevice
```

↪ Determines the TAPI device used by the TApdFaxServer.

If TapiDevice is nil (the default), TAPI will not be used to select the device or open the physical port.

If the TApdComPort specified in the ComPort property has its TapiMode set to tmAuto or tmOn, and a TApdTapiDevice is assigned to the TapiDevice property, TAPI will be used to select the device and open the physical port.

ToneDial

property

```
property ToneDial : Boolean
```

Default: True

↪ Determines whether tone or pulse dialing is used for fax transmissions.

If ToneDial is True (the default), tone dialing is used. Otherwise, pulse dialing is used. Setting ToneDial does not immediately issue a modem command, but determines whether 'T' or 'P' is added to the dial command later.

property TotalPages : Word

↳ The total number of pages in the current fax.

TotalPages is valid only when you are sending a fax. When you are receiving a fax, the total number of pages is not known in advance, so TotalPages is zero.

TotalPages can be used by an OnFaxStatus event handler to get the total number of bytes in the current page. The appropriate time to check TotalPages is when FaxProgress equals fpSendPage or fpGetPage. Before that, it is either zero or a value associated with the previous page.

See also: BytesTransferred, CurrentPage, PageLength

TApdFaxServerManager Component

The TApdFaxServerManager component provides fax scheduling and queuing capability. The ApdFaxServer component requests new fax jobs periodically (every SendQueryInterval seconds) or manually (by calling the ForceSendQuery method). The ApdFaxServerManager component checks the MonitorDir folder when queried for fax jobs. If the fax job is ready to be sent, the ApdFaxServerManager extracts the fax recipient information, cover page, and APF data from the APJ file, and the ApdFaxServer will then fax the document to the recipient.

Hierarchy

TComponent (VCL)

❶ TApdBaseComponent (OOMisc)	8
❷ TApdFaxJobHandler (AdFxSrv)	769
TApdFaxServerManager (AdFxSrv)	

Properties

DeleteOnComplete	MonitorDir	❶ Version
JobFileExt	Paused	

Methods

❷ AddRecipient	❷ GetJobHeader	Reset
❷ ConcatFaxes	GetNextFax	❷ ShowFaxJobInfoDialog
❷ ExtractAPF	❷ GetRecipient	UpdateStatus
❷ ExtractCoverFile	GetSchedTime	
GetJob	❷ MakeJob	

Events

OnCustomGetJob	OnQueried	OnRecipientComplete
----------------	-----------	---------------------

Reference Section

GetJob

method

```
function GetJob(var Recipient : TFaxRecipientRec;  
    QueryFrom : TApdCustomFaxServer; var JobFileName, FaxFile,  
    CoverFile : ShortString) : Boolean;
```

↳ Returns the next scheduled fax job.

The GetJob method is the workhorse of the TApdFaxServerManager component. This method is used internally by the TApdFaxServer component to query the TApdFaxServerManager for the next fax to send.

Call GetJob to get the next scheduled fax job in the MonitorDir. Recipient will contain the TFaxRecipientRec record consisting of the receiver's data. JobFileName will contain the file name of the fax job. FaxFile will contain the file name of the extracted APF that will be faxed. CoverFile will contain the file name of the extracted cover file text. The return value will be True if a fax job is scheduled to be sent at the time the method is called; False if a fax job is not scheduled. The TApdFaxServer component will name the FaxFile and CoverFile using the JobName or Sender fields of the TFaxJobHeaderRec. Pass an explicit file name for the FaxFile and CoverFile parameters to override the default naming scheme.

GetNextFax

method

```
function GetNextFax : ShortString;
```

↳ Returns the file name of the next fax job to send.

The GetNextFax method returns the file name of the next scheduled fax job in the directory being monitored. If a fax job is not scheduled this method will return an empty string.

This method scans the directory specified by MonitorDir and consolidates scheduling information for each fax file. Then, the fax job with the earliest scheduled time is located. If that job's scheduled time is less than or equal to the current time, the name of that job file is returned. If the scheduled time has not passed, an empty string is returned.

See also: MonitorDir

GetSchedTime

method

```
function GetSchedTime(JobName : ShortString) : TDateTime;
```

↳ Returns the TDateTime for which the first fax in the job file is scheduled.

The GetSchedTime method returns the TDateTime for which the earliest fax contained in the APJ file is scheduled. JobName is the file name of the APJ file.

This method opens the job file, parses the embedded TFaxRecipientRec structures and returns the scheduled date/time of the earliest scheduled fax in the file. The TFaxJobHeaderRec.SchedDT field will be updated according to this value.

JobFileExt

property

```
property JobFileExt : ShortString
```

Default: "APJ"

↳ The extension of Async Professional fax Job files.

The TApdFaxServerManager uses this property to filter out files that are not APJ files when looking for fax jobs. Files in the MonitorDir directory without this extension are not included when the directory is scanned.

See also: MonitorDir

MonitorDir

property

```
property MonitorDir : ShortString
```

Default: Empty string

↳ Determines the directory to scan for fax job files.

When the TApdFaxServerManager is queried for fax jobs, it scans the files in this directory. Only one TApdFaxServerManager component can look at any given directory at a time. To prevent multiple TApdFaxServerManagers from monitoring the same directory, a lock file is used. This lock file is created with exclusive access when MonitorDir is set to a valid directory and deleted when the TApdServerManager is destroyed or MonitorDir is changed. An ecAlreadyMonitored exception is raised if the directory is already being monitored by another TApdFaxServerManager.

```
property OnCustomGetJob : TManagerUserGetJobEvent;

TManagerUserGetJobEvent = procedure(Mgr : TApdFaxServerManager;
  QueryFrom : TApdCustomFaxServer; var JobFile, FaxFile,
  CoverFile : string; var RecipientNum : Integer) of object;
```

✚ Event generated to override the default fax file scheduling.

This event is generated to allow custom scheduling of fax job files. If this event handler is installed the default scheduling built into the TApdFaxServerManager (through the GetJob method) is bypassed completely; the parameters of this event will be used instead.

This event is generated when a TApdCustomFaxServer requests a fax to send, either due to the SendQueryInterval timer, the end of a send or receive operation, or a call to ForceSendQuery. The application must extract the fax file (APF) and optional cover page file from the fax job file before this event exits.

Mgr is the TApdFaxServerManager that generated the event. QueryFrom is the TApdCustomFaxServer that is requesting a fax to send. JobFile is the name of the fax job file (APJ) that contains the next job to send. FaxFile is the name of the fax file (APF) that will be sent. CoverFile is the name of the cover page file (in either ASCII text or APF format) that will be used. RecipientNum is the index of this recipient in the fax job file (APJ).

This event can be used to provide an alternate scheduling mechanism, or fax job storage mechanism. For example, fax recipients can be filtered to specific modems or the fax jobs can be stored in a database. When using this event to access a different storage mechanism, a temporary fax job file must be created (see the TApdFaxJobHandler for the applicable methods). The TApdFaxServerManager will update this file with the appropriate status flags upon completion of the fax.

See also: GetJob, OnQueried

OnQueried

event

```
property OnQueried : TFaxServerManagerQueriedEvent;

TFaxServerManagerQueriedEvent = procedure(
  Mgr : TApdFaxServerManager; QueryFrom : TApdCustomFaxServer;
  const JobToSend : string) of object;
```

✚ Event generated when a TApdFaxServer requests a fax.

This event is generated when a TApdFaxServer component requests a fax to send, either through the SendQueryInterval timer, at the end of a fax send or receive operation, or through ForceSendQuery. Use this event to determine which fax will be sent on the TApdFaxserverManager level.

Mgr is the TApdFaxServerManager that generated the event. QueryFrom is the TApdCustomFaxServer that is requesting a fax. JobToSend is the name of the fax job file (APJ) that will be sent.

Note that this event is for notification purposes only, the JobToSend is not editable. Use the OnCustomGetJob event handler to change the behavior of the next job scheduling mechanism.

See also: OnCustomGetJob

OnRecipientComplete

event

```
property OnRecipientComplete : TManagerUpdateRecipientEvent;  
  
TManagerUpdateRecipientEvent = procedure (  
    Mgr : TApdFaxServerManager; JobFile : string;  
    JobHeader : TFaxJobHeaderRec;  
    var RecipHeader : TFaxRecipientRec) of object;
```

↳ This event is generated when a recipient in a fax job is complete.

The TApdFaxServerManager component will generate this event when a TApdFaxServer notifies it of a completed fax. This event is generated shortly after the TApdFaxServer component generates the OnFaxServerFinish or OnFaxServerFatalError event.

Mgr is the TApdFaxServerManger that generated the event. JobFile is the name of the fax job file (APJ) that contains the completed fax. JobHeader is the TFaxJobHeaderRec that describes the fax jobs contained in JobFile. RecipHeader is the TFaxRecipientRec that describes the fax that just completed.

This event is generated immediately before the JobFile is updated. The RecipHeader parameter of this event can be modified in this event for rescheduling if needed. Once this event exits, the JobFile's JobHeader, and the recipient's RecipHeader are updated.

Paused

property

```
property Paused : Boolean
```

Default: False

↳ Temporarily pauses the TApdFaxServerManager's queueing functionality.

If Paused is False (the default) the GetJob method scans the directory specified by MonitorDir for fax jobs. If Paused is True, GetJob returns False and exits. Use this property to temporarily pause processing of fax jobs in the TApdServerManager's monitored directory.

See also: GetJob, MonitorDir

```
procedure Reset;
```

↪ Reset resets the internal fax list.

Use Reset to reset the TApdFaxServerManger's internal fax file list. This list is maintained to increase the speed of multiple queries by TApdFaxServer components. Use the Reset method to clear the list, which will force a re-scan of the MonitorDir the next time the TApdFaxServerManager is queried.

UpdateStatus**method**

```
procedure UpdateStatus(JobFileName : ShortString;  
    JobNumber, Result : Word; Failed : Boolean);
```

↪ Updates the status flag of the specified job.

UpdateStatus updates the TFaxJobHeaderRec.Status and TFaxRecipientRec.Status fields of the specified fax job file. JobFileName is the file name of the fax job file to update. JobNumber is the index of the job to update. Result is the ErrorCode of the fax operation that just completed. Failed is False if the job can be retried, or True if all retries have been attempted.

TApdFaxClient Component

The TApdFaxClient component provides the ability to create APJ fax job files with a design-time interface. The MakeFaxJob method creates a single-recipient fax job; additional recipients can be added to the job file with the AddFaxRecipient method. To submit the job for faxing, place the APJ in the ApdFaxServerManager.MonitorDir folder. The ApdFaxClient can be used on a remote system to provide fax capability across a network.

Hierarchy

TComponent (VCL)

❶ TApdBaseComponent (OOMisc)	8
❷ TApdFaxJobHandler (AdFaxSrv)	769
TApdFaxClient (AdFaxSrv)	

Properties

CoverFileName	HeaderTitle	PhoneNumber
FaxFileName	JobFileName	ScheduledDateTime
HeaderLine	Recipient	Sender
HeaderRecipient	JobName	❶ Version

Methods

❷ AddRecipient	❷ ExtractCoverFile	❷ MakeJob
AddFaxRecipient	❷ GetJobHeader	❷ ShowFaxJobInfoDialog
❷ ConcatFaxes	❷ GetRecipient	
❷ ExtractAPF	MakeFaxJob	

Reference Section

AddFaxRecipient **method**

```
procedure AddFaxRecipient;
```

- ✚ Adds the Recipient record to an existing job file.

The AddFaxRecipient method will add the recipient information provided through the Recipient property to the fax job file specified in the JobFileName property. See the example under the MakeJob method for an illustration of the usage of this method.

See also: MakeFaxJob, Recipient

CoverFileName **property**

```
property CoverFileName : ShortString
```

Default: Empty string

- ✚ Determines the text file to include as a cover page.

CoverFileName determines the text file to include in the job file as a cover page. If a cover page is not desired, set this property to an empty string. See “Async Professional Job File format” on page 766 for details on job file cover pages.

FaxFileName **property**

```
property FaxFileName : ShortString
```

Default: empty string

- ✚ Determines the APF file to include in the job file.

Set FaxFileName to the file name of the fax file to be sent by this job. The file specified by this property must be in the Async Professional Fax format (APF).

HeaderLine **property**

```
property HeaderLine : ShortString
```

- ✚ The line of text that is sent at the top of each fax page.

A header line consists of normal text and replacement tags. A replacement tag is one of several characters prefixed with ‘\$’. When the header line is transmitted, the tags are replaced with appropriate text. The available replacement tags are listed in “Cover pages” on page 738.

No check is made to make sure your header line fits on a page. If your header line does not fit, it is truncated when it is transmitted. Using the default fonts, you can fit approximately 144 characters on a standard width page.

- ☛ **Caution:** Recently passed United States legislation makes it unlawful to send faxes within the United States without showing certain sender information on the fax. The new requirement states, in part: “It shall be unlawful for any person within the United States to use any computer or other electronic device to send any message via facsimile machine unless such message clearly contains, in a margin at the top or bottom of each transmitted page or on the first page of the transmission, the date and time it is sent and an identification of the business, other entity, or individual sending the message and the telephone number of the sending machine of such business, other entity, or individual.”

See also: `TApdSendFax.EnhHeaderFont`, `TApdSendFax.EnhTextEnabled`,
`TApdSendFax.HeaderSender`, `HeaderRecipient`, `HeaderTitle`

HeaderRecipient **property**

`property HeaderRecipient : ShortString`

- ☛ The fax recipient's name.

This string replaces the \$R replacement tag in a cover page text file or a header line.

See “Cover pages” on page 738 for more information and examples.

See also: `HeaderLine`, `TApdSendFax.HeaderSender`, `HeaderTitle`

HeaderTitle **property**

`property HeaderTitle : ShortString`

- ☛ The fax title.

This string replaces the \$\$ replacement tag in a cover page text file or a header line.

See “Cover pages” on page 738 for more information and examples.

See also: `HeaderLine`, `HeaderRecipient`, `TApdSendFax.HeaderSender`

JobFileName

property

```
property JobFileName : ShortString
```

Default: Empty string

↪ The name of the APJ file that is being handled.

Set the JobFileName property to the path and name of an existing APJ file to add recipients, or to a new name to create a new APJ. The MakeFaxJob and AddFaxRecipient methods use JobFileName to determine which APJ file to work with.

See also: AddFaxRecipient, MakeFaxJob

JobName

property

```
property JobName : ShortString
```

Default: Empty string;

↪ Determines the friendly name for this fax job.

Set JobName to a string summarizing the fax job. This property will be placed in the TFaxHeaderRec.JobName field when the MakeJob method is called. This field can be used for a user-friendly display of all pending fax jobs. This field is also used to determine the name of the extracted fax file and cover file when transmitting.

See also: MakeJob

MakeFaxJob

method

```
procedure MakeFaxJob;
```

↪ Creates a new APJ file containing information determined by other TApdFaxClient properties.

The MakeFaxJob method is used to create a new APJ file containing a single recipient. MakeFaxJob, in turn, calls the TApdFaxJobHandler.MakeJob method, passing the TApdFaxClient properties as parameters. The resulting APJ will be named according to the FaxJobName property. A TFaxHeaderRec containing the JobName and Sender properties; and the recipient information as defined by the Recipient property will be added. The CoverFileName and FaxFileName properties determine which cover page text and APF file to include. Additional recipients will need to be added using the AddFaxRecipient method.

The following example demonstrates how to create a fax job using MakeFaxJob method and how to add another recipient using the AddFaxRecipient method:

```
begin
  { set properties that apply to this job }
  ApdFaxClient.CoverFileName := 'C:\COVER.TXT';
  ApdFaxClient.FaxFileName := 'C:\TPFAX.APF';
  ApdFaxClient.JobFileName := 'C:\FAXSRVR\TPFAX.APJ';
  ApdFaxClient.JobName := 'Fax to TurboPower';
  ApdFaxClient.Sender := 'Mike';
  { set properties that apply to first recipient }
  ApdFaxClient.Recipient.PhoneNumber := '260 7151';
  ApdFaxClient.Recipient.HeaderLine :=
    'Fax from $F to $R, $D $T';
  ApdFaxClient.Recipient.HeaderRecipient := 'TurboPower';
  { send this job immediately }
  ApdFaxClient.Recipient.SchedDT := Now;
  { make the job }
  ApdFaxClient.MakeFaxJob;
  { set properties that apply to second recipient }
  ApdFaxClient.PhoneNumber := '555 1212';
  ApdFaxClient.HeaderRecipient := 'Purchasing department';
  { send this to the second recipient at 6pm tonight }
  ApdFaxClient.SchedDT := Date + 0.75;
  ApdFaxClient.AddRecipient;
end;
```

PhoneNumber

property

property PhoneNumber : ShortString

Default: Empty string

✚ Determines destination fax number for a single recipient.

Set the PhoneNumber property to the fax number of the recipient for this fax. This property is used in the MakeFaxJob and AddFaxRecipient methods to fill the TFaxRecipientRec.PhoneNumber field.

```
property Recipient : TFaxRecipientRec

TFaxRecipientRec = packed record
    Status : Byte;
    JobID : Byte;
    SchedDT : TDateTime;
    AttemptNum : Byte;
    LastResult : Word;
    PhoneNumber : String[50];
    HeaderLine : String[100];
    HeaderRecipient : String[30];
    HeaderTitle : String[30];
    Padding : Array[228..256] of Byte;
end;
```

↪ The TFaxRecipientRec containing information about a single recipient.

The Recipient property is a run-time only property that contains information about a fax recipient, and recipient-specific information. This fields of this property can also be referenced through the PhoneNumber, HeaderLine, HeaderRecipient, and HeaderTitle properties.

ScheduledDateTime**read-only property**

```
property ScheduledDateTime : TDateTime

Default: Now
```

↪ Determines when the current job will be scheduled.

The ScheduledDateTime property is used to schedule a fax job to a recipient at a specific time, or to implement a priority queuing system for the Fax Server Components. Set the ScheduledDateTime property to the TDateTime that you want the fax to be sent. The ScheduledDateTime property is used to fill the TFaxJobHeaderRec.SchedDT and TFaxRecipientRec.SchedDT fields.

Since the TApdFaxServerManager component uses the SchedDT field to determine which fax to return to the TApdFaxServer component, the ScheduledDateTime property can be used to implement a priority system. Set ScheduledDateTime to Now for normal priority faxes. Set ScheduledDateTime to Now – 1 for a higher priority fax. The job with a SchedDT that has already passed will always take precedence over a job that is scheduled for later.

Sender

property

property Sender : ShortString

Default: Empty string

↪ Designates the creator of this fax job.

The Sender property is used to fill the TFaxHeaderRec.Sender field, which is used for status displays and to show who generated the fax job. Set Sender to the name of the person creating the fax job, the name of the machine that it was created on, or any designation that describes the owner of the fax job.

TApdAbstractFaxStatus Class

TApdAbstractFaxStatus is an abstract class that defines the methods and properties needed by a component that automatically displays status while a TApdSendFax or TApdReceiveFax component is sending or receiving a fax. You generally won't need to create a descendent class of your own, since Async Professional supplies one, the TApdFaxStatus component (see page 826).

However, TApdFaxStatus shows a particular set of information about a fax transfer in a predefined format. If this format is not suitable for your needs, you can create your own descendant of TApdAbstractFaxStatus. The best way to start is to study the source code of TApdFaxStatus (in the AdFStat unit) and its associated form, TStandardFaxDisplay.

The TApdAbstractFaxStatus class contains an instance of a TForm that holds controls used to display the fax status. You design the form, create an instance, and assign the instance to the Display property of TApdAbstractFaxStatus.

TApdAbstractFaxStatus replaces the standard VCL properties Caption, Ctl3D, Position, and Visible and the standard VCL method Show. When these routines are used in the status component, the overridden versions perform the same actions on the associated Display form. Thus you can display the status form by calling Show, erase it by setting Visible to False, adjust its position by assigning to Position, and use 3D effects by setting Ctl3D to True.

Once you create an instance of your TApdAbstractFaxStatus descendant, you must assign it to the StatusDisplay property of your TApdSendFax or TApdReceiveFax component. When the background fax process needs to update the status display, it calls the UpdateDisplay method of TApdAbstractFaxStatus, which you must override to update your status window.

Hierarchy

TComponent (VCL)

- TApdBaseComponent (OOMisc) 8
- TApdAbstractFaxStatus (AdFax)

Properties

Display	Visible
Fax	● Version

Methods

CreateDisplay	DestroyDisplay	UpdateDisplay
---------------	----------------	---------------

Reference Section

CreateDisplay

dynamic abstract method

```
procedure CreateDisplay; dynamic; abstract;
```

↳ An abstract method that creates a form to display the fax status.

A descendant of `TApdAbstractFaxStatus` must override this method with a routine that creates a `TForm` component that contains various controls (typically of type `TLabel`) for displaying the fax status. The `TForm` should also contain a `TButton` control and associated `OnClick` event handler that allow the user to cancel the fax session.

`CreateDisplay` must then assign the instance of this form to the `Display` property.

See also: `DestroyDisplay`, `Display`

DestroyDisplay

dynamic abstract method

```
procedure DestroyDisplay; dynamic; abstract;
```

↳ An abstract method that destroys the display form.

A descendant of `TApdAbstractFaxStatus` must override this method to destroy the `TForm` instance created by `CreateDisplay`.

See also: `CreateDisplay`, `Display`

Display

run-time property

```
property Display : TForm
```

↳ A reference to the form created by `CreateDisplay`.

`CreateDisplay` must assign a properly initialized instance of a `TForm` to this property. `UpdateDisplay` can refer to this property to update the status window.

See also: `CreateDisplay`, `UpdateDisplay`

Fax

property

```
property Fax : TApdAbstractFax
```

↳ The fax component that is using the status component.

When you derive components from `TApdAbstractFaxStatus`, you will probably reference `TApdSendFax` or `TApdReceiveFax` properties to display information about the progress of the fax session. Use this property to do so. It is automatically initialized when you assign the status component to the `StatusDisplay` property of `TApdSendFax` or `TApdReceiveFax`.

```
procedure UpdateDisplay(  
    const First, Last : Boolean); virtual; abstract;
```

↪ An abstract method that writes the contents of the status window.

A descendant of `TApdAbstractFaxStatus` must override this method to update the display form. The `TApdSendFax` or `TApdReceiveFax` component calls this method regularly from its `OnFaxStatus` event handler.

On the first call to `UpdateDisplay`, `First` equals `True` and `UpdateDisplay` should call the `Show` method of `Display` to draw the outline and background of the status form. On the last call to `UpdateDisplay`, `Last` equals `True` and `UpdateDisplay` should set the `Visible` property of `Display` to `False` to erase the status window.

For all other calls to `UpdateDisplay`, `First` and `Last` are both `False`. During these calls `UpdateDisplay` should update the various labels in the `Display` form. To get information about the fax session, use the `Fax` field of `TApdAbstractFaxStatus` to read the values of various properties such as `FaxFile` and `BytesTransferred`. See “`TApdFaxStatus` Component” on page 826 for a list of the most commonly used properties.

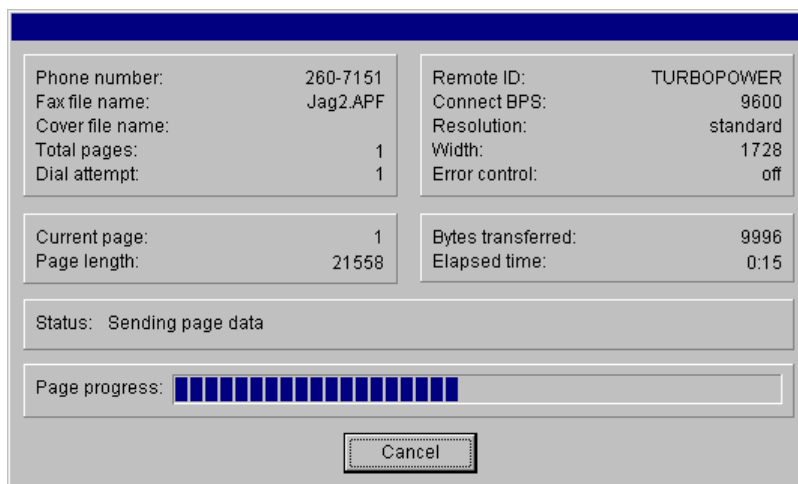
The `CancelClick` event handler, if one is provided, should call the `CancelFax` method of `TApdSendFax` or `TApdReceiveFax` to terminate the fax session.

TApdFaxStatus Component

TApdFaxStatus is a descendant of TApdAbstractFaxStatus that implements a standard fax status display. To use it, just create an instance and assign it to the StatusDisplay property of your TApdSendFax or TApdReceiveFax component. TApdFaxStatus includes all of the most frequently used information about a fax transfer and it provides a Cancel button so that the user can abort the session at any time.

TApdFaxStatus overrides all the abstract methods of TApdAbstractFaxStatus. TApdFaxStatus has no methods that you must call or properties that you must adjust. You might want to change the settings of the Ctl3D and Position properties to modify the appearance and placement of the window.

Figure 15.3 shows the TStandardFaxDisplay form that is associated with a TApdFaxStatus component.



The screenshot shows a Windows-style dialog box titled "TStandardFaxDisplay". It contains several fields organized into groups. The top group has two columns: the left column contains "Phone number: 260-7151", "Fax file name: Jag2.APF", "Cover file name:", "Total pages: 1", and "Dial attempt: 1"; the right column contains "Remote ID: TURBOPOWER", "Connect BPS: 9600", "Resolution: standard", "Width: 1728", and "Error control: off". Below this is another group with two columns: the left column contains "Current page: 1" and "Page length: 21558"; the right column contains "Bytes transferred: 9996" and "Elapsed time: 0:15". A status bar below these fields displays "Status: Sending page data". Underneath the status bar is a "Page progress:" label followed by a progress bar consisting of 15 blue vertical bars. At the bottom center of the dialog is a "Cancel" button.

Figure 15.3: TStandardFaxDisplay form.

For an example of using a TApdFaxStatus component, see the TApdSendFax example on page 740 or the TApdReceiveFax example on page 756.

Hierarchy

- TComponent (VCL)
 - TApdBaseComponent (OOMisc) 8
 - TApdAbstractFaxStatus (AdFax)822
 - TApdFaxStatus (AdFStat)

TApdFaxLog Class

TApdFaxLog is a small class that can be associated with a TApdSendFax or TApdReceiveFax to provide automatic fax logging services. Simply create an instance of TApdFaxLog and assign it to the FaxLog property of the TApdSendFax or TApdReceiveFax component.

TApdFaxLog creates or appends to a text file whose name is given by the FaxHistoryName property. Each time the OnFaxLog event is generated, the associated TApdFaxLog instance opens the file, writes a new line to it, and closes the file.

Following is a sample of the text file created by TApdFaxLog:

```
-----Receive skipped at 2/20/96 3:32:48 PM

Receive FAX0014.APF from 719 260 7151 started at 2/22/96 10:47:34 AM
Receive finished OK at 2/22/96 10:47:48 AM

Transmit BIG.APF to 260-7151 started at 2/26/96 2:23:41 PM
Transmit to 719 260 7151 finished OK at 2/26/96 2:24:14 PM

Transmit BIG.APF to 260-1643 started at 2/26/96 2:26:16 PM
Transmit failed at 2/26/96 2:26:41 PM
  (Cancel requested)

Transmit BIG.APF to 260-1643 started at 2/26/96 5:39:35 PM
Transmit failed at 2/26/96 5:40:17 PM
  (Called fax number was busy)
```

Hierarchy

TComponent (VCL)	
❶ TApdBaseComponent (OOMisc)	8
TApdFaxLog (AdFax)	

Properties

Fax	FaxHistoryName	❶ Version
-----	----------------	-----------

Methods

UpdateLog

Reference Section

Fax

property

```
property Fax : TApdCustomAbstractFax
```

↳ The fax component that is using the log component.

Fax is automatically initialized when the FaxLog property of the owning fax component is set. Programs can change Fax to assign the log component to a different fax component.

FaxHistoryName

property

```
property FaxHistoryName : string
```

Default: "APROFAX.HIS"

↳ Determines the name of the fax log file.

Set the value of FaxHistoryName before calling StartTransmit or StartReceive. However, because the log file is opened and closed for each update, you can change FaxHistoryName at any time. If you set FaxHistoryName to an empty string, automatic logging is disabled until you assign a non-empty string.

See also: TApdSendFax.StartReceive, TApdReceiveFax.StartTransmit

UpdateLog

virtual method

```
procedure UpdateLog(const Log : TFaxLogCode); virtual;
```

```
TFaxLogCode = (lfaxNone, lfaxTransmitStart, lfaxTransmitOk,  
    lfaxTransmitFail, lfaxReceiveStart, lfaxReceiveOk,  
    lfaxReceiveSkip, lfaxReceiveFail);
```

↳ Called for each fax logging event.

The Log parameter has the same values passed to the OnFaxLog event handler of TApdAbstractFax. UpdateLog creates or appends to the log file, builds and writes a text string for each event, and closes the log file.

TApdFaxLog contains a field named Fax that UpdateLog uses to obtain additional information (FaxFile, BytesTransferred, and CurrentPage) about the fax transfer.

See also: TApdAbstractFax.OnFaxLog

Fax Printer Drivers

An Async Professional fax printer driver is a Windows printer driver that generates APF format fax files when printed to from any Windows application. After the appropriate fax printer driver is installed on the user's system, it shows up in the list of available printers. The user can then choose to print to the fax printer driver rather than to a physical printer. The fax printer driver creates a fax file that can then be transmitted using the Async Professional fax components.

For example, you could implement a fax server program that sends faxes when the user prints to the fax printer driver. The user can simply print to the fax printer driver from any Windows application. Your fax server program could run in the background, waiting to be notified when printing to the fax printer driver begins and when it is finished. When the OnEndDoc event occurs, your fax server program could then send the fax using the StartTransmit method of the TApdSendFax component.

By default, when the Async Professional fax printer driver prints, it creates a "C:\DEFAULT.APF" fax file. To specify a different name or location, use a TApdFaxDriverInterface component.

The TApdFaxDriverInterface component provides control for the fax printer drivers. Async Professional provides printer drivers for Windows 95/98/ME, Windows NT 4.0, and Windows 2000.

Installation

A fax printer driver can be installed to your user's system either through code or through printer driver setup files (INF files). Installation can be accomplished using the provided installation units: PDrvInst, PDrvInNT, PDrvUni, and PDrvUnNT. The following example shows how to use these function. See the PINST example for a compilable file showing this functionality.

```
var
  QuietOperation : Boolean;
  { Suppresses success/failure prompts when True }
begin
  QuietOperation := (ParamCount > 0) and
    (pos('Q',UpperCase(paramStr(1))) <> 0) ;
  if IsWinNT then
    InstallDriver32('')
  else
    InstallDriver('APFGEN.DRV');
```

```

if not QuietOperation then
case DrvInstallError of
ecOK :
    MessageDlg('APF Fax Printer Driver Installed OK',
        mtInformation, {mbOK}, 0 ;
ecUniAlreadyInstalled : ;
ecUniCannotGetSysDir :
    MessageDlg('Couldn't determine Windows\System directory',
        mtError, {mbOK}, 0) ;
ecUniCannotGetPrinterDriverDir
    MessageDlg('Couldn't determine Windows NT printer driver
        directory', mtError, {mbOK}, 0) ;
ecUniCannotGetWinDir :
    MessageDlg('Couldn't determine Windows directory',
        mtError, {mbOK}, 0) ;
ecUniUnknownLayout :
    MessageDlg('    -- Unknown Windows Layout --'+#13+
        'Unable to locate require support'+#13+
        'files', mtError, [mbOK], 0) ;
ecUniCannotParseInfFile :
    MessageDlg('Cannot locate unidriver files in'+#13+
        'Windows Setup (INF) file.', mtError, [mbOK], 0) ;
ecUniCannotInstallFile
    MessageDlg('Unidriver files'+#13+'not installed. The print
        driver'+#13+'may not be configured properly.',
        mtError, [mbOK], 0) ;
ecNotNTDriver :
    MessageDlg('This printer driver is not compatible with
        Windows NT', mtError, [mbOK], 0) ;
ecDrvCopyError :
    MessageDlg('Unable to copy printer driver to Windows
        system directory', mtError, [mbOK], 0) ;
ecCannotAddPrinter :
    MessageDlg('Could not install APFGEN.DRV as a Windows
        printer', mtError, [mbOK], 0) ;
ecDrvBadResources :
    MessageDlg('Printer driver file contains bad or missing
        string resources', mtError, [mbOK], 0) ;
ecDrvDriverNotFound :
    MessageDlg('A required printer driver file was not found',
        mtError, [mbOK], 0) ;
else
    MessageDlg('Unknown installation error : '+
        IntToStr(DrvInstallError), mtError, [mbOK], 0) ;
end;
end.

```

The Async Professional fax printer drivers rely on several files supplied by Microsoft. These are usually located on your Windows installation medium. The InstallDriver32 and InstallDriver methods will first verify whether these files are already installed. If they are not installed, a dialog will be displayed where the user can browse for the files. The following list shows the required files (APF* are supplied in the APRO\Redist folder, others are supplied by Microsoft)

Required printer driver files:

Windows 95/98/ME

- APFGEN.DRV
- UNIDRV.DLL
- UNIDRV.HLP
- ICONLIB.DLL

Windows NT 4/2000

- APFPDENT.DLL
- APFMON40.DLL
- APFAXCNV.DLL
- RASDD.DLL
- RASDDUI.DLL
- RASDDUI.HLP

Note: Windows 2000 Professional does not contain the RASDD* files required by our printer driver. These files can be obtained from a Windows NT4 or Windows 2000 Server installation CD.

Recompiling

It is possible to rebuild the Async Professional fax printer drivers, but it is seldom required to do so. Before undertaking the build process, first determine that the existing design is not suitable to your application. Most developers who want to rebuild the printer drivers do so because they want to build in functionality that is already supported by the `TApdFaxDriverInterface`.

Async Professional provides two fax printer drivers: one for Windows 95/98/ME, another for Windows NT/2000. The Windows 95/98/ME printer subsystem is 16-bit, and it requires a 16-bit compiler (Delphi 1) to compile the `APFGEN.DRV` printer driver. The Windows NT/2000 printer subsystem is 32-bit, building all three parts of the driver requires a 32-bit Delphi compiler and Microsoft Visual C++ 4.0 or later.

For the latest information concerning rebuilding the Async Professional printer drivers, search the TurboPower KnowledgeBase (<http://www.turbopower.com/search/>) for “printer driver.”

TApdFaxDriverInterface Component

The TApdFaxDriverInterface component provides control for the fax printer drivers. Using this component you can set the output file name that the fax printer driver uses and receive notification when the fax printer driver has finished writing the fax file.

Hierarchy

TComponent (VCL)

- ❶ TApdBaseComponent (OOMisc) 8
 - TApdFaxDriverInterface (AdFaxCtl)

Properties

DocName	FileName	❶ Version
---------	----------	-----------

Events

OnDocEnd	OnDocStart
----------	------------

Reference Section

DocName	read-only, run-time property
----------------	-------------------------------------

```
property DocName : string
```

↪ Contains the name of the print job as it appears in the print spooler.

DocName is a read-only property that contains a textual description of the document that is being printed.

FileName	property
-----------------	-----------------

```
property FileName : string
```

Default: "C:\DEFAULT.APF"

↪ Contains the name of the output file used by the printer driver.

FileName holds the name of the file used by the printer driver to store the output. The default file name is C:\DEFAULT.APF.

OnDocEnd	event
-----------------	--------------

```
property OnDocEnd : TNotifyEvent
```

↪ Defines an event handler that is called when the printer driver finishes processing a print job.

When a fax print server application receives this event, it can take some further action on the file (e.g., send it or do some other processing on it).

The following example uses the OnDocEnd event for notification that the fax printer driver has finished creating the fax file. When notification is received, the fax file is transmitted.

```
procedure TForm1.ApdFaxDriverInterface1DocEnd(Sender : TObject);
begin
  { Done printing to the fax file so send the fax }
  ApdSendFax1.FaxFile := ApdFaxDriverInterface1.FileName;
  ApdSendFax1.PhoneNumber := '260-7151';
  ApdSendFax1.StartTransmit;
end;
```

See also: OnDocStart

```
property OnDocStart : TNotifyEvent
```

↳ Defines an event handler that is called when the printer driver is ready to start printing a new document.

This event is primarily used to supply an output file name for the printer driver to use. Control does not return to the printer driver until the event handler is done.

The following example uses OnDocStart to set the output file name prior to the fax printer driver writing the fax file:

```
procedure TForm1.ApdFaxDriverInterfacelDocStart(  
    Sender : TObject);  
begin  
    ApdFaxDriverInterfacel.FileName := 'TEMPFAX.APF';  
end;
```

See also: FileName, OnDocEnd

Chapter 16: Paging Components

The ability to locate and send some kind of message to someone carrying a small electronic device (a pager) has become a mainstay of modern existence. Once the province of the wealthy and of high level business figures; one now sees pagers hanging from the belts and purses of dock workers, secretaries, and even teenagers.

The ability to send and receive alphanumeric messages (i.e., those containing textual information beyond the simple digits easily entered from a telephone keypad), is increasingly common among modern paging equipment.

The desire to be able to send and receive such information has also increased, perhaps somewhat in advance of the technology for doing so, as until recently the primary method for entering and sending such data has been restricted to proprietary (and often expensive and difficult to use) hardware devices or computer software provided by paging services.

The fairly recent development of new standards for preparing and sending alphanumeric messages has made it possible to create generalized software solutions for performing these tasks. However, these “protocols” remain somewhat obscure and awkward to implement at the basic software level.

Given Async Professional's mission of providing well structured and encapsulated solutions to serial, telephony, and basic TCP/IP communications needs; and with the new availability of standardized means of performing the required transmissions, Async Professional now includes paging components.

Sending Alphanumeric Pages

Async Professional provides components that support sending messages to alphanumeric paging devices using either TAP (Telelocator Alphanumeric Protocol) with a modem over a phone line, or SNPP (Simple Network Paging Protocol) over Internet using a TCP/IP socket connection.

While these two protocols are fairly different in their implementations, the Async Professional paging components provide a consistent interface to sending messages regardless of which transmission medium is in use.

Requirements

The primary difference between TAP paging and SNPP paging from the Async Professional programmer's perspective is that `TApdSNPPPager` requires a `TApdWinsockPort` to work, as it needs an open TCP/IP socket in order to transmit. `TApdTAPPager` can use either a `TApdComPort` or a `TApdWinsockPort` using the appropriate `DeviceLayer` property setting.

In order to successfully send a page by either protocol, the recipient's pager must be serviced by a "Paging Server" that responds to the relevant protocol. A TAP server will typically have a phone number to dial, which is almost certainly not the same number that is normally called to access the pager directly. An SNPP server will have an IP address associated with it, as well as a port number.

Values for these paging parameters must be entered into the relevant properties of a paging component before paging may be performed. The programmer will have to gather this information from the intended recipients (or from their paging companies) in order to send pages successfully with the Async Professional paging components.

Certain additional requirements are generally automatically handled by the components, see the individual component descriptions for more details.

TApdAbstractPager Component

All of the Async Professional paging components are descended from `TApdAbstractPager`. This abstract paging component provides a set of properties and methods common to sending a page regardless of the transmission medium, such as the Pager ID, the message text to be sent, and a method to actually send the page.

Hierarchy

`TComponent` (VCL)

`TapdAbstractPager` (`AdPager`)

Properties

`Message`

`PagerID`

`PagerLog`

Methods

`Send`

Reference Section

Message

property

```
property Message : TStrings
```

- ✚ Contains the text of the alphanumeric message to be sent as a page.

As a TStrings, message may be filled using all the methods open to TStrings and TStrings descendants. The following code will fill the Message property from the contents of a TMemo:

```
ApdPager1.Message.Assign(Memo1.Lines);
```

PagerID

property

```
property PagerID : string
```

- ✚ Defines the identification string (frequently the phone number) of the pager to which the message is being sent.

PagerLog

property

```
property PagerLog : TApdPagerLog
```

- ✚ An instance of a pager logging component.

If PagerLog is nil (the default), the pager component does not perform automatic logging. You can install pager component event handlers to perform logging in this case.

If you create an instance of (or a descendant of) a TApdPagerLog component (see page 862), and assign it to PagerLog, the pager component will call the log component's UpdateLog method automatically.

Send

method

```
procedure Send; virtual; abstract;
```

- ✚ Provides a “root” for send methods in descendant components.

Send is a virtual, abstract method. All descendants must override this method to implement their protocol and transmission medium specific behaviors. See the implementations of Send for TApdTAPPager and TApdSNPPPager for more details.

TApdTAPPager Component

The TApdTAPPager Component is used to send alphanumeric pages to paging services that support Telelocator Alphanumeric Protocol (TAP) also known as the “IXO” protocol and the Motorola Personal Entry Terminal (PET) protocol. It publishes properties and methods inherited from the TApdCustomModemPager that provide logic for call management: dialing the telephone, detecting busy signals, redialing on an error, waiting to redial, etc.

Dialing events

The TApdTAPPager component goes through a number of stages in the process of sending a page, these can be separated into broad categories of Errors in modem dialing: modem/line states as the call progresses, and events/errors specific to the actual TAP protocol.

The TApdTAPPager component provides the mechanism for tracking the first two kinds of event in the OnDialError and OnDialEvent event handler properties (see page 849). The TApdTAPPager component provides a further level of status processing for TAP specific events via the OnTAPStatus event handler. (See page 851.)

Dialing error handling

The Error parameter to OnDialError is of type TDialError, which is a subrange (deNone..deNoConnection) of the TDialingCondition enumerated type that indicates the detected Error condition at the time OnDialError event is tripped. The values that TDialError defines are shown in Table 16.1.

Table 16.1: *TDialError values*

Value	Explanation
deNone	Un-initialized starting state.
deNoDialTone	Modem reported no dial tone.
deLineBusy	Modem reported line busy.
deNoConnection	Modem unable to establish connection.

Dialing status

The Event parameter to OnDialStatus is of type TDialingStatus, which is a subrange (dsNone..dsCleanup) of the TDialingCondition enumerated type that indicates the detected status condition at the time OnDialStatus event is tripped. The values defined by TDialingStatus are described in Table 16.2.

Table 16.2: *TDialingStatus* values

Value	Explanation
dsNone	Un-initialized starting state.
q	Modem has gone off hook.
dsDialing	Modem dialing phone number.
dsRinging	Line is ringing.
dsWaitForConnect	Line answered waiting for connection negotiation.
dsConnected	Line connected.
dsWaitingToRedial	In redial wait period.
dsRedialing	Dialing a number again.
dsMsgNotSent	No message sent.
dsCancelling	Call request cancelled by user.
dsDisconnect	Line disconnected.
dsCleanup	Doing post call cleanup.

TAP paging status

During the course of a TAP Page transmission, the TAP server returns several codes indicating the success or failure of a particular stage in the connection. TAPdTAPPager provides an additional mechanism (over that provided by its ancestor's) for tracking TAP specific events and errors.

The Event parameter to OnTAPStatus is of an enumerated type (TTAPStatus) that indicates the detected TAP status at the time OnTAPStatus event is tripped. Table 16.3 describes the values which TTAPStatus defines.

Table 16.3: *TTAPStatus* values

Value	Explanation
psNone	Un-initialized starting state.
psLoginPrompt	TAP server login prompt received.
psLoggedIn	Successful login to TAP server.
psLoginErr	Error in TAP login procedure.
psLoginFail	TAP Login procedure failure.
psMsgOkToSend	TAP server ready for message transmission.
psSendingMsg	TAPdTAPPager sending message.

Table 16.3: *TTAPStatus* values

Value	Explanation
psMsgAck	TAP server acknowledged (successfully received) message block.
psMsgNak	TAP server does not acknowledge message block; unable to proceed.
psMsgRs	TAP server does not acknowledge message block; resend block.
psMsgCompleted	TAP server received entire message successfully.
psDone	TAP server logout.

Hierarchy

TComponent (VCL)

- ❶ TApdBaseComponent (OOMisc) 8
 - ❷ TApdAbstractPager (AdPager) 839
 - TApdCustomModemPager (AdPager)
 - TApdTAPPager (AdPager)

Properties

AbortNoConnect	ExitOnError	Port
BlindDial	MaxMessageLength	TAPStatusMsg
DialAttempt	Message	ToneDial
DialAttempts	ModemInit	UseEscapes
DialPrefix	❷ PagerID	UseTapi
DialRetryWait	PagerLog	❶ Version
DialWait	PhoneNumber	

Methods

CancelCall	Disconnect	Send
DialStatusMsg	ReSend	

Events

OnDialError	OnTAPFinish
OnDialStatus	OnTAPStatus

Reference Section

AbortNoConnect

property

```
property AbortNoConnect : Boolean
```

Default: `adpgDefAbortNoConnect`

- ↳ Defines what happens when the connection to a paging terminal number cannot be made after the default number of retries.

If `True`, the paging process ends and generates an `OnDialStatus` event with `dsCancelling`. If `False`, the pager re-attempts the page as defined by the `DialAttempts` property.

See also: `DialAttempts`, `OnDialStatus`

BlindDial

property

```
property BlindDial : Boolean
```

Default: `adpgDefBlindDial`

- ↳ Allows a page to be sent regardless of whether the modem detects a dial tone.

If `BlindDial` is `True`, a different initialization sequence is sent to the modem before a page is sent (ATX3 is sent instead of ATX4). This initialization sequence allows the modem to use a phone line, even if it can't detect a dial tone.

CancelCall

method

```
procedure CancelCall;
```

- ↳ Cancels the current phone call being sent.

`TApdTAPPager` sends the TAP “Cancel” command to the remote TAP server before proceeding with shut down. It essentially terminates the call processing logic. Any custom `DoDisconnect`, `DoFailedToSend`, or `DoCloseCall` logic is processed.

The following example shows how to cancel a page:

```
procedure TForm1.Button1Click(Sender : TObject);  
begin  
    ApdTAPPager1.CancelCall;  
end;
```

See also: `OnDialStatus`, `OnDialError`

DialAttempt

property

```
property DialAttempt : Word
```

↳ Indicates the number of times the current paging server number has been dialed.

If the dialed number is busy, TApdTAPPager waits briefly and calls the number again. It tries up to DialAttempts times. The DialAttempt property returns the number of the current attempt. DialAttempt is incremented immediately upon encountering a busy line.

See also: DialAttempts, DialRetryWait

DialAttempts

property

```
property DialAttempts : Word
```

Default: adpgDefDialAttempts

↳ Determines the number of times to automatically dial a paging server number.

This is the number of times a page is attempted, it is not the number of retries. When DialAttempts is one, for example, the number is dialed only once. If the line is busy, it is not tried again. When DialAttempts is three, the paging server number is dialed a maximum of three times.

See also: DialAttempt, DialRetryWait

DialPrefix

property

```
property DialPrefix : TModemString
```

```
TModemString = string[40];
```

↳ An optional dial prefix.

DialPrefix specifies an optional dial prefix that is inserted in the dial command between “ATDT” and the number to dial. If the telephone system requires special numbers or codes when dialing out, they can be specified once here rather than in every pager number.

Do not include “ATD” or a ‘T’ or ‘P’ tone/pulse modifier in the dial prefix. “ATD” is automatically prefixed by Send and the ‘T’ or ‘P’ is controlled by ToneDial.

See also: Send, ToneDial

DialRetryWait

property

property DialRetryWait : Word

Default: adpgDefDialRetryWait

↳ The number of seconds to wait after a busy signal before trying the number again.

After encountering a busy signal, TApdTAPPager checks to see if it should try this number again by comparing DialAttempts to DialAttempt. If more attempts are required, it waits DialRetryWait seconds before dialing again to give the dialed paging server time to have an open line.

If no more dialing attempts are required, TApdTAPPager does not wait, but immediately progresses to Canceling the call.

See also: DialAttempt, DialAttempts

DialStatusMsg

method

function DialStatusMsg(Status : TDialingCondition) : string;

↳ Returns an English string for a call progress status or error code.

This routine is intended primarily for use in calling status routines. It returns a status string from the string table resource linked into your EXE. The string ID numbers correspond to the values of the TDialingCondition enumerated type (see “Dialing status” on page 841). If the string table doesn’t contain a string resource with the requested ID, an empty string is returned.

The returned string is never longer than MaxMessageLen (80) characters.

DialWait

property

property DialWait : Word

Default: adpgDefDialWait

↳ The number of seconds to wait for a connection after dialing the number.

The default is 60 seconds.

Disconnect

method

```
procedure Disconnect;
```

- ✚ Sends the TAP “logout” code to the paging server.

ExitOnError

property

```
property ExitOnError : Boolean
```

Default: `adpgDefExitOnError`

- ✚ Determines what happens when an error occurs during an attempt to send a page.

If `ExitOnError` is `True`, no more attempts are made to send the page, regardless of the status of `DialAttempts`. If `ExitOnError` is `False` (the default), the paging component continues to try dialing the paging server.

See also: `AbortNoConnect`, `DialAttempts`

MaxMessageLength

property

```
property MaxMessageLength : Integer
```

Default: `MAX_MSG_LEN` (80)

- ✚ Defines a maximum length for message blocks sent to the paging server.

The TAP specification permits blocks of up to 256 characters (including all delimiters, which reduces actual message data to about 250 characters). Some paging servers seem to have trouble with blocks of over 80 characters however; thus the default.

Also, some TAP paging servers will not allow a single message to a single pager to exceed 256 characters total length.

Message

property

```
property Message : TStrings
```

- ✚ Contains the text of the alphanumeric message to be sent as a page.

See `UseEscapes` on page 854 for how to embed control characters inside TAP messages.

```
property ModemInit : TModemString
```

```
TModemString = string[40];
```

↳ A custom modem initialization string.

If you assign a custom modem initialization string to `ModemInit`, `TApdTAPPager` always sends this string to the modem just before it dials the paging server number. This occurs whenever you call `Send`.

The string should not contain an “AT” prefix or a trailing carriage return.

See also: `Send`

```
property OnDialError : TDialErrorEvent
```

```
TDialErrorEvent = procedure(  
    Sender : TObject; Error : TDialError) of object;
```

```
TDialingCondition = (dsNone, dsOffHook, dsDialing, dsRinging,  
    dsWaitForConnect, dsConnected, dsWaitingToRedial, dsRedialing,  
    dsMsgNotSent, dsCancelling, dsDisconnect, dsCleanup, deNone,  
    deNoDialTone, deLineBusy, deNoConnection);
```

```
TDialError = deNone..deNoConnection;
```

↳ Defines an event handler that is called when an error occurs in the dialing procedure.

`Sender` is the pager component that generated the error. `Error` is a `TDialError` value number indicating the type of error. See “Dialing error handling” on page 841 for the meaning of these error codes.

See also: `OnDialStatus`

```
property OnDialStatus : TDialStatusEvent

TDialStatusEvent = procedure(
    Sender : TObject; Event : TDialingStatus) of object;

TDialingCondition = (dsNone, dsOffHook, dsDialing, dsRinging,
    dsWaitForConnect, dsConnected, dsWaitingToRedial, dsRedialing,
    dsMsgNotSent, dsCancelling, dsDisconnect, dsCleanup, deNone,
    deNoDialTone, deLineBusy, deNoConnection);

TDialingStatus = dsNone..dsCleanup;
```

↳ Defines an event handler that is called regularly during a page call.

This event is generated after the completion of each major operation (e.g., going off hook, dialing). You can use this event to update a status display that informs the user about the progress of the call.

Sender is the pager component that is in progress. Event is a value of TDialingStatus type that indicates which phase of the call is in progress. See “Dialing status” on page 841 for the meaning of these status codes.

OnTAPFinish**event**

```
property OnTAPFinish : TNotifyEvent
```

↳ Defines an event handler that is called when a TAP page operation completes.

The OnTAPFinish event occurs when the TApdTAPPager component has received a connection termination code from the TAP server. It indicates that the TAP communication was completed (whether successfully or not). This may be used in logic for sending multiple pages to know when a given page attempt has completed.

To determine the success vs. failure of a page attempt, use the OnPageStatus event.

See also: OnTAPStatus

```
property OnTAPStatus : TTAPStatusEvent

TTAPStatus = (psNone, psLoginPrompt, psLoggedIn, psLoginErr,
  psLoginFail, psMsgOkToSend, psSendingMsg, psMsgAck, psMsgNak,
  psMsgRs, psMsgCompleted, psDone);

TTAPStatusEvent = procedure(
  Sender : TObject; Event : TTapStatus) of object;
```

- ↳ Defines an event handler that is called regularly during communication with the paging server.

This event is generated after the completion of stages in connecting with the paging server (e.g. logging in, sending the message). You can use it to update a status display that informs the user about the progress of the page connection.

Sender is the pager component that is in progress. Event is of type TTAPStatusEvent which indicates which phase of the communication is in progress. For a description of the stages, see “TAP paging status” on page 842.

PagerLog**property**

```
property PagerLog : TApdPagerLog
```

- ↳ An instance of a pager logging component.

If PagerLog is nil (the default), the pager component does not perform automatic logging. You can install pager component event handlers to perform logging in this case.

If you create an instance of (or a descendant of) a TApdPagerLog component (see page 862), and assign it to PagerLog, the pager component will call the log component’s UpdateLog method automatically.

PhoneNumber**property**

```
property PhoneNumber : string
```

- ↳ Defines the phone number to be dialed to access the Alphanumeric Paging Server.

The phone number is usually not the pager’s phone number. Set PhoneNumber to the number to dial prior to calling Send. If the phone system requires prefix codes (like ‘9’), the codes must be specified in PhoneNumber or in DialPrefix.

PhoneNumber can be used with status and logging routines to return the phone number dialed for the current page.

See also: DialPrefix, Send

```
property Port : TApdCustomComPort
```

↳ Determines the serial port used by the Modem Pager component.

A properly initialized comPort component must be assigned to this property before sending pages. When a TApdComPort is assigned to this property, the pager component forces the Port to the property values shown in the following example:

```
ApdComPort1.DataBits := 7;
ApdComPort1.StopBits := 1;
ApdComPort1.Parity := pEven;
ApdComPort1.Baud := 9600;
```

These values are the most common requirements for TAP paging servers and should be changed only if you are certain of the impact of those changes.

You can set these properties manually in code prior to sending a page. If you wish to make the change permanent to the component, you will need either to alter the source code, or create a descendant component of TApdTAPPager that overrides the SetPortOpts method.

ReSend**method**

```
procedure ReSend;
```

↳ Causes the pager component to attempt to resend the paging message.

This is only effective when logged into the paging terminal.

Send**method**

```
procedure Send;
```

↳ Causes the paging component to dial the phone number specified in PhoneNumber.

The number may be modified by DialPrefix. If it receives a successful answer, Send calls the protected method DoDial, which in turn (potentially) calls the virtual DoStartCall, DoCloseCall, DoDisconnect, and DoFailedToSend methods. The programmer creating a custom Modem pager should override these methods to provide protocol specific processing for the call progress.

The following example shows how to send a simple page with the TApdTAPPager component:

```
procedure TForm1.Button1Click(Sender : TObject);
begin
  ApdTAPPager1.PhoneNumber := '555-1234';
  ApdTAPPager1.PagerID := '12345';
  ApdTAPPager1.Message.Add('Hi There!');
  ApdTAPPager1.Send;
end;
```

See also: DialPrefix, PhoneNumber

TAPStatusMsg

method

```
function TAPStatusMsg(Status : TTAPStatus) : string;
```

↳ Returns an English string for a TAP status code.

This routine is intended primarily for use in status monitoring routines. It returns a status string from the string table resource linked into your EXE. The string ID numbers correspond to the values of the TTAPStatus constants (see “TAP paging status” on page 842). If the string table doesn’t contain a string resource with the requested ID, an empty string is returned.

The returned string is never longer than MaxMessageLen (80) characters.

ToneDial

property

```
property ToneDial : Boolean
```

Default: adpgDefToneDial

↳ Determines whether tone or pulse dialing is used for paging calls.

If ToneDial is True (the default), tone dialing is used. Otherwise, pulse dialing is used. Setting ToneDial does not immediately issue a modem command, but determines whether ‘T’ or ‘P’ is added to the dial command later.

See also: DialPrefix

property UseEscapes : Boolean

Default: False

- ↳ Determines whether or not the paging component scans the text of Message prior to sending and expands any “control characters” found into TAP escapes.

If True, escapes are processed according to TAP 1.8. If False, escapes are stripped from the message.

Version 1.8 of the TAP specification allows for embedding control characters in pages by use of an escaping mechanism. Basically a special character (ASCII #26: “SUB”, otherwise known as Ctrl-Z), followed by the desired character whose ASCII value has been incremented by 64.

For example, Ctrl-H (ASCII #8, “BS”, or BackSpace) would be sent as #26‘H’; and the escape character (ASCII #27) would be sent as #26‘[’.

TApdTAPPager allows such characters to be embedded in the Message property using the standard Borland Pascal notations, making it easier to create such messages without resorting to complex string constructions at run time.

You may use ‘#’ style character constants with either decimal or hex numeric values; and you may use ‘^’ letter values. For example, the Ctrl-H character mentioned above may be represented as “#8” (decimal), “#\$08” (hexadecimal), or “^H” in the string and TApdTAPPageSend will convert those to the proper #26‘H’ sequences when the page is sent.

Some paging terminals are not using the TAP 1.8 specification and don’t provide this mechanism. Such terminals’ response to the presence of the escapes is “unpredictable” (some just pass them through, converting the #26s to spaces, others freeze). Because of this, TApdTAPPager provides the UseEscapes property. If set True the above conversion is performed as stated; if False (the default) escape sequences of the above types are stripped from the message before sending.

property UseTapi : Boolean

Default: False

- ↪ UseTapi determines whether or not the Pager component uses TAPI (“Telephony API”) to present dialable devices to the user.

This is desirable because it allows the user to pick the port/modem by installed name rather than having to identify it solely by port number.

TApdSNPPPager Component

TApdSNPPPager is an implementation of Internet based paging using the Simple Network Paging Protocol (SNPP) standard as defined in RFC 1861.

TApdSNPPPager implements the simplest form (“Level One”) of SNPP transaction, and automates sending so that even some Level One commands are irrelevant (e.g. “HELP”).

Hierarchy

TComponent (VCL)

- ❶ TApdBaseComponent (OOMisc) 8
- ❷ TApdAbstractPager (AdPager) 839
 - TApdCustomINetPager (AdPager)
 - TApdSNPPPager (AdPager)

Properties

CommDelay	Send	ServerResponseFailTerminate
❷ Message	ServerDataInput	ServerSuccessString
❷ PagerID	ServerDoneString	❶ Version
❷ PagerLog	ServerInitString	
Port	ServerResponseFailContinue	

Methods

Send

Events

OnLogin	OnSNPPError
OnLogout	OnSNPPSuccess

Reference Section

CommDelay

property

```
property CommDelay : Integer
```

Default: 1

- ↳ The number of seconds to insert between sending SNPP commands.

CommDelay is intended to give the TapdSNPPPager component user some control if the SNPP server does not respond in a timely fashion to commands.

OnLogin

property

```
property OnLogin : TNotifyEvent
```

- ↳ Defines an event handler that is called when the TapdSNPPPager detects the ServerInitString.

OnLogout

event

```
property OnLogout : TNotifyEvent
```

- ↳ Defines an event handler that is called when a SNPP page operation logs out of the paging server.

The OnLogout event occurs when the TapdSNPPPager component has sent the SNPP “QUIT” command and received a connection termination code from the SNPP server. It indicates that the SNPP communication was completed (whether the page was sent successfully or not) and successfully logging out of the paging server has occurred. This may be used in logic for sending multiple pages to know when a given page attempt has completed.

To determine the success vs. failure of a page attempt, monitor the OnSNPPError and OnSNPPSuccess events.

See also: OnLogin, OnSNPPError, OnSNPPSuccess

OnSNPPError

property

property OnSNPPError : TSNPPMessage

```
TSNPPMessage = procedure(  
    Sender : TObject; Code : Integer; Msg : string) of object;
```

- ↳ Defines an event handler that is called whenever TapdSNPPPager detects a “400” or “500-Level” error code returned from the SNPP server.

Sender is the TapdSNPPPager component generating the event. Code is the numeric code of the error. Msg is the message string associated with the error as returned by the SNPP server.

OnSNPPSuccess

property

property OnSNPPSuccess : TSNPPMessage

```
TSNPPMessage = procedure(  
    Sender : TObject; Code : Integer; Msg : string) of object;
```

- ↳ Defines an event handler that is fired whenever TapdSNPPPager detects a “200-Level” success/proceed response code returned from the SNPP server.

Sender is the TapdSNPPPager component generating the event. Code is the numeric code of the success response. Msg is the message string associated with the response as returned by the SNPP server.

TapdSNPPPager overrides the TapdAbstractPager.Send method to implement the required behaviors for SNPP.

Port

property

property Port : TapdWinsockPort

- ↳ Determines the Winsock communications port used by the TapdCustomINet pager component.

A properly initialized Winsock Port component must be assigned to this property before pages can be transmitted over TCP/IP.

```
procedure Send;
```

- ↳ Causes TApdSNPPPager to connect and send a page.

When you call Send, the TApdSNPPPager component instructs its associated TApdWinsockPort component to initiate a TCP/IP connection. The TApdWinsockPort uses the IP address set in its wsAddress property, and the IP port defined in its wsPort property to make the connection.

TApdSNPPPager waits for the response defined in the ServerInitString property and then begins an SNPP transaction; first sending the desired destination for the page (in the PagerID property), then the text of the message (in the Message property); then it instructs the SNPP server to send. Upon receipt of a successful send TApdSNPPPager logs out of the SNPP server.

The following example shows how to send a simple page with the TApdSNPPPager component:

```
procedure TForm1.Button1Click(Sender : TObject);
begin
    ApdWinsockPort1.WsAddress := 'snpp.myservice.com';
    ApdWinsockPort1.WsPort    := '9999';
    ApdSNPPPager1.PagerID := '12345';
    ApdSNPPPager1.Message.Add('Hi There!');
    ApdSNPPPager1.Send;
end;
```

ServerDataInput

property

```
property ServerDataInput : string
```

Default: SNPP_RESP_DATAINPUT (“3??”)

- ↳ Defines the string that indicates the server is ready for multi-line input.

The ServerDataInput property is the string that the user wishes TApdSNPPPager to watch for to indicate that the SNPP server has recognized the request to send a multi-line command and is waiting for input. This response can vary among servers, but is generally prefixed by a success response code in the range 300-399.

The string may contain wildcards in the form of question marks to indicate that any character (usually a digit) is acceptable at that point.

ServerDoneString

property

```
property ServerDoneString : string
```

Default: “221”

↳ Defines the string that indicates server log-out.

The `ServerDoneString` property is the string that the user wishes `TapdSNPPPager` to watch for to indicate that the SNPP server is responding to the SNPP “QUIT” command and has logged out of the paging session. This response may vary among servers, but the common response seems to be prefixed by a code of 221.

The string may contain wildcards in the form of question marks to indicate that any character (usually a digit) is acceptable at that point.

ServerInitString

property

```
property ServerInitString : string
```

Default: “220”

↳ Defines the string indicating successful login.

The `ServerInitString` property is the string that the user wishes `TapdSNPPPager` to watch for to indicate that the SNPP server has responded to login and is ready to receive SNPP commands. This may vary among servers, but is generally prefixed by a “220” success response code.

The string may contain wildcards in the form of question marks to indicate that any character (usually a digit) is acceptable at that point.

ServerResponseFailContinue

property

```
property ServerResponseFailContinue : string
```

Default: `SNPP_RESP_FAILCONTINUE` (“5??”)

↳ Defines the string that indicates a non-fatal error in the paging transaction.

The `ServerResponseFailContinue` property is the string that the user wishes `TapdSNPPPager` to watch for to indicate that the SNPP server has encountered a problem with the SNPP transaction but is able to proceed. These responses may vary among servers, but they are generally prefixed by a code in the range 500-599.

The string may contain wildcards in the form of question marks to indicate that any character (usually a digit) is acceptable at that point.

ServerResponseFailTerminate

property

```
property ServerResponseFailTerminate : string
```

Default: SNPP_RESP_FAILTERMINATE (“4??”)

↳ Defines the string that indicates a fatal error in the paging transaction.

The ServerResponseFailTerminate property is the string that the user wishes TApdSNPPPager to watch for to indicate that the SNPP server has encountered a problem with the SNPP transaction from which it is unable to recover; often the SNPP server will shut down upon such an event.

These responses may vary among servers, but they are generally prefixed by a code in the range 400-499. The string may contain wildcards in the form of question marks to indicate that any character (usually a digit) is acceptable at that point.

ServerSuccessString

property

```
property ServerSuccessString : string
```

Default: SNPP_RESP_SUCCESS (“25?”)

↳ Defines the string that indicates a processing of a paging command.

The ServerSuccessString property is the string that the user wishes TApdSNPPPager to watch for to indicate that the SNPP server has responded that a SNPP command has been successfully received and processed. This response can vary among servers, but is generally prefixed by a success code in the range 250-259.

The string may contain wildcards in the form of question marks to indicate that any character (usually a digit) is acceptable at that point.

TApdPagerLog Component

TApdPagerLog is a small class that can be associated with a TApdAbstractPager descendant (e.g., TApdTAPPager or TApdSNPPPager) to provide automatic page logging services. Simply create an instance TApdPagerLog and assign it to the PagerLog property of the pager component.

TApdPagerLog creates or appends to a text file whose name is given by the PageHistoryName property. As the pager component processes a page transaction it instructs the TApdPagerLog instance to open the log file, write a new line indicating the current status, and close the file.

Following is a sample of the text that might be created by a TApdPagerLog attached to a TApdSNPPPager:

```
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

09/09/1999 18:06:52  SNPP page to 1269694 at snpp.pageinc.com:9797
                      Started
09/09/1999 18:07:01  SNPP page to 1269694 at snpp.pageinc.com:9797
                      Failed - Reason: Cancel Requested
```

Hierarchy

TComponent (VCL)	
❶ TApdBaseComponent (OOMisc)	8
TApdPagerLog (AdPager)	

Properties

HistoryName	Pager	❶ Version
-------------	-------	-----------

Methods

UpdateLog

Reference Section

HistoryName **property**

`property HistoryName : string`

Default: `adpgDefPagerHistoryName` ("APROPAGR.HIS")

↳ Determines the name of the file used to store the protocol log.

You should generally set the value of `HistoryName` before calling the pager component's `Send` method. However, because the log file is opened and closed for each update, you can change the `HistoryName` at any time you wish. If you set `HistoryName` to an empty string, automatic logging is disabled until you assign a non-empty string.

Pager **property**

`property Pager : TApdAbstractPager`

↳ The pager component that is using the log component.

`Pager` is automatically initialized when the `PagerLog` property of the owning pager component is set. You can change `Pager` to assign the log component to a different pager component.

UpdateLog **virtual method**

`procedure UpdateLog(LogStr : string); virtual;`

↳ Called for each page logging event.

You may call this method with your own strings to add items to the log at any time.

TApdGSMPhone Component

The TApdGSMPhone component provides access to cellular phones and other GSM compliant devices. One of the more popular uses of GSM is the sending and receiving of SMS messages. The TApdGSMPhone component can send SMS messages through the GSM device, provide notification when SMS messages are received, and provide access to the GSM device's internal message store.

Async Professional and SMS messaging

The TApdGSMPhone component encroaches upon a technology that is often misunderstood. GSM is the Global System for Mobile communications, a consortium of several leading cellular companies. GSM defines the communication protocol between the cell phone and the cellular service provider. GSM also defines a communications protocol between a terminal device (PC) and the cell phone. The TApdGSMPhone uses the text-mode AT command set defined by GSM Technical Specification 07.05 version 5.1.0, dated December 1996.

One of the features that make GSM popular is the text messaging that it provides. When people think of GSM text messaging they think of SMS. SMS stands for "Short Message Service", and defines a message format that can be transmitted and received by GSM compliant cell phones. In Europe and Asia, SMS is very popular. The idea behind SMS is gaining popularity in the US, but most US cellular service providers take some liberties with their definition of SMS. Most US cellular service providers advertise SMS, but they do not use the SMS defined by GSM.

There are several cellular protocols being used around the world. GSM is used primarily in Europe and Asia; TDMA and CDMA are used in the US. TDMA and CDMA do not define the protocol between the PC and the cell phone, and cell phones using these protocols use different AT command sets.

The TApdGSMPhone component requires a GSM capable device that support text-mode AT commands. This usually means a GSM cell phone. At the time of this writing, very few US cellular service providers offered GSM cell phones. The TApdGSMPhone was tested in-house with a Nokia 5190. The TApdGSMPhone component uses only the commands specified as mandatory in the GSM Technical Specification for text-mode operation. 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.

GSM—more than paging

The `TApdGSMPhone` component supports sending and receiving SMS messages through the GSM device. It also supports managing the message store maintained by the device. The `TApdGSMPhone` component can retrieve all of the messages in the GSM device's memory, delete messages, add messages, and otherwise manage the message store.

The `TApdSMSMessage` class

When the `TApdGSMPhone.Connect` method is called, the cell phone is initialized to support text-mode, and the cell phone's message store is loaded into the `MessageStore` property of the `TApdGSMPhone`. `MessageStore` is a `TStringList` descendent, each `Items` and `Objects` value represents a message in the phone's message store. The `Items` property contains a string indicating the timestamp for the message. The `Objects` property contains a `TApdSMSMessage` object, which contains properties that define the message itself. The `TApdSMSMessage` class contains the properties shown in Table 16.4 that describe the message.

Table 16.4: *TApdSMSMessage* class properties

Property	Type	Description
Address	String	The SMS address of the sender for inbound messages, of the destination for outbound messages.
Message	String	The text of the SMS message.
Status	TApdSMSStatus	The status of the message (read, sent, etc.).
TimeStamp	TDateTime	The timestamp on the message.

The `TApdSMSStatus` type is an enumeration of the possible status flags for an SMS message. The `Status` property of the `TApdSMSMessage` class can be one of the following values shown in Table 16.5.

Table 16.5: *TApdSMSStatus* enumeration

Value	Description
ssUnread	The message has not been read.
ssRead	The message has been read.
ssUnsent	The message has not been sent.
ssSent	The message has been sent.

The TApdMessageStore class

The cell phone maintains a list of messages in a message store. The TApdGSMPhone component provides that message store in the MessageStore property. When the Synchronize method of the TApdGSMPhone component is called, commands are sent to the phone to retrieve all messages in the phone's message store. These messages are stored in the MessageStore property as TApdSMSMessage objects. When messages are added to the MessageStore property they are added to the phone's message store. When messages are deleted from the MessageStore property they are deleted from the phone's message store.

When adding messages to the MessageStore property, the message is not sent automatically, it is merely placed in the phone's message store outbox. The TApdGSMPhone's SendAllMessages and SendFromMemory methods will transmit the messages once they are placed in the phone's message store. SendAllMessages will send commands to the phone to send all messages in the phone's message store outbox. SendFromMemory will only transmit the specified message.

The TApdMessageStore class is a TStringList descendent. Most public methods are overridden to manage the phone's message store as items are added, deleted or moved. The Add method will add a message to the phone's message store outbox. The Delete method will erase the specified message from the phone's message store. The Capacity property is a reflection of the phone's message store capacity. The Clear method will erase all messages in the phone's message store.

Protocol implementation

The GSM Technical Specification (GTS) 07.05 version 5.1.0 dated December 1996 defines three interface protocols to communicate between the GSM device and computer serial port:

- Binary protocol (Block mode).
- Character-based interface using "AT" commands (Text mode).
- Character-based interface with hex-encoded binary transfer of message blocks (PDU mode).

The TApdGSMPhone component implements the Text-mode interface. In text mode, the TApdGSMPhone will transmit GSM AT commands to the GSM device and collect the responses. The commands that the TApdGSMPhone uses are defined in the GTS as mandatory for the device to support. In reality, not all GSM devices support all of the

mandatory commands. If your device does not support these mandatory commands, it is unlikely that you will be able to use all of the `TApdGSMPhone` methods. The GSM AT commands that the `TApdGSMPhone` uses are listed in Table 16.6.

Table 16.6: *Supported GSM AT commands*

GSM AT Command	Description
General configuration commands	
+CSMS	Select Message Service - used to verify GSM compliance.
+CPMS	Selects Preferred Message Storage.
Message configuration commands	
+CSMP	Set text mode parameters.
+CSDH	Show text mode parameters.
Message receiving and reading commands	
+CNMI	Request New Message Indications (notification when a message is received).
+CMGL	List messages.
Message sending and writing commands	
+CMGS	Send message.
+CMSS	Send message from memory.
+CMGW	Write message to memory.
+CMGD	Delete message from memory.

The `QuickConnect` property determines whether or not the `TApdGSMPhone` component will automatically synchronize the `MessageStore` with the phone's message store upon connection. If `QuickConnect` is `False` (the default) the `MessageStore` will be synchronized.

Using the TApdGSMPhone component

The TApdGSMPhone component is designed to be functional and easy to use. The first example will demonstrate sending a single SMS message through a GSM compliant cell phone. The second example will demonstrate one way to display a phone's message store in a TListBox component.

Sending a single SMS message

Create a new project; drop a TApdComPort and TApdGSMPhone component onto the form. Set the ComNumber property of the TApdComPort component to the serial port number where the GSM data cable is connected. Drop a TLabel component onto the form and change the Caption property to "Destination address." Drop a TEdit component on the form next to the label. Drop another TLabel component on the form and change the Caption to "Message." Drop another TEdit component onto the form next to that label. Finally, drop the obligatory TButton onto the form and change the Caption property to "Send." Create the button's OnClick event handler and enter the following code:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    { set the message properties }
    ApdGSMPhone1.SMSAddress := Edit1.Text;
    ApdGSMPhone1.SMSMessage := Edit2.Text;
    { tell the component not to synchronize the message store }
    ApdGSMPhone1.QuickConnect := True;
    { 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: ' + ErrorMessage(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.

To expand upon this example, you can send multiple messages in the same session. To do this, add the `OnNextMessage` event handler. This event is generated when the response to the previous message has been received. If another message is ready to be sent, set the `Address` and `Message` properties to the appropriate values for the new message and set the `NextMessageReady` parameter to `True`. If another message is not ready to be sent, set the `NextMessageReady` parameter to `False`. Once all messages have been sent (after you set `NextMessageReady` to `False`), the `OnSessionFinish` event will be generated.

Displaying the phone's message store

This example demonstrates how to connect to your GSM compliant cell phone, and retrieve and display the phone's message store. Create a new project and drop a `TApdComPort` and `TApdGSMPhone` component onto the form. Set the `ComNumber` property of the `TApdComPort` component to the serial port number where the GSM data cable is connected. Drop three `TEdit` components onto the form, change the names to `edtAddress`, `edtTimestamp`, and `edtStatus`. Drop a `TMemo` on the form to contain the message and name that `MemoMessage`. Finally, drop the obligatory `TButton` onto the form and change the `Caption` property to "Connect." Create the button's `OnClick` event handler and enter the following code:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    { connect to phone and synchronize the message store }
    ApdGSMPhone1.QuickConnect := False;
    ApdGSMPhone1.Connect;
end;
```

If the `QuickConnect` property is `False`, the `MessageStore` property of the `TApdGSMPhone` component will be populated with the cell phone's internal message store. When the `TApdGSMPhone` component completes the initialization and retrieves the message store from the phone, the `OnGSMComplete` event will be generated. The `State` parameter of that event handler will be `gsListAll`, and the `ErrorCode` will be `ecOK` if the component was able to retrieve the message store. Create the `OnGSMComplete` event handler and add the following code to display the timestamps 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. Add the OnDbClick event handler for the TListBox and add the following code:

```
procedure TForm1.ListBox1DbClick(Sender: TObject);
var
  I : Integer;
  Msg : TApdSMSMessage;
begin
  if ListBox1.ItemIndex > 0 then begin
    I := ListBox1.ItemIndex;
    Msg := ApdGSMPHONE1.MessageStore.Messages[I];
    edtAddress.Text := Msg.Address;
    edtTimestamp.Text := Msg.TimeStampStr;
    edtStatus.Text := ApdGSMPHONE1.StatusToStr(Msg.Status);
    MemoMessage.Text := Msg.Message;
  end;
end;
```

TApdSMSMessage Class

The TApdGSMPhone component uses the TApdSMSMessage class to defines SMS messages contained in the MessageStore of the component. The TApdSMSMessage class descends from TObject and encapsulates an SMS message.

Hierarchy

TObject (VCL)
 TApdSMSMessage (AdGSM)

Properties

Address	MessageIndex	TimeStamp
Message	Status	TimeStampStr

Reference Section

Address

property

```
property Address : string
```

↪ Contains the SMS address of the message.

Address is the SMS address of the SMS message. For received messages, Address is the address of the sender. For outbound message, Address is the address of the destination.

Message

property

```
property Message : string
```

↪ The SMS message text.

Message is the text of the SMS message. SMS text messages are usually limited to 160 characters, including the address and the text of the message. No validation is performed on the size of the message through the components.

MessageIndex

property

```
property MessageIndex : Integer
```

↪ The position in the phone's message store.

The MessageIndex property reflects the position of the message in the phone's message store. The MessageIndex is dependent on the phone's message store, and may not be consecutive.

Status

property

```
property Status : TApdSMSStatus
```

```
TApdSMSStatus = (ssUnread, ssRead, ssUnsent, ssSent, ssAll);
```

↳ Status determines the status of the message.

Status is the phone's flag for the message. Status can be one of the following:

Value	Description
ssUnread	The message has not been read.
ssRead	The message has been read.
ssUnsent	The message has not been sent.
ssSent	The message has been sent.

The `StatusToString` method of the `TApdGSMPhone` component can convert a `TApdSMSStatus` value to a string.

TimeStamp

property

```
property TimeStamp : TDateTime
```

↳ TimeStamp is a `TDateTime` indicating the timestamp of the message.

TimeStamp is the timestamp for the message. The SMS timestamp is usually inserted into the message by the cell phone, and usually indicates when the message was transmitted. You cell phone may treat TimeStamp differently.

See also: `TimeStampStr`

TimeStampStr

property

```
property TimeStampStr : string
```

↳ A string containing the message timestamp.

The GSM specification defines a fairly specific format to indicate the timestamp of the message. The `TimeStampStr` property reflects the literal timestamp provided by the cell phone. The `TimeStamp` property is a converted representation of this literal timestamp string.

See also: `TimeStamp`

TApdMessageStore Class

The TApdGSMPhone maintains a list of messages which reflects the message store on the cell phone. The TApdMessageStore class defines the interface with the cell phone's message store.

Hierarchy

TStringList (VCL)

 TApdMessageStore (AdGSM)

Properties

Capacity

Messages

Methods

AddMessage

Clear

Delete

Reference Section

AddMessage

method

```
function AddMessage(const Dest, Msg : string) : Integer;
```

↳ Adds an SMS message to the message store.

Use the AddMessage method to add an SMS message to the message store. If the TApdGSMPhone is connected to the cell phone, the message is added immediately.

Dest is the destination address of the message. Msg is the text of the message.

Capacity

read-only, run-time property

```
property Capacity : Integer
```

↳ Indicates the message store capacity.

Different cell phones will have internal message stores with different capacities. The Capacity property indicates the number of messages that the phone's message store can contain. This property is valid once the Synchronize method of the TApdGSMPhone has been called.

Clear

method

```
procedure Clear;
```

↳ Deletes all messages from the message store.

Use the Clear method to delete all messages from the TApdMessageStore and the cell phone.

Delete

method

```
procedure Delete(Index: Integer);
```

↳ Deletes a single message from the TApdMessageStore and the cell phone.

Delete is an indexed property that deletes the specified message from both the cell phone's message store and the TApdMessageStore. Index is the index of the message to delete.

```
property Messages[Index: Integer] : TApdSMSMessage
```

↳ Contains the messages stored in the TApdMessageStore.

The Messages property of the TApdMessageStore class is an indexed property that provides access to the SMS messages contained in the class. Use Messages the same way the Strings and Objects properties of the TStringList are used. The return value will be the TApdSMSMessage stored in the location determined by Index.

TApdGSMPhone Component

TApdGSMPhone forms the foundation for accessing cell phones or other GSM devices and provides a set of properties and methods to control TApdSMSMessage class and TApdMessageStore.

Hierarchy

TComponent (VCL)

- ❶ TApdBaseComponent (OOMISC) 8
 - TApdCustomGSMPhone (AdGSM)
 - TApdGSMPhone (AdGSM)

Properties

ComPort	MessageStore	SMSMessage
Connected	NotifyOnNewMessage	❶ Version
ErrorCode	QuickConnect	
GSMState	SMSAddress	

Methods

Connect	SendMessage
SendAllMessages	Synchronize

Events

OnGSMComplete	OnNewMessage	OnNextMessage
---------------	--------------	---------------

Reference Section

ComPort

property

```
property ComPort : TApdCustomComPort
```

↪ Determines the serial port used by the TApdGSMPhone component.

A comport component must be assigned to this property before connecting to the phone with the AutoOpen set to True (default). If the AutoOpen property of ComPort is False, the port must be explicitly opened before use.

Connect

method

```
procedure Connect;
```

↪ Connects to the cell phone or GSM device.

The Connect method configures the device, verifies that it supports the GSM AT command set, and synchronizes the message store. When QuickConnect is False, the TApdGSMPhone component will be synchronized automatically with the phone's message store. When QuickConnect is True, the Synchronize method will not be called and one can send a message without using the memory store.

Once Connect completes, the OnGSMComplete event will be generated. If QuickConnect is False, the State parameter of that event will be gsListAll. If QuickConnect is True, the State parameter of that event will be gsConfig.

See also: Connected, OnGSMComplete, QuickConnect

Connected

property

```
property Connected : Boolean
```

↪ Determines whether a connection to the phone has been made or not.

Connected is True when a successful link to the device has been made, and False when not linked to the device.

See also: Connect

property ErrorCode : Integer

↳ The result of the last operation

ErrorCode contains the result of the last operation. ErrorCode can be one of the following values:

Error Code	Description
0...127	GSM 04.11 values.
128...255	GSM 03.40 values.
300	ME failure.
301	SMS service of ME reserved.
302	Operation not allowed.
303	Operation not supported.
304	Invalid PDU mode parameter.
305	Invalid text mode parameter.
310	SIM not inserted.
311	SIM PIN required.
312	PH-SIM PIN required.
313	SIM failure.
314	SIM busy.
315	SIM wrong.
316	SIM PUK required.
317	SIM PIN2 required.
318	SIM PUK2 required.
320	Memory failure.

Error Code	Description
321	Invalid memory index.
322	Memory full.
330	SMSC address unknown.
331	No network service.
332	Network timeout.
340	No +CNMA acknowledgement expected.
500	Unknown error.
512...	Manufacturer specific.

GSMState **property**

property GSMState : TApdGSMStates

```
TApdGSMStates = (
    gsNone, gsConfig, gsSendAll, gsListAll, gsSend, gsWrite)
```

↳ The state the GSM Phone is in at that moment.

The GSMStates are used internally primarily to determine that state of the GSM state machine. GSMState will be set when one of the TApdGSMPhone public methods is called to indicate what the TApdGSMPhone is doing.

The following table shows the TApdGSMStates values:

Value	Description
gsNone	Idle.
gsConfig	Configuration and GSM validation.
gsSendAll	Sending all messages from the message store.
gsListAll	Retrieving messages from the message store.
gsSend	Sending a single SMS message.
gsWrite	Writing a single SMS message to the message store.

```
property MessageStore : TApdMessageStore
```

↳ Contains a list of SMS messages.

The MessageStore property contains a list of SMS messages. The MessageStore property is a TApdMessageStore, which is a TStringList descendent. See the TApdMessageStore and TApdSMSMessage definitions earlier in this section for details.

NotifyOnNewMessage

```
property NotifyOnNewMessage : Boolean
```

↳ Determines whether notification is provided for new messages.

The GSM specification permits initializing the device to provide notification when new messages are received. The NotifyOnNewMessage property determines whether the device is configured to provide this notification or not.

When a new message is received, and the device has been properly configured, the OnNewMessage event will be generated to provide notification of the newly received message.

See also: OnNewMessage

OnGSMComplete

```
property OnGSMComplete : TApdGSMCompleteEvent
```

```
TApdGSMCompleteEvent = procedure(Pager : TApdCustomGSMPhone;  
    State : TApdGSMStates; ErrorCode : Integer) of object;
```

↳ Defines an event handler that is called when the GSM operation is complete.

A GSM operation is started when the Connect, SendAllMessages, SendMessage, and Synchronize methods are called, as well as when the messages contained in the MessageStore property are modified. The OnGSMComplete event is generated when the operation is complete.

Pager is the TApdCustomGSMPhone component that generated the event. State is a TApdGSMState that indicates the operation that just completed. ErrorCode is the result of the operation.

See also: ErrorCode, GSMState

```
property OnNewMessage : TApdGSMNewMessageEvent
```

```
TApdGSMNewMessageEvent = procedure(Pager : TApdCustomGSMPhone;  
    Index : Integer; Message : string) of object;
```

- ↳ Defines an event handler that is called when there is a new SMS message in the memory store.

If the `NotifyOnNewMessage` property is `True`, and the device supports it, the device will send notification to the `TApdGSMPhone` component when a new message is received. When this notification is received, the newly received message is stored in the `MessageStore` property and this event is generated.

`Pager` is the `TApdCustomGSMPhone` component that generated the event. `Index` is the position in the `MessageStore` property where the new message was placed. `Message` is a string containing the text of the received message.

See also: `MessageStore`, `NotifyOnNewMessage`

OnNextMessage

```
property OnNextMessage : TApdGSMNextMessageEvent
```

```
TApdGSMNextMessageEvent = procedure(Pager : TApdCustomGSMPhone;  
    Index : Integer; NextMessageReady : Boolean) of object;
```

- ↳ Defines an event handler that returns the next message to send.

The `TApdGSMPhone` component supports sending multiple SMS messages in a single operation. When the `SendMessage` method is called, the message defined by the `SMSAddress` and `SMSMessage` properties is sent immediately. Once confirmation that this message has been sent is received, the `OnNextMessage` event is generated.

If another message is ready to be sent, change the `SMSAddress` and `SMSMessage` properties to reflect the new message to send, and set the `NextMessageReady` parameter to `True`. If another message is not ready, set the `NextMessageReady` parameter to `False`. The `OnGSMComplete` event handler will be generated once `NextMessageReady` is set to `False`.

`ErrorCode` contains the result of the last message sent.

See also: `OnGSMComplete`, `SendMessage`, `SMSAddress`, `SMSMessage`

QuickConnect

property

```
property QuickConnect : Boolean
```

Default: False

✚ This property determines whether to connect without synchronizing the phone.

When the Connect method of the TAPdGSMPhone component is called, the device is initialized and the GSM capabilities of the device are verified. If the QuickConnect property is False (the default), the MessageStore property is synchronized with the message store of the device. If QuickConnect is True, the operation terminates once the device has been connected.

Set QuickConnect to True if your device does not support an internal message store, or if you do not want to synchronize the message stores.

See also: Connect

SendAllMessages

method

```
procedure SendAllMessages;
```

✚ This procedure sends all the SMS messages in the memory store.

SendAllMessages iterates through all of the messages contained in the MessageStore property and sends all messages with a Status of ssUnsent.

SendMessage

method

```
procedure SendMessage;
```

✚ This routine will send a message without placing the message in memory.

This method is used to send a single message, or to start sending a series of messages, without accessing the GSM device's message store. Set the SMSAddress property to the SMS address of the recipient for the message, and the SMSMessage property to the message that you want to send, then call the SendMessage method. If the OnNextMessage event handler is assigned, that event will be generated once SendMessage sends the message. The OnNextMessage event handler can be used to send a sequence of messages.

The OnGSMComplete event handler will be generated once SendMessage completes sending messages.

See also: OnGSMComplete, OnNextMessage, SMSAddress, SMSMessage

SMSAddress

property

```
property SMSAddress : string
```

↳ The SMS address of the recipient for an SMS message.

The SMSAddress property determines where the SMS message will be sent. This property is used with the SendMessage method to transmit a single message, or a series of messages. The message determined by SMSMessage is sent to the address determined by SMSAddress.

The SMSAddress and SMSMessage properties, and the SendMessage method, do not access the message store of the GSM device or the MessageStore property of the component.

See also: OnNextMessage, SendMessage, SMSMessage

SMSMessage

property

```
property SMSMessage : string
```

↳ The SMS message.

The SMSMessage property determines the message to send. This property is used with the SendMessage method to transmit a single message, or a series of messages. The message determined by SMSMessage is sent to the address determined by SMSAddress.

The SMSAddress and SMSMessage properties, and the SendMessage method, do not access the message store of the GSM device or the MessageStore property of the component.

See also: OnNextMessage, SendMessage, SMSAddress

Synchronize

method

```
procedure Synchronize;
```

↳ This routine will synchronize the message store of the GSM device.

The Synchronize method is used to synchronize the messages contained in the MessageStore property with the messages contained in the GSM device's internal message store. This method is called internally during a Connect sequence if the QuickConnect property is False.

See also: Connect, MessageStore, QuickConnect

Chapter 17: Low-level Facilities

The routines described in this chapter come from a few units that Async Professional uses internally. These routines may also prove useful in your applications and therefore are documented here.

The first section documents procedures and functions that manage event timers. These non-object-oriented routines are used to provide tick-resolution (18.2 ticks/second) timing services for the rest of the library.

The second section documents a few functions that return strings for numeric codes: serial port names, error messages, and protocol names from the corresponding numeric type used by Async Professional.

The third section documents the `IsPortAvailable` method, which can be used to determine whether a specific serial port is available.

Timers

The OoMisc unit provides the timer routines used internally by Async Professional. You might find these routines handy for your programs as well. In almost all cases you will find it more convenient to use timer triggers with a TApdComPort component (see page 22) rather than working with the timer routines directly. If you don't want to use OoMisc timers directly then you don't need to read this section.

OoMisc's basic time unit is the BIOS clock tick. One clock tick is approximately 55 milliseconds. Put another way, there are about 18.2 clock ticks per second. This means that 55 milliseconds is the smallest interval that you can time and the timing of any event has an uncertainty of 55 milliseconds.

Unless otherwise specified, all Async Professional routines that have time-out parameters—the Ticks parameter to the TApdComPort SetTimerTrigger method, for example—require a value expressed in clock ticks. OoMisc also offers routines to convert between ticks and seconds in case you prefer to work with seconds.

The basic timer record is called an EventTimer. When an EventTimer is initialized it is passed the duration of the event. EventTimers can also be used to measure elapsed times, in which case the initial duration isn't important and can be set to zero.

Here is an example program that uses some timer functions:

```
uses OoMisc;

procedure TForm1.Button1Click(Sender: TObject);
var
    ET: EventTimer;
begin
    Aborted := False;           { Global Boolean Flag }
    Label2.Caption := '';
    NewTimer(ET, Secs2Ticks(60));
    repeat
        Label1.Caption := Format(
            'Elapsed ticks: %d   Remaining ticks: %d',
            [ElapsedTime(ET), RemainingTime(ET)]);
        Application.ProcessMessages;
    until Abort or TimerExpired(ET);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    Aborted := True;
    Label2.Caption := 'Aborted';
end;
```

This example uses a simple WinCrt-based interface because the EventTimer is user-interface independent.

An EventTimer is initialized by calling NewTimer, which is passed an EventTimer and the number of ticks until expiration. This example uses the Secs2Ticks function to start an EventTimer that expires in 60 seconds (or 1092 clock ticks).

This program loops continuously, displaying the elapsed ticks and the remaining ticks, until the timer expires or a key is pressed.

It is perfectly legal to use the ElapsedTime routines even after a timer has expired. If the example program called ElapsedTime after the timer had expired, it would still return the number of ticks since the timer was started.

A timer is good for 24 hours at most. If you reference a timer after 24 hours, the results are modulo 24 hours.

Routines

DelayTicks	NewTimerSecs	Ticks2Secs
ElapsedTime	RemainingTime	TimerExpired
ElapsedTimeInSecs	RemainingTimeInSecs	
NewTimer	Secs2Ticks	

Reference Section

DelayTicks

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

↳ Delays for a specified number of clock ticks.

If Yield is False, DelayTicks does not yield to other applications and returns control only after Ticks ticks elapse. In this case the function result is always zero.

If Yield is True, DelayTicks yields to other applications and to the owning application. The return value is a long integer whose low word is the last Windows message number and high word is the window handle that received the message. If the owning application posts a quit message (WM_QUIT) DelayTicks reposts the message and exits, and the message part of the function result is WM_QUIT.

Even with Yield set to True, DelayTicks should not be used to delay for periods of time longer than a few seconds. For longer delays you should set up an EventTimer or a timer trigger and continue the execution of your program until the timer expires.

ElapsedTime

```
function ElapsedTime(ET : EventTimer) : LongInt;
```

```
EventTimer = record  
  StartTicks : LongInt; {Tick count when timer was initialized}  
  ExpireTicks : LongInt; {Tick count when timer will expire}  
end;
```

↳ Returns the elapsed time, in ticks, for this timer.

This routine returns the number of ticks that have elapsed since NewTimer was called to initialize the specified EventTimer. In the EventTimer record, both fields hold tick counts since midnight. A tick is a PC hardware interval that occurs roughly 18.2 times per second, or about once per 55 milliseconds.

See also: ElapsedTimeInSecs, NewTimer

ElapsedTimeInSecs

```
function ElapsedTimeInSecs(ET : EventTimer) : LongInt;
```

↳ Returns the elapsed time, in seconds, for this timer.

This routine returns the same result as `ElapsedTime`, but converted to seconds. Partial seconds are truncated.

The following example shows how to use an `EventTimer` within a simple WinCrt application. `Async Professional` uses timers internally, without direct connection to the user interface.

This example displays the elapsed time until a key is pressed or 20 seconds have elapsed:

```
var
    ET : EventTimer;
begin
    WriteLn('You have 20 seconds to press a key:');
    NewTimerSecs(ET, 20);
    repeat
        Write(^M, ElapsedTimeInSecs(ET));
    until KeyPressed or TimerExpired(ET);
    WriteLn;
    if not KeyPressed then WriteLn('Time is up.');
```

end.

See also: `ElapsedTime`, `NewTimer`

NewTimer

```
procedure NewTimer(var ET : EventTimer; Ticks : LongInt);
```

↳ Initializes a timer that will expire in the specified number of clock ticks.

This routine initializes an `EventTimer` record, which is used to measure elapsed time or to schedule an event. `NewTimer` does two things: 1) it stores the current time in the `StartTicks` field of `ET`, and 2) it calculates what the time will be when `Ticks` number of clock ticks expire and stores that value in `ExpireTicks`. `Ticks` must be less than or equal to `TicksPerDay` (1,573,040). The timer handles rollover at midnight when needed.

The following example initializes a timer and then loops until the timer expires in 20 seconds.

```
var
    ET : EventTimer;
...
NewTimer(ET, Secs2Ticks(20));
repeat
    ...
until TimerExpired(ET);
```

See also: [ElapsedTime](#), [NewTimerSecs](#), [RemainingTime](#), [TimerExpired](#)

NewTimerSecs

```
procedure NewTimerSecs(var ET : EventTimer; Secs : LongInt);
```

↳ Initializes a timer that will expire in the specified number of seconds.

This routine is identical to [NewTimer](#) except that the time-out period is expressed in terms of seconds rather than clock ticks.

See also: [ElapsedTime](#), [NewTimer](#), [RemainingTimeInSecs](#), [TimerExpired](#)

RemainingTime

```
function RemainingTime(ET : EventTimer) : LongInt;
```

↳ Returns the amount of time remaining, in clock ticks, until the specified timer expires.

If the timer has already expired, [RemainingTime](#) returns zero.

See also: [ElapsedTime](#), [RemainingTimeInSecs](#), [TimerExpired](#)

RemainingTimeInSecs

```
function RemainingTimeInSecs(ET : EventTimer) : LongInt;
```

↳ Returns the remaining time, in seconds, for the specified timer.

Partial seconds are truncated. If the timer has already expired, [RemainingTimeInSecs](#) returns zero.

See also: [ElapsedTime](#), [RemainingTime](#), [TimerExpired](#)

Secs2Ticks

```
function Secs2Ticks(Secs : LongInt) : LongInt;
```

↳ Converts seconds to clock ticks.

The conversion uses long integer arithmetic to do the conversion, which does not throw away any accuracy given that an integer number of ticks is being returned.

See also: Ticks2Secs

Ticks2Secs

```
function Ticks2Secs(Ticks : LongInt) : LongInt;
```

↳ Converts clock ticks to seconds.

This routine uses long integer arithmetic and rounds to the nearest second.

The following example gets the elapsed time in ticks from a timer and displays it to the nearest number of seconds.

```
var
    ET : EventTimer;
...
WriteLn(Ticks2Secs(ElapsedTime(ET)));
```

See also: Secs2Ticks

TimerExpired

```
function TimerExpired(ET : EventTimer) : Bool;
```

↳ Returns True if the specified timer has expired.

The timer expires when the time originally passed to NewTimer or NewTimerSecs has elapsed. A timer's elapsed time can still be used even after the timer expires.

See also: ElapsedTime, NewTimer, NewTimerSecs

Name Routines

Async Professional provides several routines that are useful in message boxes, status dialogs, and logging reports. These routines simply convert a numeric value such as an error code into a string that describes the meaning of the number. These routines are described in this section.

Routines

ComName

ErrorMsg

ProtocolName

Reference Section

ComName

```
function ComName(const ComNumber : Word) : string;
```

↳ Returns the name of a serial port.

ComName simply appends ComNumber to “COM.” For example, if ComNumber is 3, ComName returns “COM3.”

ComName is interfaced by the AdPort unit.

ErrorMsg

```
function ErrorMsg(const ErrorCode : SmallInt) : string;
```

↳ Returns an English string describing an error code.

These strings are stored in a string table in APW.RES, which is linked into the application. The string table can be translated into another language if desired.

The error code you pass to ErrorMsg is usually obtained from the ErrorCode property of any exception class derived from EAPDException. All Async Professional exceptions initialize this property. In fact, when the EAPDException Create constructor is called, it passes the string returned by ErrorMsg on to the Create constructor of the VCL Exception class.

The error code can also be obtained from the TApdProtocol component's ProtocolError property when a protocol is terminated abnormally, or from the ErrorCode parameter passed to the OnProtocolError event handler. Because protocols run in the background, they do not generate exceptions but instead pass error codes.

A complete list of error codes, exceptions, and error messages is given in “Error Handling and Exception Classes” on page 900.

ErrorMsg is interfaced by the AdExcept unit.

The following example shows an exception handler that takes specific action for some error codes and shows an error message for all others:

```
try
  ApdComPort.Open := True;
  ...work with comport component
except
  on E : EAPDException do
    case E.ErrorCode of
      ...handle specific errors
    else
      {show error message for all errors not handled above}
      ShowMessage(ErrorMessage(E.ErrorCode));
    end;
  end;
```

IsPortAvailable

```
function IsPortAvailable(ComNum : Cardinal) : Boolean;
```

↳ Determines whether a serial port is valid or not.

The `IsPortAvailable` method will return true if a given serial port is valid, or false if the serial port is not valid. The serial port to verify is passed in the `ComNum` parameter.

Serial port validity is determined by two typed constants: `ApdShowPortsInUse` and `ApdUseDispatcherForAvail`.

`ApdShowPortsInUse` defaults to true. When `ApdShowPortsInUse` is true, `IsPortAvailable` will consider a serial port that is in use by another application to be a valid port and return true. When `ApdShowPortsInUse` is false, `IsPortAvailable` will consider a serial port that is in use by another application to be an invalid port and return false.

`ApdUseDispatcherForAvail` also defaults to true. This typed constant determines whether the validity check is made by the Async Professional dispatcher, or by the Win32 `CreateFile` API method. Some serial ports are not accessible directly through our dispatcher, such as pure virtual serial port emulations. While these ports are not accessible through the default dispatcher, they may be accessible through a custom dispatcher or through the TAPI interface.

The serial device selection dialog, displayed when the `TApdComPort.ComNumber = 0` and `TApdComPort.PromptForPort = True`, will display all available serial ports using the `IsPortAvailable` method. Changing the `ApdShowPortsInUse` and `ApdUseDispatcherForAvail` values will filter the available port drop down list according to the new values.

The following example will populate a TListBox with a list of all ports that are available and not in use. ComName is interfaced in the AdPort unit.

```
uses
    AdSelCom, AdPort;
procedure TForm1.Button1Click(Sender: TObject);
var
    I : Integer;
begin
    ApdShowPortsInUse := False;
    for I := 1 to 50 do
        if IsPortAvailable(I) then
            ListBox1.Items.Add(ComName(I) + ' is available');
    end;
```

ProtocolName

```
function ProtocolName(const ProtocolType : TProtocolType) : string;
TProtocolType = (
    ptNoProtocol, ptXmodem, ptXmodemCRC, ptXmodem1K, ptXmodem1KG,
    ptYmodem, ptYmodemG, ptZmodem, ptKermit, ptAscii, ptBPlus);
```

↪ Returns the name of a protocol.

The ProtocolType property of a TApdProtocol component can be passed to this function to return the name of the protocol. For example, passing ptXmodem to this function returns “Xmodem” and passing ptBPlus returns “B+”.

ProtocolName is interfaced by the AdProtcl unit.

See also: TApdProtocol.ProtocolType

Chapter 18: Appendices

This chapter contains a discussion of error handling, a description of the Async Professional conditional defines, a glossary of communications terms, and some general tips on debugging asynchronous communications programs in the Windows environment.

Error Handling and Exception Classes

Async Professional takes a consistent approach to error handling throughout the library. It uses VCL exceptions to report errors wherever it makes sense. Each exception corresponds to a particular error condition, which also corresponds to a numeric error code.

In two particular cases, exceptions are not a sensible way of reporting errors. The first case is for serial line errors. These are usually caused by line noise which can occur randomly throughout a communications session. They can either be ignored or they can cause the application to request that the remote resend a block of data. The second case is for errors that occur during protocol file transfers. Most such errors are handled automatically by the protocol. The few that cause the protocol to terminate cannot be reported by exceptions because Async Professional protocols run in the background, and there is no telling what the foreground application is doing when the fatal error occurs.

In both of these cases, errors are reported by using status codes that the application can check. Line errors are reported using a unique set of status codes that are described in the reference entry for the LineError property of the TApdComPort component (see page 22). Protocol errors are reported using status codes that fit into the same numeric system used for errors that are reported as exceptions. If a protocol error occurs in the foreground (when a property is set to an invalid value for example), an exception is generated.

All Async Professional exceptions descend from a class named EAPDException, itself descended from the VCL Exception class. EAPDException has a read/write, run-time property named ErrorCode. This property returns an integer status code that can be tested to determine the cause for a particular exception. In special cases, a new value can be assigned to ErrorCode to update the meaning of an error or to cause other error handlers to disregard the error.

Further descended from EAPDException are eight exception classes that encompass error groups as shown in Table 18.1.

Table 18.1: *Exception class descendants of EAPDException*

Exception	Group Description
EGeneral	General programmer error.
EOpenComm	Error occurred while opening a serial port.
ESerialIO	Call to a communications API failed.
EModem	Improper use of a modem or dialer component.
ETrigger	Error while adding or modifying a trigger.
EPacket	Improper use of data packet component.

Table 18.1: *Exception class descendants of EAPDException (continued)*

Exception	Group Description
EProtocol	Improper use of a protocol component.
EINI	INI database error.
EFax	Fax conversion, unpacking, sending, or receiving error.
ETapi	TAPI dial, answer, or configuration error.

Additional exception classes derived from these groups correspond to particular error conditions and numeric error codes. The Table 18.2 shows the most specific Async Professional exception classes in alphabetical order.

Table 18.2: *Additional Async Professional exception classes*

Exception	Ancestor	Error Code(s)
EAlreadyDialing	EModem	ecAlreadyDialing
EAlreadyOpen	EOpenComm	ecAlreadyOpen
EApdSocketException	Exception	All ADWSXXX and WSAXXX error codes
EBadArgument	EGeneral	ecBadArgument
EBadFieldForIndex	EINI	ecBadFieldForIndex
EBadFieldList	EINI	ecBadFieldList
EBadGraphicsFormat	EFax	ecBadGraphicsFormat
EBadId	EOpenComm	ecBadId
EBadTriggerHandle	ETrigger	ecBadTriggerHandle
EBaudRate	EOpenComm	ecBaudRate
EBufferIsEmpty	ESerialIO	ecBufferIsEmpty
EBufferTooBig	EGeneral	ecBufferTooBig
EByteSize	EOpenComm	ecByteSize
ECannotUseWithWinsock	EGeneral	ecCannotUseWithWinsock
ECantMakeBitmap	EFax	ecCantMakeBitmap
ECommNotOpen	EOpenComm	ecCommNotOpen
EConvertAbort	EFax	ecConvertAbort
EDatabaseEmpty	EINI	ecDatabaseEmpty

Table 18.2: *Additional Async Professional exception classes (continued)*

Exception	Ancestor	Error Code(s)
EDatabaseFull	EINI	ecDatabaseFull
EDataTooLarge	EINI	ecDataTooLarge
EDefault	EOpenComm	ecDefault
EFaxBadFormat	EFax	ecFaxBadFormat
EFaxBadMachine	EFax	ecFaxBadMachine
EFaxBadModemResult	EFax	ecFaxBadModemResult
EFaxBusy	EFax	ecFaxBusy
EFaxDataCall	EFax	ecFaxDataCall
EFaxInitError	EFax	ecFaxInitError
EFaxNoCarrier	EFax	ecFaxNoCarrier
EFaxNoDialTone	EFax	ecFaxNoDialTone
EFaxPageError	EFax	ecFaxPageError
EFaxSessionError	EFax	ecFaxSessionError
EFaxTrainError	EFax	ecFaxTrainError
EFaxVoiceCall	EFax	ecFaxVoiceCall
EFontFileNotFound	EFax	ecFontFileNotFound
EGetBlockFail	ESerialIO	ecGetBlockFail
EGotQuitMsg	EGeneral	ecGotQuitMsg
EHardware	EOpenComm	ecHardware
EIniRead	EINI	ecIniRead
EIniWrite	EINI	ecIniWrite
EInternal	EGeneral	ecInternal, ecNoFieldsDefined, ecNoIndexKey, ecDatabaseNotPrepared
EInvalidPageNumber	EFax	ecInvalidPageNumber
EInvalidProperty	EPacket	ecStartStringEmpty, ecPacketTooSmall, ecNoEndCharCount, ecEmptyEndString, ecZeroSizePacket
EKeyTooLong	EINI	ecKeyTooLong

Table 18.2: *Additional Async Professional exception classes (continued)*

Exception	Ancestor	Error Code(s)
ELoggingNotEnabled	ESerialIO	ecLoggingNotEnabled
EMemory	EOpenComm	ecMemory
EModemBusy	EModem	ecModemBusy
EModemNotAssigned	EGeneral	ecModemNotAssigned
EModemNotDialing	EModem	ecModemNotDialing
EModemNotResponding	EModem	ecModemNotResponding
EModemNotStarted	EModem	ecModemNotStarted
EModemRejectedCommand	EModem	ecModemRejectedCommand
EModemStatusMismatch	EModem	ecModemStatusMismatch
ENoHandles	EOpenComm	ecNoHandles
ENoImageBlockMarked	EFax	ecNoImageBlockMarked
ENoImageLoaded	EFax	ecNoImageLoaded
ENoPortSelected	EOpenComm	ecNoPortSelected
ENotDialing	EModem	ecNotDialing
ENoTimers	EOpenComm	ecNoTimers
ENullApi	ESerialIO	ecNullApi
EOutputBufferTooSmall	ESerialIO	ecOutputBufferTooSmall
EPhonebookNotAssigned	EGeneral	ecPhonebookNotAssigned
EPortNotAssigned	EGeneral	ecPortNotAssigned
EPutBlockFail	ESerialIO	ecPutBlockFail
ERecordExists	EINI	ecRecordExists
ERecordNotFound	EINI	ecRecordNotFound
ERegisterHandlerFailed	ESerialIO	ecRegisterHandlerFailed
ESequenceError	EProtocol	ecSequenceError
EStringSizeError	EPacket	ecPacketTooLong
ETapi16Disabled	ETapi	ecTapi16Disabled
ETapiAddressBlocked	ETapi	ecAddressBlocked
ETapiAllocated	ETapi	ecAllocated
ETapiBadDeviceID	ETapi	ecBadDeviceID

Table 18.2: *Additional Async Professional exception classes (continued)*

Exception	Ancestor	Error Code(s)
ETapiBearerModeUnavail	ETapi	ecBearerModeUnavail
ETapiBillingRejected	ETapi	ecBillingRejected
ETapiBusy	ETapi	ecTapiBusy
ETapiCallUnavail	ETapi	ecCallUnavail
ETapiCompletionOverrun	ETapi	ecCompletionOverrun
ETapiConferenceFull	ETapi	ecConferenceFull
ETapiDialBilling	ETapi	ecDialBilling
ETapiDialDialtone	ETapi	ecDialDialtone
ETapiDialPrompt	ETapi	ecDialPrompt
ETapiDialQuiet	ETapi	ecDialQuiet
ETapiGetAddrFail	ETapi	ecTapiGetAddrFail
ETapiIncompatibleApiVersion	ETapi	ecIncompatibleApiVersion
ETapiIncompatibleExtVersion	ETapi	ecIncompatibleExtVersion
ETapiIniFileCorrupt	ETapi	ecIniFileCorrupt
ETapiInUse	ETapi	ecInUse
ETapiInvalAddress	ETapi	ecInvalAddress
ETapiInvalAddressID	ETapi	ecInvalAddressID
ETapiInvalAddressMode	ETapi	ecInvalAddressMode
ETapiInvalAddressState	ETapi	ecInvalAddressState
ETapiInvalAppHandle	ETapi	ecInvalAppHandle
ETapiInvalAppName	ETapi	ecInvalAppName
ETapiInvalBearerMode	ETapi	ecInvalBearerMode
ETapiInvalCallComplMode	ETapi	ecInvalCallComplMode
ETapiInvalCallHandle	ETapi	ecInvalCallHandle
ETapiInvalCallParams	ETapi	ecInvalCallParams
ETapiInvalCallPrivilege	ETapi	ecInvalCallPrivilege
ETapiInvalCallSelect	ETapi	ecInvalCallSelect
ETapiInvalCallState	ETapi	ecInvalCallState
ETapiInvalCallStatelist	ETapi	ecInvalCallStatelist

Table 18.2: *Additional Async Professional exception classes (continued)*

Exception	Ancestor	Error Code(s)
ETapiInvalCard	ETapi	ecInvalCard
ETapiInvalCompletionID	ETapi	ecInvalCompletionID
ETapiInvalConfCallHandle	ETapi	ecInvalConfCallHandle
ETapiInvalConsultCallHandle	ETapi	ecInvalConsultCallHandle
ETapiInvalCountryCode	ETapi	ecInvalCountryCode
ETapiInvalDeviceClass	ETapi	ecInvalDeviceClass
ETapiInvalDeviceHandle	ETapi	ecInvalDeviceHandle
ETapiInvalDialParams	ETapi	ecInvalDialParams
ETapiInvalDigitList	ETapi	ecInvalDigitList
ETapiInvalDigitMode	ETapi	ecInvalDigitMode
ETapiInvalDigits	ETapi	ecInvalDigits
ETapiInvalExtVersion	ETapi	ecInvalExtVersion
ETapiInvalFeature	ETapi	ecInvalFeature
ETapiInvalGroupID	ETapi	ecInvalGroupID
ETapiInvalLineHandle	ETapi	ecInvalLineHandle
ETapiInvalLineState	ETapi	ecInvalLineState
ETapiInvalLocation	ETapi	ecInvalLocation
ETapiInvalMediaList	ETapi	ecInvalMediaList
ETapiInvalMediaMode	ETapi	ecInvalMediaMode
ETapiInvalMessageID	ETapi	ecInvalMessageID
ETapiInvalParam	ETapi	ecInvalParam
ETapiInvalParkID	ETapi	ecInvalParkID
ETapiInvalParkMode	ETapi	ecInvalParkMode
ETapiInvalPointer	ETapi	ecInvalPointer
ETapiInvalPrivSelect	ETapi	ecInvalPrivSelect
ETapiInvalRate	ETapi	ecInvalRate
ETapiInvalRequestMode	ETapi	ecInvalRequestMode
ETapiInvalTerminalID	ETapi	ecInvalTerminalID
ETapiInvalTerminalMode	ETapi	ecInvalTerminalMode

Table 18.2: *Additional Async Professional exception classes (continued)*

Exception	Ancestor	Error Code(s)
ETapiInvalTimeout	ETapi	ecInvalTimeout
ETapiInvalTone	ETapi	ecInvalTone
ETapiInvalToneList	ETapi	ecInvalToneList
ETapiInvalToneMode	ETapi	ecInvalToneMode
ETapiInvalTransferMode	ETapi	ecInvalTransferMode
ETapiLineMapperFailed	ETapi	ecLineMapperFailed
ETapiLoadFail	ETapi	ecTapiLoadFail
ETapiNoConference	ETapi	ecNoConference
ETapiNoDevice	ETapi	ecNoDevice
ETapiNoDriver	ETapi	ecNoDriver
ETapiNoMem	ETapi	ecNoMem
ETapiNoMultipleInstance	ETapi	ecNoMultipleInstance
ETapiNoRequest	ETapi	ecNoRequest
ETapiNoSelect	ETapi	ecTapiNoSelect
ETapiNotOwner	ETapi	ecNotOwner
ETapiNotRegistered	ETapi	ecNotRegistered
ETapiNotSet	ETapi	ecTapiNotSet
ETapiOperationFailed	ETapi	ecOperationFailed
ETapiOperationUnavail	ETapi	ecOperationUnavail
ETapiRateUnavail	ETapi	ecRateUnavail
ETapiReinit	ETapi	ecReinit
ETapiRequestOverrun	ETapi	ecRequestOverrun
ETapiResourceUnavail	ETapi	ecResourceUnavail
ETapiStructureTooSmall	ETapi	ecStructureTooSmall
ETapiTargetNotFound	ETapi	ecTargetNotFound
ETapiTargetSelf	ETapi	ecTargetSelf
ETapiTranslateFail	ETapi	ecTapiTranslateFail
ETapiUnexpected	ETapi	ecTapiUnexpected
ETapiUninitialized	ETapi	ecUninitialized

Table 18.2: *Additional Async Professional exception classes (continued)*

Exception	Ancestor	Error Code(s)
ETapiUserUserInfoTooBig	ETapi	ecUserUserInfoTooBig
ETapiVoiceNotSupported	ETapi	ecTapiVoiceNotSupported
ETapiWaveFail	ETapi	ecTapiWaveFail
ETimeout	EProtocol	ecTimeout
ETooManyErrors	EProtocol	ecTooManyErrors
ETracingNotEnabled	ESerialIO	ecTracingNotEnabled
ETriggerTooLong	ETrigger	ecTriggerTooLong
EUnpackAbort	EFax	ecUnpackAbort

As you can see, there is largely a one-to-one correspondence between exceptions and error codes.

In some cases, Async Professional catches and either handles or re-raises standard VCL exceptions. Examples include `EOutOfMemory` (when a memory allocation call fails), `EInOutError` (when an attempt to open, read, or write a file fails), and `EInvalidOperation` (when an improperly initialized form is used).

The `ErrorMsg` function of Async Professional (see page 895) can be used to generate an English-language string for each error code. The strings are stored in a string table in `APW.RC`, which has been compiled to `APW.R32`. This file is linked to your application's EXE file so that the error message text is available to your application. The following table shows the default string for each error code. These strings provide additional explanation of each error.

The `ErrorMsg` function of Async Professional (see page 895) can be used to generate an English-language string for each error code. The strings are stored in a string table in `APW.STR`, which has been compiled to either `APW.R16` (Delphi 1.0) or `APW.R32` (32-bit Delphi). The resources are then linked to your application's EXE file.

The Table 18.3 shows the default string for each error code. These strings provide additional explanation of each error. The error codes are shown here as absolute values. Note that the values may be positive or negative at different points during code execution.

Table 18.3: *Error code default strings*

Error Code	Value	Error Message
ecOK	0	OK
ecFileNotFound	2	File not found
ecPathNotFound	3	Path not found
ecTooManyFiles	4	Too many open files
ecAccessDenied	5	File access denied
ecInvalidHandle	6	Invalid file handle
ecOutOfMemory	8	Insufficient memory
ecInvalidDrive	15	Invalid drive
ecNoMoreFiles	18	No more files
ecDiskRead	100	Attempt to read beyond end of file
ecDiskFull	101	Disk is full
ecNotAssigned	102	File/device not assigned
ecNotOpen	103	File/device not open
ecNotOpenInput	104	File/device not open for input
ecNotOpenOutput	105	File/device not open for output
ecWriteProtected	150	Disk is write-protected
ecUnknownUnit	151	Unknown disk unit
ecDriveNotReady	152	Drive is not ready
ecUnknownCommand	153	Unknown command
ecCrcError	154	Data error
ecBadStructLen	155	Bad request structure length
ecSeekError	156	Seek error
ecUnknownMedia	157	Unknown media type
ecSectorNotFound	158	Disk sector not found
ecOutOfPaper	159	Printer is out of paper
ecDeviceWrite	160	Device write error

Table 18.3: *Error code default strings (continued)*

Error Code	Value	Error Message
ecDeviceRead	161	Device read error
ecHardwareFailure	162	General failure
ecBadHandle	1001	Bad handle passed to com function
ecBadArgument	1002	Bad argument passed to function
ecGotQuitMsg	1003	Got quit message
ecBufferTooBig	1004	Terminal buffer greater than 65521
ecPortNotAssigned	1005	ComPort component not assigned
ecInternal	1006	Internal error processing INI database
ecModemNotAssigned	1007	Modem component not assigned
ecPhonebookNotAssigned	1008	Phonebook component not assigned
ecCannotUseWithWinsock	1009	Component not compatible with Winsock
ecBadId	2001	ie_BadId -specified comport doesn't exist
ecBaudRate	2002	ie_Baudrate - unsupported baud rate
ecByteSize	2003	ie_Bytesize - invalid byte size
ecDefault	2004	ie_Default - error in default parameters
ecHardware	2005	ie_Hardware - specified comport in use
ecMemory	2006	ie_Memory - unable to allocate queues
ecCommNotOpen	2007	ie_NOpen - device not open
ecAlreadyOpen	2008	ie_Open - device already open
ecNoHandles	2009	No more handles, can't open port
ecNoTimers	2010	No timers available
ecNoPortSelected	2011	No port selected (attempt to open com0)
ecNullApi	3001	No device layer specified
ecNotSupported	3002	Function not supported by driver
ecRegisterHandlerFailed	3003	EnableCommNotification failed
ecPutBlockFail	3004	Failed to put entire block
ecGetBlockFail	3005	Failed to get entire block

Table 18.3: *Error code default strings (continued)*

Error Code	Value	Error Message
ecOutputBufferTooSmall	3006	Output buffer too small for block
ecBufferIsEmpty	3007	Buffer is empty
ecTracingNotEnabled	3008	Tracing not enabled
ecLoggingNotEnabled	3009	Logging not enabled
ecBaseAddressNotSet	3010	Base addr not found, RS485 mode
ecModemNotStarted	4001	StartModem has not been called
ecModemBusy	4002	Modem is busy elsewhere
ecModemNotDialing	4003	Modem is not currently dialing
ecNotDialing	4004	Dialer is not dialing
ecAlreadyDialing	4005	Dialer is already dialing
ecModemNotResponding	4006	No response from modem
ecModemRejectedCommand	4007	Bad command sent to modem
ecModemStatusMismatch	4008	Wrong modem status requested
ecNoMoreTriggers	5001	No more trigger slots
ecTriggerTooLong	5002	Data trigger too long
ecBadTriggerHandle	5003	Bad trigger handle
ecStartStringEmpty	5501	Start string is empty
ecPacketTooSmall	5502	Packet size cannot be smaller than start
ecNoEndCharCount	5503	CharCount packets must hand end condition
ecEmptyEndString	5504	End string is empty
ecZeroSizePacket	5505	Packet size cannot be zero
ecPacketTooLong	5506	Packet too long
ecBadFileList	6001	Bad format in file list
ecNoSearchMask	6002	No search mask specified during transmit
ecNoMatchingFiles	6003	No files matched search mask
ecDirNotFound	6004	Directory in search mask doesn't exist
ecCancelRequested	6005	Cancel requested

Table 18.3: *Error code default strings (continued)*

Error Code	Value	Error Message
ecTimeout	6006	Fatal time out
ecProtocolError	6007	Unrecoverable event during protocol
ecTooManyErrors	6008	Too many errors during protocol
ecSequenceError	6009	Block sequence error in Xmodem
ecNoFilename	6010	No filename specified in protocol receive
ecFileRejected	6011	File was rejected
ecCantWriteFile	6012	Can't write file
ecTableFull	6013	Kermit window table is full, fatal error
ecAbortNoCarrier	6014	Aborting due to carrier loss
ecBadProtocolFunction	6015	Function not supported by protocol
ecKeyTooLong	7001	Key string too long
ecDataTooLarge	7002	Data string too long
ecNoFieldsDefined	7003	No fields defined in database
ecIniWrite	7004	Generic INI file write error
ecIniRead	7005	Generic INI file read error
ecNoIndexKey	7006	No index defined for database
ecRecordExists	7007	Record already exists
ecRecordNotFound	7008	Record not found in database
ecMustHaveIdxVal	7009	Invalid index key name
ecDatabaseFull	7010	Maximum database records (999) reached
ecDatabaseEmpty	7011	No records in database
ecDatabaseNotPrepared	7012	iPrepareIniDatabase not called
ecBadFieldList	7013	Bad field list in INI component
ecBadFieldForIndex	7014	Bad field index in INI component
ecFaxBadFormat	8001	File is not an APF file
ecBadGraphicsFormat	8002	Unsupported graphics file format
ecConvertAbort	8003	User aborted fax conversion
ecUnpackAbort	8004	User aborted fax unpack

Table 18.3: *Error code default strings (continued)*

Error Code	Value	Error Message
ecCantMakeBitmap	8005	CreateBitmapIndirect API failure
ecNoImageLoaded	8050	No image loaded into viewer
ecNoImageBlockMarked	8051	No block of image marked
ecFontFileNotFound	8052	APFAXFNT not found, or resource bad
ecInvalidPageNumber	8053	Invalid page number specified for fax
ecBmpTooBig	8054	BMP size exceeds Windows max of 32767
ecEnhFontTooBig	8055	Font too big for enh text converter
ecFaxBadMachine	8060	Fax incompatible with remote fax
ecFaxBadModemResult	8061	Bad response from modem
ecFaxTrainError	8062	Modems failed to train
ecFaxInitError	8063	Error while initializing modem
ecFaxBusy	8064	Called fax number was busy
ecFaxVoiceCall	8065	Called fax number answered with voice
ecFaxDataCall	8066	Incoming data call
ecFaxNoDialTone	8067	No dial tone
ecFaxNoCarrier	8068	Failed to connect to remote fax
ecFaxSessionError	8069	Fax failed in mid-session
ecFaxPageError	8070	Fax failed at page end
ecFaxGDIPrintError	8071	NextBand GDI error in fax print driver
ecFaxMixedResolution	8072	Multiple resolutions in one session
ecFaxConverterInitFail	8073	Initialization of fax converter failed
ecUniAlreadyInstalled	8080	Unidrv support files already installed
ecUniCannotGetSysDir	8081	Cannot determine windows system dir
ecUniCannotGetWinDir	8082	Cannot determine windows dir
ecUniUnknownLayout	8083	Cannot determine setup file layout
ecUniCannotParseInfFile	8084	Cannot find Unidrv files in setup file
ecUniCannotInstallFile	8085	Cannot install Unidrv files to sys dir
ecNotNTDriver	8086	Printer driver not NT compatible
ecDrvCopyError	8087	Error copying printer driver

Table 18.3: *Error code default strings (continued)*

Error Code	Value	Error Message
ecCannotAddPrinter	8088	32-bit AddPrinter call failed
ecDrvBadResources	8089	Bad/missing resources in driver
ecDrvDriverNotFound	8090	Driver not found
ecUniCannotGetPrinterDir	8091	Cannot determine WinNT printer driver dir
ecInstallDriverFailed	8092	AddPrinterDriver API failed
ADWSERROR	9001	Async Professional Error
ADWSLOADERROR	9002	Error loading Winsock DLL
ADWSVERSIONERROR	9003	Incorrect version of Winsock
ADWSNOTINIT	9004	Winsock not initialized
ADWSINVPOR	9005	Specified port is not valid
ADWSCANTCHANGE	9006	Cannot change parameter while socket is connected
ADWSCANTRESOLVE	9007	Cannot resolve destination address
WSAEINTR	10004	Interrupted function call
WSAEBADF	10009	Bad file number
WSAEACCES	10013	Permission denied
WSAEFAULT	10014	Unknown error
WSAEINVAL	10022	Invalid argument
WSAEMFILE	10024	Too many open files
WSAEWOULDBLOCK	10035	Warning: the socket would block on this call
WSAEINPROGRESS	10036	A blocking call is in progress
WSAEALREADY	10037	WSAEALREADY: watch out, A1 is ready
WSAENOTSOCK	10038	Socket descriptor is (1) not a socket, or (2) is of wrong type
WSAEDESTADDRREQ	10039	The destination address is required for this operation
WSAEMSGSIZE	10040	The datagram was too large to fit into the buffer and was truncated
WSAEPROTOTYPE	10041	WSAEPROTOTYPE

Table 18.3: *Error code default strings (continued)*

Error Code	Value	Error Message
WSAENOPROTOOPT	10042	The option is unknown or not supported
WSAEPROTONOSUPPORT	10043	Either (1) no buffer space available so socket cannot be created or (2) protocol not supported
WSAESOCKTNOSUPPORT	10044	Specified socket type not supported in this address family
WSAEOPNOTSUPP	10045	Operation is not supported by this socket
WSAEPFNOSUPPORT	10046	Specified protocol family is not supported
WSAEAFNOSUPPORT	10047	Specified address family is not supported by this protocol
WSAEADDRINUSE	10048	The address is already in use for this operation
WSAEADDRNOTAVAIL	10049	The address is not available from this machine
WSAENETDOWN	10050	The network subsystem has failed
WSAENETUNREACH	10051	The network is unreachable from this machine at this time
WSAENETRESET	10052	The network has been reset
WSAECONNABORTED	10053	The virtual circuit has been aborted due to timeout, etc
WSAECONNRESET	10054	The virtual circuit has been reset by the partner
WSAENOBUFS	10055	The descriptor is not a socket, or no buffer space is available
WSAEISCONN	10056	The socket is already connected
WSAENOTCONN	10057	The socket is not connected
WSAESHUTDOWN	10058	The socket has been shutdown
WSAETOOMANYREFS	10059	WSAETOOMANYREFS
WSAETIMEDOUT	10060	The operation timed out
WSAECONNREFUSED	10061	The attempt to connect was forcibly refused
WSAELOOP	10062	WSAELOOP: see WSAELOOP

Table 18.3: *Error code default strings (continued)*

Error Code	Value	Error Message
WSAENAMETOOLONG	10063	The name is too long
WSAEHOSTDOWN	10064	The host machine is down
WSAEHOSTUNREACH	10065	The host machine is unreachable
WSAENOTEMPTY	10066	WSAENOTEMPTY
WSAEPROCLIM	10067	WSAEPROCLIM
WSAEUSERS	10068	WSAEUSERS
WSAEDQUOT	10069	WSAEDQUOT
WSAESTALE	10070	WSAESTALE
WSAEREMOTE	10071	WSAEREMOTE
WSASYSNOTREADY	10091	Network subsystem unusable
WSAVERNOTSUPPORTED	10092	Version requested by WSASStartUp not supported by loaded Winsock DLL
WSANOTINITIALISED	10093	WSASStartUp not yet called
WSAEDISCON	10101	WSAEDISCON
WSAHOST_NOT_FOUND	11001	Host not found
WSATRY_AGAIN	11002	Host not found, or SERVERFAIL, can try again
WSANO_RECOVERY	11003	Non recoverable errors, FORMERR, REFUSED, NOTIMP
WSANO_DATA	11004	Valid name, but no data record of requested type
WSANO_ADDRESS	11004	No address
ecAllocated	13801	Already allocated
ecBadDeviceID	13802	Bad device ID
ecBearerModeUnavail	13803	Bearer mode unavailable
ecCallUnavail	13805	Call unavailable
ecCompletionOverrun	13806	Completion overrun
ecConferenceFull	13807	Conference full
ecDialBilling	13808	Dial failed
ecDialDialtone	13809	Dial failed, no dialtone
ecDialPrompt	13810	Dial failed

Table 18.3: *Error code default strings (continued)*

Error Code	Value	Error Message
ecDialQuiet	13811	Dial failed
ecIncompatibleApiVersion	13812	Incompatible API version
ecIncompatibleExtVersion	13813	Incompatible EXT version
ecIniFileCorrupt	13814	INI file corrupt
ecInUse	13815	Resource in use
ecInvalAddress	13816	Invalid address
ecInvalAddressID	13817	Invalid address ID
ecInvalAddressMode	13818	Invalid address mode
ecInvalAddressState	13819	Invalid address state
ecInvalAppHandle	13820	Invalid application handle
ecInvalAppName	13821	Invalid application name
ecInvalBearerMode	13822	Invalid bearer mode
ecInvalCallComplMode	13823	Invalid call completion mode
ecInvalCallHandle	13824	Invalid call handle
ecInvalCallParams	13825	Invalid call parameters
ecInvalCallPrivilege	13826	Invalid call privilege
ecInvalCallSelect	13827	Invalid call select
ecInvalCallState	13828	Invalid call state
ecInvalCallStatelist	13829	Invalid call state list
ecInvalCard	13830	Invalid card
ecInvalCompletionID	13831	Invalid completion ID
ecInvalConfCallHandle	13832	Invalid conference call handle
ecInvalConsultCallHandle	13833	Invalid consultation call handle
ecInvalCountryCode	13834	Invalid country code
ecInvalDeviceClass	13835	Invalid device class
ecInvalDeviceHandle	13836	Invalid device handle
ecInvalDialParams	13837	Invalid dial params
ecInvalDigitList	13838	Invalid digit list
ecInvalDigitMode	13839	Invalid digit mode

Table 18.3: *Error code default strings (continued)*

Error Code	Value	Error Message
ecInvalDigits	13840	Invalid digits
ecInvalExtVersion	13841	Invalid ext version
ecInvalGroupID	13842	Invalid group ID
ecInvalLineHandle	13843	Invalid line handle
ecInvalLineState	13844	Invalid line state
ecInvalLocation	13845	Invalid location
ecInvalMediaList	13846	Invalid media list
ecInvalMediaMode	13847	Invalid media mode
ecInvalMessageID	13848	Invalid message ID
ecInvalParam	13850	Invalid parameter
ecInvalParkID	13851	Invalid park ID
ecInvalParkMode	13852	Invalid park mode
ecInvalPointer	13853	Invalid pointer
ecInvalPrivSelect	13854	Invalid privilege select
ecInvalRate	13855	Invalid rate
ecInvalRequestMode	13856	Invalid request mode
ecInvalTerminalID	13857	Invalid terminal ID
ecInvalTerminalMode	13858	Invalid terminal mode
ecInvalTimeout	13859	Invalid timeout
ecInvalTone	13860	Invalid tone
ecInvalToneList	13861	Invalid tone list
ecInvalToneMode	13862	Invalid tone mode
ecInvalTransferMode	13863	Invalid transfer mode
ecLineMapperFailed	13864	Line mapper failed
ecNoConference	13865	No conference
ecNoDevice	13866	No device
ecNoDriver	13867	No driver
ecNoMem	13868	No memory
ecNoRequest	13869	No request

Table 18.3: *Error code default strings (continued)*

Error Code	Value	Error Message
ecNotOwner	13870	Not owner
ecNotRegistered	13871	Not registered
ecOperationFailed	13872	Operation failed
ecOperationUnavail	13873	Operation unavailable
ecRateUnavail	13874	Rate unavailable
ecResourceUnavail	13875	Resource unavailable
ecRequestOverrun	13876	Request overrun
ecStructureTooSmall	13877	Structure too small
ecTargetNotFound	13878	Target not found
ecTargetSelf	13879	Target is self
ecUninitialized	13880	Uninitialized
ecUserUserInfoTooBig	13881	User info too big
ecReinit	13882	Reinit failed
ecAddressBlocked	13883	Address blocked
ecBillingRejected	13884	Billing rejected
ecInvalFeature	13885	Invalid feature
ecNoMultipleInstance	13886	No multiple instance
ecTapiBusy	13928	TAPI already open, dialing or answering
ecTapiNotSet	13929	TapiMode not set in TApdComPort
ecTapiNoSelect	13930	No TAPI device selected
ecTapiLoadFail	13931	Failed to find/load TAPIDLL
ecTapiGetAddrFail	13932	Failed to get TAPI address
ecTapi16Disabled	13933	TAPI disabled for 16-bit environments
ecTapiUnexpected	13934	Unexpected TAPI error
ecTapiVoiceNotSupported	13935	TAPI device does not support voice
ecTapiWaveFail	13936	TAPI wave file error
ecTapiCIDBlocked	13937	Caller ID blocked
ecTapiCIDOutOfArea	13938	Out of area call
ecTapiWaveFormatError	13939	The selected file is not a wave file

Table 18.3: *Error code default strings (continued)*

Error Code	Value	Error Message
ecTapiWaveReadError	13940	Unable to read wave file data
ecTapiWaveBadFormat	13941	Unsupported wave format
ecTapiTranslateFail	13942	Unable to translate address
ecTapiWaveDeviceInUse	13943	Wave device in use
ecTapiWaveFileExists	13944	Wave file already exists
ecTapiWaveNoData	13945	No wave data available

The “ie_” designations refer to error codes returned by the Windows communications API.

All of the Async Professional error codes are defined in numeric order in the source file OOMISC.PAS. All of the exception classes are declared in ADEXCEPT.PAS.

Conditional Defines

Before using Async Professional, you need to understand and perhaps modify various conditional defines in `AWDEFINE.INC`. This file is included into all Async Professional units. To modify it, load it into a text editor and find the appropriate conditional symbol. Insert a period between the ‘{’ and the ‘\$’ of a conditional define to deactivate it. With the period in place, the compiler sees the line as a plain comment, which has no effect on compilation.

After modifying `AWDEFINE.INC`, save it to disk and use the compiler to rebuild all affected files. To be sure the library is rebuilt properly follow these steps:

1. Delete all `*.OBJ`, `*.HPP`, and `*.DCU` files in the `\ASYNCPRO` directory. Be careful not to delete any other files.
2. From the C++Builder main menu choose Component|Rebuild Library.
3. After modifying `AWDEFINE.INC`, save it to disk and use the compiler to rebuild all affected files.

The following paragraphs describe the options that can be controlled through `AWDEFINE.INC`. The default state of each define is also shown.

`{.$DEFINE EnableTapi16}`

This define enables TAPI components in 16-bit applications. Typically TAPI can't be used in 16-bit environments due to a lack of service providers (notably, the lack of `UNIMDM.TSP`) in Windows 3.XX, and due to a bug in Windows 95 that prevents obtaining a 16-bit comm handle to a TAPI port. Therefore this define is off by default. You can enable it to use TAPI in 16-bit environments, but only if you are certain that environment properly supports TAPI.

`{.$DEFINE Prot16OpenStrings}`

This define activates an Async Professional 1.0 behavior for Async Professional 2.0 programs. When this define is active, Async Professional 2.0 uses “OpenString” (the same type used by Async Professional 1.0) for string parameters passed in the `TapdProtocol` component `OnProtocolNextFile` and `OnProtocolAccept` events. When this define is not active, Async Professional 2.0 uses `TPassString` for those events. This define is off by default and is intended only to provide backward compatibility with existing Async Professional 1.0 programs.

Glossary

This glossary contains a combination of industry accepted definitions and, where noted, definitions that are unique to Async Professional.

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-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.

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.

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.

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 applications 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.

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.

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.

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.

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 unrati ed 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.

Debugging Windows Communications Programs

This is a list of tips and techniques for debugging Windows communications programs. Some of these suggestions are very simple and you probably already use them. However, some of them are specific to communications programs and might cover issues you haven't had to deal with before.

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 TCom. If TCom 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.

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, and probably most useful, 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.

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. 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 page 33 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, precompiled demonstration programs TermDemo or TCom. 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 DispatchLogging, 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 TCom 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 go any further. Common problems are two or more UARTs using the same IRQ, another board (e.g., a mouse or network board) using a serial port IRQ, or two or more UARTs using the same I/O address.

Another possibility is misnamed ports if there is a gap in the serial ports in your machine. For example, if your machine has serial ports COM1, COM2, and COM4, Windows names these ports COM1, COM2, and COM3. If a Windows communication program attempts to open COM4, it will fail since Windows doesn't recognize that COM4 exists.

The simplest solution to this problem is to accept the Windows name for the port and add appropriate COMXBASE and COMXIRQ statements to SYSTEM.INI to reflect the actual hardware configuration. To make Windows use COM3 in the above example, you would add

```
COM3BASE = 2E8
```

```
COM3IRQ = 4
```

to SYSTEM.INI.

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 Trace 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 psLong-Packet 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. Consult your modem manual for the hardware flow control enable command for your modem.

Identifier Index

A

Abort 562
AbortNoCarrier 526
AbortNoConnect 717, 845
AcceptDragged 656
AcceptSocket 117
Account 562
ActivateDeviceLayer 49
Active 181, 276
ActiveColor 181
ActivePage 656
ActualBPS 526
Add 241, 248
AddDataTrigger 49
AddFaxRecipient 816
AddMessage 876
AddModem 457
AddRecipient 770
Address 104, 873
AddStatusTrigger 50
AddTimerTrigger 51
AddTraceEntry 52
AdXxx 4
aetXxx 520, 527, 529
Age 315
ANSIMode 265
AnswerOnRing 409, 464, 758, 781
apFirstCall 489
ApiVersion 409
apLastCall 489
AppKeyMode 265
AppKeypadMode 265
AproReg 6
APW_XXX 70
Argument 230, 235
ArgumentCount 230
AsciiCharDelay 527
AsciiCRTranslation 527
AsciiEOFTimeout 528
AsciiEOLChar 528
AsciiLFTranslation 529
AsciiLineDelay 529
AsciiSuppressCtrlZ 529
AskAreaCode 352
AskForDate 360
AskForDateEx 360
AskForExtension 361
AskForExtensionEx 362
AskForList 362
AskForListEx 363
AskForPhoneNumber 363
AskForPhoneNumberEx 364
AskForSpelling 365
AskForSpellingEx 365
AskForTime 366
AskForTimeEx 366
AskForYesNo 367
AskForYesNoEx 367
AskLastFour 352
AskNextThree 353
asXxx 631, 657
Attempt 409
Attributes 277
AudioInDevice 302
AudioOutDevice 302
AutoAnswer 410, 464
AutoEnable 138
AutoOpen 52
AutoRepeat 266
AutoScaleMode 631, 657
AvailableTerminalDevices 302
AvgWaveInAmplitude 410

B

BackColor 205, 278
 BaseAddress 53
 Batch 530
 Baud 53
 bcmXxx 530
 BeginUpdate 657
 BGColor 658
 BindSocket 117
 BlindDial 743, 781, 845
 BlinkPaint 256
 BlinkTime 278
 BlockCheckMethod 530
 BlockErrors 531
 BlockLength 531
 BlockNumber 531
 BPSRate 411, 465
 BREAKLight 191
 BreakOffTimeout 193
 Buffer 256
 BufferFull 54
 BufferMinimum 743, 781
 BufferResume 54
 BusyCursor 658
 BytesRemaining 532
 BytesTransferred 532, 562, 717, 782

C

CallBackNumber 374
 CallerID 412
 CallerIDName 412
 CallInfo 303
 Cancel 173
 CancelCall 303, 413, 465, 845
 CancelFax 717, 782
 Cancelled 413
 CancelProtocol 533
 CancelRecipient 771
 CancelScript 157

Capacity 876
 Caption 480, 680
 Capture 278
 CaptureFile 279
 ChangeDir 563
 CharHeight 280
 CharReady 54
 CharSet 205, 280, 338
 CharSetMapping 256
 CharWidth 281
 CheckLoaded 118
 Clear 231, 241, 249, 281, 876
 ClearAll 282
 ClearAllHorzTabStops 205
 ClearAllVertTabStops 206
 ClearHorzTabStop 206
 ClearVertTabStop 206
 CloseFile 604
 CloseSocket 118
 Col 206
 Col132Mode 266
 ColCount 207
 Columns 282
 Command 231
 CommDelay 857
 ComName 895
 ComNumber 55
 ComPort 138, 157, 173, 193, 283, 413, 466,
 533, 718, 783, 879
 CompressionMode 374
 CompressRasterLine 604
 ConcatFax 771
 ConcatFaxes 744
 Conditions 182
 ConfigAndOpen 414, 466
 Connect 304, 879
 Connected 304, 563, 879
 Connection 374
 ConnectSocket 118
 ConnectState 374
 ConnectTimeout 563
 ConstantStatus 758, 783

Convert 605
 ConvertBitmapToFile 607
 ConvertCover 744
 ConvertToFile 607
 CopyCallInfo 415
 CopyToClipboard 283, 659
 Count 241, 249, 315, 325
 CoverFile 744
 CoverFileName 816
 coXxx 619
 Create 207, 231, 283
 CreateDisplay 441, 580, 691, 824
 CreateNewDetailFile 457
 CreatePhonebookEntry 375
 CreateSocket 119
 CreateWnd 284
 CTS 55
 CTSLight 191
 CTSMask 68
 CurrentEngine 325
 CurrentJobName 783
 CurrentJobNumber 784
 CurrentPage 719, 784
 CurrentPrintingPage 680
 CurrentRecipient 784
 CurrentState 174
 CurrentVoice 316

D

Data 174
 DataBits 56
 DataSize 174
 DataString 174
 DCD 56
 DCDLight 191
 DCDMask 68
 dcXxx 299
 DefBackColor 207
 DefCharSet 208
 DefForeColor 208

DefUserExtension 608
 DelayBetweenSends 785
 DelayTicks 890
 Delete 564, 876
 DeleteChars 208
 DeleteFailed 584
 DeleteLines 208
 DeleteModem 458
 DeleteModemRecord 458
 DeletePhonebookEntry 375
 DeltaCTS 56
 DeltaCTSMask 68
 DeltaDCD 57
 DeltaDCDMask 68
 DeltaDSR 57
 DeltaDSRMask 68
 DeltaRI 57
 DeltaRIMask 68
 Description 119
 DesiredBPS 719, 785
 DesiredECM 720, 785
 DestinationDir 759, 786
 DestinationDirectory 533
 Destroy 284
 DestroyDisplay 441, 580, 691, 824
 DestroyWnd 284
 DetectBusy 745
 DeviceClass 299
 DeviceCount 415
 DeviceInUse 299
 DeviceLayer 58, 109
 DeviceName 300, 375
 DeviceType 376
 deXxx 841
 dfXxx 584
 Dial 376, 415, 467
 DialAttempt 745, 786, 846
 DialAttempts 746, 786, 846
 DialDlg 377
 Dialect 316, 325
 Dialing 416
 DialMode 377

DialOptions 378
 DialPrefix 746, 787, 846
 DialRetryWait 746, 847
 DialStatusMsg 847
 DialTimeout 467
 DialWait 747, 787, 847
 Dictation 338
 Disconnect 848
 Display 441, 580, 691, 824
 DisplayToTerminal 157
 dlXxx 58, 109
 DoBackHorzTab 209
 DoBackspace 209
 DoBackVertTab 209
 DoCarriageReturn 209
 DocName 835
 DocumentFile 608
 DoHorzTab 210
 DoLineFeed 210
 Domain 378
 DoVertTab 210
 DSR 58
 DSRLight 191
 DSRMask 68
 dsXxx 842
 DTR 59

E

ecXxx 706, 901
 EditPhonebookEntry 378
 ElapsedTicks 534
 ElapsedTime 890
 ElapsedTimeInSecs 891
 Emulator 284
 Enabled 138
 EnableVoice 417
 EndCond 139
 EndString 140
 EndUpdate 659
 EngineFeatures 316, 326

EngineID 317, 326
 EnhFont 609, 747, 787
 EnhHeaderFont 747, 788
 EnhTextEnabled 748, 788
 EntryName 378
 EraseChars 211
 EraseFromBOL 211
 EraseFromBOW 211
 EraseLine 211
 EraseScreen 212
 EraseToEOL 212
 EraseToEOW 212
 ErrorCode 880, 900
 ERRORLight 191
 ErrorMessage 895
 ErrorOffTimeout 193
 EstimateTransferSecs 534
 EventTimer 890
 ExitOnError 720, 788, 848
 ExtractAPF 772
 ExtractCoverFile 773
 ExtractPage 631
 EXxx (exception) 901

F

FailureCode 468
 FailureCodeMsg 468
 Fax 824, 829
 FaxAndData 759
 FaxClass 721, 789
 FaxFile 721, 789
 FaxFileExt 721, 789
 FaxFileList 748
 FaxFileName 816
 FaxFooter 680
 FaxHeader 681
 FaxHistoryName 829
 FaxLog 722, 790
 FaxNameMode 760, 790
 FaxPrinter 691, 696, 790

FaxPrinterLog 681
 FaxProgress 722, 791
 FaxResolution 632, 682
 FaxWidth 633, 682
 fcXxx 59, 721, 734, 789
 Features 317, 326
 ffXxx 610
 FGColor 659
 FileDate 535
 FileLength 536, 564
 FileMask 536
 FileName 536, 660, 682, 835
 FileType 565
 FilterUnsupportedDevices 419
 Find 318
 FinishWait 537
 FirstPage 660
 FirstPageToPrint 683
 FlowState 59
 FlushInBuffer 60
 FlushOutBuffer 60
 fnXxx 755, 760
 FontFile 609
 FontType 610
 ForceSendQuery 791
 ForeColor 212, 285
 fpXxx 709
 frXxx 621, 632, 682
 FtpLog 565
 fwXxx 633, 682
 fwXxx (fax converter) 623

G

Gender 320
 GenDevConfig 419
 GenerateDrawScript 249
 Get 242
 GetBlock 61
 GetChar 62
 GetCharAttrs 213

GetCursorPos 257
 GetDefCharAttrs 213
 GetDevConfig 468
 GetDialParameters 379
 GetErrorText 379
 GetInvalidRect 214
 GetJob 810
 GetJobHeader 773
 GetLineAttrPtr 214
 GetLineBackColorPtr 215
 GetLineCharPtr 215
 GetLineCharSetPtr 215
 GetLineForeColorPtr 215
 GetModem 458
 GetModems 459
 GetNextDrawCommand 250
 GetNextFax 810
 GetRasterLine 610
 GetRecipient 774
 GetSchedTime 811
 GetStatusText 379
 Glyph 182, 189
 GlyphCells 183
 GPOMode 266
 Grammars 328
 GSMState 881
 gsXxx 881

H

HalfDuplex 286
 Handle 119
 HandshakeRetry 538
 HandshakeWait 538
 Hangup 380
 HangupCode 722, 791
 HangupOnDestroy 380
 HasCursorMoved 216
 HasDisplayChanged 216
 HeaderLine 748, 816
 HeaderRecipient 749, 817

HeaderSender 749
 HeaderTitle 750, 817
 Help 353, 566
 Help2 353
 HighVersion 119
 HistoryName 584, 864
 HonorDirectory 538
 HorizDiv 633, 661
 HorizMult 634, 661
 HorizScroll 661
 htonl 120
 htons 120
 HWFlowOptions 62
 hwXxx 62

I

idXxx 608, 611
 IgnoreCase 140
 InactiveColor 183
 InBuffFree 63
 InBuffUsed 64
 IncludeDirectory 539
 IncludeStrings 140
 InFileName 634
 InitBaud 724, 793
 InitialPosition 539
 InitModemForFaxReceive 760
 InProgress 158, 540, 566
 InputDocumentType 611
 InsertLines 216
 InSize 64
 Interfaces 320, 329
 Interlace 266
 InterruptWave 420
 InVT52Mode 236
 itXxx 339

J

JobFileExt 811

JobFileName 818
 JobName 818

K

KermitCtlPrefix 540
 KermitHighbitPrefix 540
 KermitLongBlocks 541
 KermitMaxLen 541
 KermitMaxWindows 541
 KermitPadCharacter 542
 KermitPadCount 542
 KermitRepeatPrefix 542
 KermitSWCTurnDelay 543
 KermitTerminator 543
 KermitTimeoutSecs 544
 KermitWindowsTotal 544
 KermitWindowsUsed 544
 KeyboardMapping 257
 KeyDown 258
 KeyPress 258

L

LanguageID 321, 329
 LastError 120
 LastErrorCode 175
 LastPage 662
 LastPageToPrint 683
 LazyByteDelay 286
 LazyPaint 259
 LazyTimeDelay 286
 lcXxx 684, 696, 697
 LEDs 267
 LeftMargin 613
 leXxx 65
 lfaxXxx 713, 729
 lfXxx 547
 Lights 193
 Line 287
 LineBreak 64

LineError 65
 LinesPerPage 613
 ListConnections 380
 ListDir 566
 Listen 338
 ListenSocket 121
 ListEntries 381
 Lit 189
 LitColor 189
 LoadFromFile 242, 250
 LoadFromRes 243, 252
 LoadWholeFax 662
 LocalAddress 121
 LocalHost 121
 LogFileName 696
 Logging 66
 LogHex 67
 Login 567
 LogName 67
 Logout 568
 LogSize 67
 LookupAddress 122
 LookupName 122
 LookupPort 123
 LookupService 124
 lsXxx 89
 ltapiXxx 399, 446

M

Main 354
 Main2 354
 MakeDir 568
 MakeEndOfPage 614
 MakeFaxJob 818
 MakeJob 775
 MaxAttempts 420
 MaxMessageLength 420, 848
 MaxSendCount 750, 794
 MaxSockets 124
 MaxWordsState 329

MaxWordsVocab 330
 mdXxx 300
 MediaDirection 300
 MediaType 300
 Message 840, 848, 873
 MessageIndex 873
 Messages 877
 MessageStore 882
 MfgName 321, 330
 mlXxx 474
 ModeID 321, 330
 ModemBPS 725, 795
 ModemCapFolder 459, 469
 ModemChip 726, 795
 ModemECM 726, 795
 ModemInit 726, 796, 849
 ModemLogToString 469
 ModemModel 727, 796
 ModemRevision 727, 796
 ModemState 469
 ModemStatus 68
 ModeName 322, 331
 MonitorDir 811
 MonitorDlg 381
 Monitoring 194, 797
 MoveCursorDown 217
 MoveCursorLeft 217
 MoveCursorRight 217
 MoveCursorUp 218
 msXxx 88, 470
 MultiPage 683

N

NeedsUpdate 259
 NegotiationResponses 470
 NetAddr2String 124
 NewLineMode 267
 NewTimer 891
 NewTimerSecs 892
 NextPage 663

NoAnswerMax 368
 NoAnswerTime 368
 NormalBaud 727, 797
 NotifyOnNewMessage 882
 NotLitColor 190
 ntohl 125
 ntohs 125
 Number 421
 NumDigits 368
 NumPages 635, 663

O

OnCloseUserFile 614
 OnConnect 304
 OnConnected 382
 OnDialError 382, 849
 OnDialStatus 383, 850
 OnDisconnect 305
 OnDisconnected 383
 OnDocEnd 835
 OnDocStart 836
 OnDropFile 664
 OneFax 761
 OnFail 305
 OnFaxAccept 761
 OnFaxError 728
 OnFaxFinish 728
 OnFaxLog 729
 OnFaxName 762
 OnFaxNext 751
 OnFaxPrintLog 684
 OnFaxPrintStatus 684
 OnFaxServerAccept 797
 OnFaxServerFatalError 798
 OnFaxServerFinish 799
 OnFaxServerLog 799
 OnFaxServerName 800
 OnFaxServerPortOpenClose 800
 OnFaxServerStatus 800
 OnFaxStatus 729
 OnFtpError 568
 OnFtpLog 569
 OnFtpReply 569
 OnFtpStatus 569
 OnGSMComplete 882
 OnIncomingCall 306
 OnInterference 339
 OnLogin 857
 OnLogout 857
 OnModemCallerID 471
 OnModemConnect 472
 OnModemDisconnect 472
 OnModemFail 473
 OnModemLog 473
 OnModemStatus 474
 OnNewMessage 883
 OnNextMessage 883
 OnNextPage 685
 OnOpenUserFile 615
 OnOutputLine 616, 635
 OnPacket 141
 OnPageChange 665
 OnPhraseFinish 340
 OnPhraseHypothesis 340
 OnProtocolAccept 545
 OnProtocolError 545
 OnProtocolFinish 546
 OnProtocolLog 546
 OnProtocolNextFile 547
 OnProtocolStatus 548
 OnReadUserLine 617
 OnScriptCommandFinish 158
 OnScriptCommandStart 159
 OnScriptDisplay 159
 OnScriptFinish 160
 OnScriptParseVariable 160
 OnScriptUserFunction 161
 OnSNPPError 858
 OnSNPPSuccess 858
 OnSpeakStart 340
 OnSpeakStop 340
 OnSRError 341

- OnSRWarning 341
- OnSSAttributeChanged 341
- OnSSError 342
- OnSSWarning 342
- OnStateActivate 184
- OnStateChange 175
- OnStateFinish 184
- OnStateMachineFinish 176
- OnStatus 617, 636
- OnStringPacket 142
- OnTAPFinish 850
- OnTapiCallerID 422
- OnTapiConnect 422
- OnTapiDTMF 423
- OnTapiFail 423
- OnTapiLog 424
- OnTapiPortClose 424
- OnTapiPortOpen 425
- OnTapiStatus 425
- OnTapiWaveNotify 426
- OnTapiWaveSilence 427
- OnTAPStatus 842, 851
- OnTimeout 141
- OnTrainingRequested 342
- OnTrigger 26, 69
- OnTriggerAvail 71
- OnTriggerData 73
- OnTriggerLineError 73
- OnTriggerModemStatus 74
- OnTriggerOutbuffFree 75
- OnTriggerOutbuffUsed 76
- OnTriggerOutSent 77
- OnTriggerStatus 25, 78
- OnTriggerTimer 79
- OnViewerError 665
- OnVUMeter 343
- OnWsAccept 109, 126
- OnWsConnect 110, 126
- OnWsDisconnect 111, 126
- OnWsError 111, 127, 571
- OnWsRead 127
- OnWsWrite 127

- Open 80, 112
- OpenFile 619
- Options 368, 619, 637
- OriginCol 218
- OriginRow 218
- OutBuffFree 80
- OutBuffUsed 81
- OutFileName 620, 638
- Output 81
- OutputOnActivate 185
- OutSize 82
- Overhead 548

P

- PacketSize 142
- PageBitmaps 666
- PageHeight 667
- PageLength 730, 801
- Pager 864
- PagerID 840
- PagerLog 851
- PageWidth 667
- Paint 260
- Parity 82
- Parser 260
- Password 104, 383, 571
- Paused 813
- PauseListening 343
- PauseSpeaking 343
- Phonebook 384
- PhonebookDlg 384
- PhoneNumber 384, 752, 819, 851
- PlatformID 385
- PlayWaveFile 427
- Port 104, 852, 858
- ppXxx 685, 686
- PrepareConnectInProgress 762
- PrepareScript 161
- PrevPage 667
- PrintAbort 685

PrintFax 686
 PrintOnReceive 801
 PrintProgress 686
 PrintScale 687
 PrintSetup 687
 ProcessChar 232, 236
 ProcessCommunications 82
 ProcessWideChar 233
 ProductName 322, 331
 PromptForPort 83
 Prompts 369
 Protocol 162, 580, 585
 ProtocolError 549
 ProtocolLog 549
 ProtocolName 897
 ProtocolStatus 549
 ProtocolType 550
 prXxx 356
 psXxx 369, 491, 687, 842
 ptXxx 550
 ptXxx (protocol type) 897
 PutBlock 84
 PutChar 84
 PutString 84
 pXxx 82
 ReplyCode 572
 ReSend 852
 Reset 219
 Resolution 621
 RestartAt 572
 ResumeListening 343
 ResumeSpeaking 344
 Retrieve 572
 RetryWait 428
 RevScreenMode 268
 RI 86
 RIMask 68
 RingCount 474
 RINGLight 191
 RingOffTimeout 194
 RingWaitTimeout 475
 Rotation 668
 Row 219
 RowCount 219
 Rows 287
 RS485Mode 86
 RTS 86
 RTSLowForWrite 550
 RXDLight 191
 RXDOffTimeout 194

Q

QuickConnect 884

R

ReadSocket 127
 Recipient 820
 RelOriginMode 267
 RemainingTime 892
 RemainingTimeInSecs 892
 RemoteID 730, 802
 RemoveAllTriggers 85
 RemoveTrigger 85
 Rename 571

S

SafeMode 752, 802
 SapiEngine 369
 SaveWaveFile 428
 Scaling 638, 669
 ScheduledDateTime 820
 ScriptCommands 162
 ScriptFile 162
 Scrollback 288
 ScrollbackRows 289
 scXxx 143, 570
 Secs2Ticks 893
 SelectDevice 429, 475
 SelectedDevice 429, 476

SelectImage 669
 SelectModem 459
 SelectRegion 670
 Send 840, 852, 859
 SendAllMessages 884
 SendBreak 87
 SendCommand 476
 Sender 821
 SendFtpCommand 573
 SendMessage 884
 SendQueryInterval 802
 SendTone 429
 Sequence 233
 Sequencing 331
 ServerAddress 574
 ServerDataInput 859
 ServerDoneString 860
 ServerInitString 860
 ServerManager 803
 ServerResponseFailContinue 860
 ServerResponseFailTerminate 861
 ServerSuccessString 861
 SessionBPS 730, 803
 SessionECM 731, 803
 SessionResolution 731, 804
 SessionWidth 732, 804
 SetAsyncStyles 128
 SetCharAttrs 220
 SetCursorPosition 220
 SetDefCharAttrs 221
 SetDevConfig 430, 477
 SetDialParameters 385
 SetHorzTabStop 221
 SetRecordingParams 430
 SetScrollRegion 221
 SetStatusTrigger 88
 SetTimerTrigger 90
 SetVertTabStop 221
 sfXxx 327
 sgXxx 328
 ShowAboutDlg 344
 ShowConfigDialog 430
 ShowConfigDialogEdit 431
 ShowFaxJobInfoDialog 778
 ShowGeneralDlg 344
 ShowLexiconDlg 344
 ShowMediaSelectDialog 306
 ShowPorts 431
 ShowTapiDevices 432
 ShowTrainGeneralDlg 345
 ShowTrainMicDlg 345
 SilenceThreshold 432
 SmoothScrollMode 268
 SMSAddress 885
 smXxx 575
 SocksVersion 104
 SoftwareFlow 732, 805
 Speak 345, 369
 Speaker 322
 SpeakerMode 385
 SpeakFile 346
 SpeakFileToFile 347
 SpeakStream 347
 SpeakToFile 348
 SRAmplitude 349
 SRAutoGain 349
 SREngines 338, 349
 SSVoices 349
 ssXxx 332, 866, 874
 Start 176
 StartCond 143
 Started 480
 StartManualReceive 762
 StartManualTransmit 752
 StartReceive 551, 763
 StartScript 163
 StartState 176
 StartString 143
 StartTransmit 551, 753
 StartWaveRecord 432
 StateNames 177
 StationID 621, 733, 805
 Status 574, 874
 StatusDialog 481

StatusDisplay 386, 433, 477, 552, 687, 733, 806
 StatusInterval 552, 734, 806
 StatusMsg 553, 734, 806
 StopBits 91
 StopListening 349
 StopSpeaking 350
 StopWaveFile 434
 StopWaveRecord 434
 Store 575
 StoreToBinFile 244, 252
 String2NetAddr 128
 stXxx 775, 777
 Style 323
 SupportedFaxClasses 734, 807
 SVRowCount 222
 svXxx 104
 SWFlowOptions 91
 swfXxx 91
 SystemStatus 129

T

TabStop 622
 TAdCharSetMapping 245
 TAdKeyboardMapping 238
 TAdKeyString 241, 248
 TAdModem 461
 TAdModemStatus 478
 TAdParserCmdType 232, 233
 TAdTerminal 269
 TAdTerminalBuffer 201
 TAdTerminalCharAttr 213, 220
 TAdTerminalCharAttrs 213, 220
 TAdTerminalEmulator 253
 TAdTerminalParser 224, 229
 TAdTTYEmulator 262
 TAdVT100Emulator 263
 TAdVT100Parser 225, 234
 TApdAbstractFax 715
 TApdAbstractFaxPrinterStatus 689

TApdAbstractFaxStatus 822
 TApdAbstractPager 839
 TApdAbstractStatus 578
 TApdAbstractTapiStatus 440
 TApdAudioInDevice 324
 TApdAudioOutDevice 314
 TApdBaseComponent 8
 TApdBaseDispatcher 49
 TApdBaseXxx 8
 TApdComPort 22
 TApdCustomModemPager 841
 TApdCustomSapiEngine 333
 TApdCustomSapiPhone 356
 TApdDataPacket 132
 TApdFaxClient 815
 TApdFaxConverter 594
 TApdFaxDriverInterface 834
 TApdFaxJobHandler 769
 TApdFaxLog 828
 TApdFaxPrinter 674
 TApdFaxPrinterLog 695
 TApdFaxPrinterMargin 680, 681
 TApdFaxPrinterStatus 693
 TApdFaxServerManager 809
 TApdFaxStatus 826
 TApdFaxUnpacker 624
 TApdFaxViewer 649
 TApdFtpClient 559
 TApdGSMPhone 865, 878
 TApdLibModem 449
 TApdMessageStore 875
 TApdPagerLog 862
 TApdProtocol 523
 TApdProtocolLog 583
 TApdProtocolStatus 582
 TApdRasCompressionMode 374
 TApdRasConnectedEvent 382
 TApdRasDialer 372
 TApdRasDialMode 377
 TApdRasErrorEvent 382
 TApdRasSpeakerMode 385
 TApdRasStatus 387

TApdRasStatusEvent 383
 TApdReceiveFax 754
 TApdSapiEngine 337
 TApdSapiPhone 359
 TApdSapiPhonePrompts 351
 TApdSendFax 736
 TApdSLController 191
 TApdSMSMessage 872
 TApdSNPPPager 856
 TApdSocket 115
 TApdState 178
 TApdStateMachine 172
 TApdStatusLight 188
 TApdTapiDevice 407
 TApdTapiLog 445
 TApdTapiStatus 443
 TApdTAPPager 841
 TApdVoIP 301
 TApdVoIPTerminal 298
 TApdWinsockPort 106
 TapiDevice 441, 446, 735, 807
 TapiHistoryName 446
 TapiLog 434
 TapiMode 92
 TapiState 435
 TapiStatusMsg 436
 TAPStatusMsg 853
 TApXScript 146
 TAsciiEOLTranslation 527, 529
 TAutoScaleMode 631, 657
 TBlockCheckMethod 530
 TCallInfo 415
 TDeleteFailed 584
 TDeviceLayer 58, 109
 TDialError 841, 849
 TDialErrorEvent 849
 TDialingCondition 841, 847, 849, 850
 TDialingStatus 841, 850
 TDialStatusEvent 850
 Terminal 163, 261
 TerminalState 177
 Terminate 185
 TFaxAcceptEvent 761, 797
 TFaxClass 721, 734, 789, 807
 TFaxClassSet 807
 TFaxCloseFileEvent 614
 TFaxCvtOptions 619
 TFaxErrorEvent 728
 TFaxFinishEvent 728
 TFaxFont 610
 TFaxInputDocumentType 611
 TFaxLogCode 729, 799, 829
 TFaxLogEvent 729, 799
 TFaxNameEvent 762, 800
 TFaxNameMode 760, 790
 TFaxNextEvent 751
 TFaxOpenFileEvent 615
 TFaxOutputLineEvent 616
 TFaxPLCode 684, 696
 TFaxPLEvent 684
 TFaxPrintProgress 684, 686
 TFaxPrintScale 687
 TFaxPrintStatusEvent 684
 TFaxPrnNextPageEvent 685
 TFaxReadLineEvent 617
 TFaxRecipientRec 770, 784, 820
 TFaxResolution 621, 632, 682
 TFaxServerFatalErrorEvent 798
 TFaxServerFinishEvent 799
 TFaxServerMode 798, 799
 TFaxServerPortOpenCloseEvent 800
 TFaxServerStatusEvent 800
 TFaxStatusEvent 617, 729
 TFaxWidth 623, 633, 682
 TFlowControlState 59
 TFtpErrorEvent 568
 TFtpLogCode 569
 TFtpLogEvent 569
 TFtpReplyEvent 569
 TFtpStatusCode 568, 569
 TFtpStatusEvent 569
 tfXxx 317
 THWFlowOptions 62
 Ticks2Secs 893

Timeout 144
 TimerExpired 893
 TimeStamp 874
 TimeStampStr 874
 TLightSet 191
 TLineDialParams 415
 tlXxx 35, 45, 66, 94
 TModemString 746, 787, 796, 846, 849
 tmXxx 92
 ToneDial 753, 807, 853
 TooFewDigits 354
 TooManyDigits 354
 TopMargin 623
 TotalErrors 553
 TotalFaxPages 688
 TotalPages 735, 808
 TPacketEndSet 139
 TPacketStartCond 143
 TParity 82
 TProtocolAcceptEvent 545
 TProtocolErrorEvent 545
 TProtocolFinishEvent 546
 TProtocolLogEvent 546
 TProtocolNextFileEvent 547
 TProtocolStatusEvent 548
 TProtocolType 550, 897
 TraceHex 93
 TraceName 93
 TraceSize 94
 Tracing 94
 TransferTimeout 576
 TranslateAddress 437
 TransmitTimeout 553
 TrimSeconds 437
 TStationID 730, 733, 802, 805
 TStringPacketNotifyEvent 142
 TSWFlowOptions 91
 tsXxx 315, 435
 TTapiLogCode 446
 TTapiLogEvent 424
 TTapiMode 92
 TTapiState 435
 TTapiStatusEvent 425
 TTAPStatus 842, 851
 TTAPStatusEvent 851
 TTraceLogState 66, 94
 TTriggerAvailEvent 71
 TTriggerDataEvent 73
 TTriggerEvent 69
 TTriggerLineErrorEvent 73
 TTriggerStatusEvent 78
 TTriggerTimerEvent 79
 TTSOptions 350
 TUnpackerOptions 637
 TUnpackOutputLineEvent 635
 TUnpackStatusEvent 636
 TurnDelay 554
 TViewerErrorEvent 665
 TViewerFileDropEvent 664
 TViewerRotation 668
 TWriteFailAction 555
 TWsAcceptEvent 109
 TWsErrorEvent 111
 TWsNotifyEvent 126, 127
 TWsSocketErrorEvent 127
 TXDLight 191
 TXDOffTimeout 194
 TZmodemFileOptions 556

U

UnpackFile 639
 UnpackFileToBitmap 640
 UnpackFileToBmp 641
 UnpackFileToDcx 641
 UnpackFileToPcx 642
 UnpackFileToTiff 642
 UnpackPage 643
 UnpackPageToBitmap 644
 UnpackPageToBmp 644
 UnpackPageToDcx 645
 UnpackPageToPcx 645
 UnpackPageToTiff 646

Unrecognized 355
 uoXxx 637
 UppcaseFileNames 554
 UpdateDisplay 442, 481, 581, 692, 825
 UpdateLog 446, 585, 696, 829, 864
 UpdateStatus 814
 UseAutoWrap 222
 UseEscapes 854
 UseEventWord 95
 UseLazyDisplay 289
 UseNewLineMode 222
 UserCode 105
 UserLoggedIn 576
 UserName 386, 577
 UseScrollRegion 223
 UseSoundCard 438
 UseTapi 855
 UseWideChars 223

V

VER_PLATFORM_WIN32_NT 385
 VER_PLATFORM_WIN32_WINDOWS 385
 Version 8
 VertDiv 646, 670
 VertMult 646, 671
 VertScroll 671
 VideoInDevice 307
 VideoOutDevice 307
 VoIPAvailable 307
 vrXxx 668

W

WaitingForCall 308
 WantAllKeys 290
 WaveFileName 438

WaveState 439
 waXxx 426
 wfXxx 555
 Where 355
 Where2 355
 WhitespaceCompression 647, 671
 WhitespaceFrom 647, 672
 WhitespaceTo 648, 673
 Width 623
 WordList 350
 WrapAround 268
 WriteChar 223, 291
 WriteFailAction 555
 WriteSocket 129
 WriteString 223, 291
 WsAddress 112
 WsMode 113
 WsPort 114
 WsTelnet 114
 WsVersion 129
 wsXxx 439

X

XOffChar 97
 XOnChar 97
 XYmodemBlockWait 555

Z

zfoXxx 510, 556
 Zmodem8K 556
 ZmodemFileOption 556
 ZmodemFinishRetry 557
 ZmodemOptionOverride 557
 ZmodemRecover 558
 ZmodemSkipNoFile 558

Subject Index

A

- AcceptFile processing 495
- accepting connection 126
- accessing terminal parser 260
- adding
 - commands 162
 - data trigger 49
 - keyboard mapping instance 241
 - recipient to job file 770
- AddModemRecord 457
- AdLibMdm 447
- AdMdm 447
- AdMdmDlg 447
- Alphanumeric pager *See* Pager
- ANSI, definition of 921
- answering
 - call 464, 759
 - data calls 759, 762
- APF file format 591
- APJ *See* Async Professional Job File
- APRO.HLP 4
- APRO.XXX 4
- APROBCB.HLP 4
- ASCII protocol
 - See also* protocol
 - delay settings 527, 529
 - end of file settings 529
 - end of line character 528
 - end of line translation 519, 527, 529
 - overview 519
 - timeout settings 528
- ASCII text documents 594
- assigning pager port 858
- Async Professional Job File (APJ) 766
- asynchronous serial communication 921
- AT commands 921
- AWDEFINE.INC 920

B

- batch file transfer 494
- baud rate
 - defined 921
 - setting 53
- Bell 103 921
- Bell 212A 921
- Berkley Sockets API 99
- blinking text, painting 256
- block size control 511
- BMP files 595
- bps 921
- break 922
- break signal 87

C

- cancelling
 - fax session 782
 - pager call 845
 - protocol transfer 533
 - script 157
 - state machine 173
- CCITT 922
- character map
 - adding new instance 248
 - clearing 249
 - count 249
 - data source 246
 - emulator and 256
 - loading from file 250
 - loading from resource 252
 - overview 245
 - storing to file 252
- character quoting 513
- character set
 - glyph 202

- mapping 199
- special graphics 199
- USASCII 199
- character-time 922
- checking
 - CTS signal 55
 - DCD signal 56
 - DSR signal 58
 - DTR signal 59
 - for received data 95
 - for syntax errors 161
- checksum 922
- client
 - connecting to server 109
 - defined 922
- clock tick 888
- collecting data 138, 748
- COMM.DRV 922
- command
 - list of 162
 - script 162
- common problems 932
- component hierarchy 8
- comport
 - activating triggers 88, 90
 - adding triggers 49, 50, 51
 - baud rate 53
 - buffer space 64
 - character ready 54
 - CTS signal 55
 - data bits 56
 - DCD signal 56
 - defined 922
 - DeltaCTS signal 56
 - DeltaDCD signal 57
 - DeltaDSR signal 57
 - DeltaRI signal 57
 - determining number of stop bits 91
 - device layer 49
 - device type 58
 - DSR signal 58
 - DTR signal 59
 - flow control 54, 59, 62, 91, 97
 - flushing buffer 60
 - input buffer space 63, 64
 - line break 64
 - line errors 65
 - modem status byte 68
 - name 895
 - number 83
 - opening 52, 80
 - output buffer space 80, 81, 82
 - overview 22
 - parity 82
 - port number 55
 - processing messages 82
 - reading data 61, 62
 - removing triggers 85
 - RI signal 86
 - RS-485 support 86
 - RTS signal 86
 - sending data 81, 84
 - XOff character 97
 - XOn character 97
- conditional defines 920
- connecting
 - socket 126
 - TAPI 422
- control character
 - escaping 508
 - pager 854
- control sequence
 - terminal buffer 202
 - terminal parser 233
 - VT100 terminal 265
- conversion
 - closing file event 614
 - closing image files 604
 - compressing raster line 604
 - converting bitmap 607
 - converting image files 605, 607
 - file extension 608
 - generating end-of-page code 614
 - lines per page 613

- opening file 619
- opening image file 615
- outputting line 616
- reading line 617
- reading raster line 610
- setting options 619
- specifying fax width 623
- specifying font file name 609
- specifying font size 610
- specifying input image file name 608
- specifying input image file type 611
- specifying left margin width 613
- specifying output fax file name 620
- specifying resolution 621
- specifying tab size 622
- specifying top margin size 623
- station ID 621
- status 617
- cover pages 738
- CRC 922
- creating fax job file 775
- CTS signal
 - checking 55
 - defined 923
- cursor
 - displayed during waits 658
 - movement sequences 265
 - moving left to tab stop 209
 - moving right 210
 - moving to new line 222
 - retrieving position 257
 - status 216
 - terminal column 206

D

- data
 - adding trigger 49
 - answering calls 759
 - bits 923
 - checking for received 95

- collecting 138, 748
- compression 923
- ownership 134
- reading 127
- transfer rate 526
- data packet
 - availability 141, 142
 - case sensitivity 140
 - collecting data 138, 748
 - data ownership 134
 - defined 132
 - determining port 138
 - determining size 142
 - end condition 135
 - end string 140
 - example 136
 - including start and end strings 140
 - receiving 139
 - re-enabling 138
 - start condition 134
 - starting data collection 143
 - starting string 143
 - timeout 141, 144
 - wild-card characters 144
- DCB 923
- DCD signal 923
- DCD signal, checking 56
- DCE 923
- DCX files 595
- debugging
 - dispatch logging 36
 - overview 33
 - tracing 33
 - Windows communications programs 930
- DEC VT100 standard 195
- defining
 - state bitmap 182
 - state condition 182
 - state string 185
 - terminal component 261
 - terminal display status 259
- DeltaCTS signal 56

- DeltaDCD signal 57
 - DeltaDSR signal 57
 - DeltaRI signal 57
 - demonstration programs 6
 - deprecated components 3
 - destination directory 533
 - detecting
 - active protocol 540
 - batch transfers 530
 - dial tone 743, 845
 - DTMF tone 423
 - determining
 - data packet size 142
 - default fax font 788
 - emulator display width 266
 - fax error action 720
 - fax font 747, 787
 - fax header font 788
 - fax job file directory 811
 - fax port used 783
 - fax send count 750
 - fax serial port 718
 - logging state 66
 - media terminal device to use 307
 - modem serial port 466
 - number of dial attempts 786
 - number of glyph cells 183
 - number of mappings 241
 - number of rings 758, 781
 - number of stop bits 91
 - port 138
 - protocol log file name 584, 585
 - script protocol 162
 - serial port 173
 - state color when not active 183
 - state component color 181
 - terminal component 163
 - whether state is active 181
 - device layers
 - defined 923
 - setting 58, 109
 - dial status of pager 850
 - dialing 745, 747
 - dialing error 849
 - Dialup Networking 10, 371
 - disconnecting
 - pager 848
 - socket 126
 - Winsock 111
 - dispatch logging 36
 - dispatcher, processing messages 82
 - displaying
 - data 157
 - script 159
 - DNS 99, 923
 - document conversion 588
 - Domain Name Service 99
 - dot notation 99, 923
 - drag and drop 651
 - DSR signal 923
 - DSR signal, checking 58
 - DTE 923
 - DTMF
 - detecting tone 423
 - send tone 430
 - sending tone 429
 - DTR signal
 - checking 59
 - defined 924
 - dumb terminal 927
- ## E
- EAPDException 900
 - editing commands 162
 - emulator
 - accessing parser 260
 - auto repeat 266
 - character set map 256
 - cursor position 257, 265, 267
 - defined 927
 - defining terminal component 261
 - defining terminal display status 259

- determining display width 266
- extensibility 196
- keyboard 195, 196
- overview 253
- painting blinking text 256
- processing key down message 258
- processing key press message 258
- processing lazy paint request 259
- processing paint request 260
- repainting 259
- retrieving keyboard map 257
- retrieving terminal buffer 256
- terminal 195
- TTY nil properties 262
- TTY overview 253
- VT100 escape sequence responses 264
- VT100 graphics processor option 266
- VT100 LED state 267
- VT100 line feed character 267
- VT100 numeric keypad sequence 265
- VT100 overview 253, 263
- VT100 scanlines 266
- VT100 smooth scroll 268
- VT100 standard 196
- VT100 support 195
- VT100 terminal control sequences 265
- VT100 text display 268
- VT100 wordwrap 268
- writing a terminal component 199
- encountering user function 161
- end of page code, generating 614
- error
 - codes 900
 - correction 924
 - exceptions 900
 - fax 728
 - handling in protocol transfers 488
 - message string 895
 - protocol code 549
 - Winsock 111
- example programs 6
- exceptions 900

- executing script 158, 163
- extension, image file 608

F

- FAQs 932

- fax

- See also* fax server, fax server manager, fax
 - job handler, fax engine
- accepting 761
- answering data calls 759, 762
- assigning file extension 721
- bps 719, 725, 730
- cancelling 717
- classes supported 734
- combining files 744
- connecting files 748
- converting font 609
- cover file name 744
- current page number 719
- detecting busy signal 745
- detecting dial tone 743
- determining number of rings 758
- determining send count 750
- determining error action 720
- determining font 747
- determining serial port 718
- dial retry 746
- dialing 745, 746, 747
- directory 759
- error 728
- file format 591
- file name 721, 760, 789
- hangup code 722
- header line 748, 816
- indicating class 721
- initialization baud rate 724
- initialization string 726
- initializing modem 760
- log 729
- log component 829

- log file name 829
- logging component 722
- manual receive 762
- manual transmit 752
- modem chip 726
- modem model 727
- modem revision 727
- next fax 751
- no connect 717
- normal baud 727
- number of pages 735
- one fax receive 761
- output buffer 743
- output file name 835
- page length 730
- print job end 835
- print job name 835
- print job start 836
- printer driver 830
- progress 722
- receiving 698, 763
- receiving status 758
- recipient name 749, 817
- remote ID 730
- resolution 731
- retrieving file name 762
- sender name 749
- sending 698
- server status 800
- session end 728
- software flow control 732
- specifying phone number 752
- station ID 733
- status 729
- status interval 734
- status message 734
- status window 733
- supporting error correction 726, 731
- title 750, 817
- tone dialing 753
- transferring bytes 717
- transmitting 753
- updating log 829
- using error control 720
- width 732
- yielding 752
- fax client
 - See also* fax server, fax server manager, fax job handler, fax engine, fax
- fax engine
 - See also* fax server, fax client, fax server manager, fax job handler, fax
 - automatic queries 802
 - bps 785, 795, 803
 - bytes transferred 782
 - class 789
 - classes supported 807
 - current page 784
 - determining number of dial attempts 786
 - determining number of rings 781
 - determining port used 783
 - dialing 781, 786, 787
 - directory 786
 - error response 788
 - fatal error 798
 - fax end 799
 - fax log 799
 - fax name 800
 - file extension 789
 - file name 790
 - initialization baud rate 793
 - initialization string 796
 - job name 783
 - job number 784
 - logging component 790
 - minimum delay 785
 - modem model 796
 - modem revision 796
 - monitoring calls 797
 - normal baud 797
 - opening/closing port 800
 - output buffer 781
 - print on receive 801
 - printer 790

- query interval 802
- receiving status events 783
- recipient information 784
- remote ID 802
- scheduling fax jobs 791
- server manager 803
- software flow control 805
- station ID 805
- status interval 806
- status window 806
- TAPI device 807
- tone dialing 807
- yield 794, 802
- fax job handler
 - See also* fax server, fax client, fax server manager, fax engine, fax adding recipient 770
 - concatenate APF files 771
 - creating job file 775
 - extracting APF data 772
 - extracting cover file 773
 - information dialog 778
 - overview 769
 - recipient record 774
 - retrieving job header record 773
- fax server
 - See also* fax client, fax server manager, fax job handler, fax engine, fax cancelling session 782
 - configuration 765
 - determining default font 788
 - determining font 787
 - determining header font 788
 - dialing 787
 - error control 785
 - error correction 803
 - hangup code 791
 - integration with other components 768
 - job header record 766
 - modem chip 795
 - number of pages 808
 - overview 764, 779
 - page length 801
 - page width 804
 - process 764
 - progress 791
 - recipient record 767
 - resolution 804
 - status 800
 - status message 806
 - supporting error correction 795
- fax server manager
 - See also* fax server, fax client, fax job handler, fax engine, fax determining directory 811
 - filter files 811
 - next time 811
 - pausing queueing 813
 - queueing 813
 - retrieving next fax 810
 - retrieving next job 810
 - updating status 814
- fax status
 - creating form 824
 - destroying form 824
 - fax component 824
 - form 824
 - updating form 825
- FIFO mode 924
- file
 - assigning extension 721
 - font 609
 - script commands 162
- File Transfer Protocol *See* FTP
- file transfers 526
- files supplied 4
- finishing print job 835
- flow control
 - buffer level 54
 - current state 59
 - defined 924
 - hardware options 62
 - software options 91
 - XOff character 97

- XOn character 97
- font
 - file 609
 - size 610
 - specifying size 610
- footers 674
- FTP
 - account information 562
 - active session 576
 - bytes transferred 562
 - changing directory 563
 - client 559
 - client support 521
 - closing port 568
 - commands supported 566
 - connecting 563
 - controlling connection 521
 - deleting file 564
 - error codes 522
 - file format 565
 - file length 564
 - help 566
 - list of files 566
 - log auditing 569
 - log in 567
 - log out 568
 - logging component 565
 - making new directory 568
 - opening port 567
 - operation in progress 566
 - overview 521
 - password 571
 - protocol error 568
 - renaming file 571
 - restarting byte location 572
 - resuming start 572
 - retrieving file 572
 - returning a reply 569
 - sending command 573
 - sending file 575
 - server address 574
 - server status 574

- status 569
- status codes 570
- store mode 575
- terminating transfer 562
- timeout value 576
- user name 577
- full duplex 924

G

- generating end of page code 614
- glossary 921
- glyph, in character set 202

H

- half duplex 924
- handshaking 499, 924
- headers 674
- host name 924

I

- image file, opening 615
- including start and end strings 140
- initializing modem 760
- installation 4
- interface layer 924
- Internet communications 99
- IP address 99, 924
- IRQ 925
- ISDN
 - overview 29
 - support 30

K

- Kermit protocol
 - See also* protocol
 - character quoting 513

- long blocks 541
- long packets 517
- overview 513
- packet length 541
- packet padding 542
- prefix characters 540, 542
- protocol options 515
- sliding windows 518, 541, 543, 544
- terminator character 543
- timeout settings 544
- keyboard emulator 196
- keyboard map
 - adding instance 241
 - clearing 241
 - custom 238
 - determining number of 241
 - extensibility 198
 - loading 198
 - loading from file 239, 242
 - loading from resource 239, 243
 - overview 238
 - retrieving 257
 - sequence definition 238
 - storing 244
 - storing as resource 198
 - value 242
 - virtual keys 238
- KnowledgeBase 19

L

- LAP M 925
- large block support 512
- lazy paint, processing request 259
- left margin, specifying width 613
- libmodem 448
- line
 - break signal 64
 - error 65, 925
 - number per page 613
 - outputting 616

- parameters 53, 56, 82, 91
- reading 617
- line-draw characters 202
- log, updating 829
- logging
 - buffer size 67
 - control of 66
 - determining state 66
 - fax 799
 - file format 37
 - file name 67
 - hex format 67
 - overview 36
 - protocol 493
- long packet support 517
- lookup 925

M

- margin, fixed left 613
- media terminal
 - audio input 300, 302
 - audio output 302
 - available devices 302
 - class 299
 - connecting 304
 - connection established 304
 - details about call 303
 - detecting incoming call 306
 - determining control 307
 - determining device to use 307
 - direction 300
 - disconnecting 305
 - displaying device dialog 306
 - establishing VoIP connection 304
 - instantiated 300
 - name 300
 - supporting Voice over IP 307
 - terminating call 303
 - using 299
 - VoIP fail 305

- waiting for incoming call 308
 - memory bitmaps 625
 - messages
 - APW_TRIGGERAVAIL 26, 70
 - APW_TRIGGERDATA 70
 - APW_TRIGGERSTATUS 26, 70
 - APW_TRIGGERTIMER 70
 - Microcom 925
 - MNP 925
 - modem
 - adding definition 457
 - adding record 457
 - answer attempt 473
 - answering 464
 - answering call 464
 - bps rate 465
 - cancelling connect attempt 465
 - changing state component 474
 - configuration structure 468, 477
 - configuring 466
 - connection parameters 470
 - creating detail file 457
 - currently selected 476
 - defined 925
 - defining location 459
 - deleteing record from index 458
 - deleting detail record 458
 - detecting caller ID information 471
 - detecting failure 473
 - detecting number of rings 474
 - determining serial port 466
 - dial attempt 473
 - dialing 467
 - directory 469
 - disconnecting 472
 - displaying selection dialog 475
 - displaying selection dialog box 459
 - error code 469
 - establishing connection 472
 - failure code 468
 - failure code message 468
 - opening 466
 - resetting answer attempt 475
 - retrieving all from detail file 459
 - retrieving from detail file 458
 - selection 461
 - sending command 476
 - state 469
 - status dialog box 477
 - time out 467
 - modem status
 - byte 68
 - creating dialog box 480
 - determining caption 480
 - determining form to display 481
 - updating 481
 - modemcap 448
- ## N
- naming conventions 18
 - network address 112
 - network shared-modem pool 925
 - newsgroups 19
 - NextFile processing 494
- ## O
- on-line help 18
 - opening image file 615
 - option
 - for conversion 619
 - setting 619
 - output
 - line 616
 - request trigger 26, 77
 - unpacked line 635
- ## P
- packet
 - end condition 135

- start condition 134
- pager
 - assigning port 858
 - cancelling call 845
 - choosing port 855
 - connecting 859
 - connection failure 845
 - control characters 854
 - creating text 840, 848
 - detecting dial tone 845
 - dial prefix 846
 - dial retry 847
 - dial status 847, 850
 - dialing 846, 852
 - dialing error 849
 - dialing error handling 841
 - dialing events 841
 - dialing options 853
 - disconnecting 848
 - error 858
 - error action 848
 - error values 841
 - fatal error 861
 - identification string 840
 - initializing modem 849
 - initializing serial port 852
 - initializing server event 857
 - log 851
 - log on 857
 - log out 857
 - login success indicator 860
 - message length 848
 - multi-line request 859
 - non-fatal error 860
 - paging status 842
 - phone number 851
 - phone number prefix 846
 - port properties 852
 - processing command 861
 - requirements 838
 - resending message 852
 - send 840
 - sending 852, 859
 - server access 851
 - server log out 860
 - server response delay 857
 - status 851
 - status message 853
 - successful response 858
 - terminating connection 850
 - tone dial 853
 - using TAPI 855
 - waiting for connection 847
- pager log
 - component 864
 - name 864
 - updating 864
- paint, processing request 260
- painting blinking text 256
- parity 925
- parser *See* terminal
- PCX files 595
- phone number
 - pager 851
 - pager prefix 846
 - RAS 384
 - TAPI 421
- port
 - closing 424
 - determining 138
 - establishing a network connection 114
 - opening 425
- print job
 - finishing 835
 - starting 836
- printer
 - cancelling 685
 - caption 680
 - current page 680
 - fax width 682
 - file name 682
 - first page 683
 - footer 680
 - header 681

- last page 683
- logging component 681
- multi page 683
- next page 685
- number of pages 688
- options 687
- print fax 686
- print log event 684
- print status 684
- progress 686
- resolution 682
- scaling 687
- status 687
- printer driver
 - defined 830
 - installation 830
 - recompiling 833
- printer log
 - fax component 696
 - file name 696
 - updating 696
- printer status
 - creating form 691
 - destroying form 691
 - printer component 691
 - referencing form 691
 - updating 692
- processing
 - key down message 258
 - key press message 258
 - messages 82
 - pager command 861
 - paint request 260
- protocol
 - See also* ASCII protocol, Kermit protocol, Xmodem protocol, Ymodem protocol, Zmodem protocol
 - aborting 487
 - aborting on carrier drop 488, 526
 - accept file 486, 545
 - accepting file event 545
 - ASCII overview 519
 - background operation 485
 - batch file transfer 494
 - block check method 530
 - block length 531
 - block number 531
 - buffer size 485
 - bytes remaining 532
 - bytes transferred 532
 - cancelling transfer 533
 - comport 533
 - data transfer rate 526
 - defined 926
 - destination directory 533
 - detecting active 540
 - detecting batch transfers 530
 - determining 162
 - elapsed time 534
 - error code 549
 - error count 531, 553
 - error handling 488
 - estimating transfer time 526, 534, 543, 548, 554
 - events 486
 - fatal error 487, 545
 - file date and time 535
 - file length 536
 - files to send 536, 539
 - initial file offset 539
 - internal logic 496
 - Kermit overview 513
 - list of files to send 495
 - logging 487, 493, 546
 - logging component 549
 - name string 897
 - next file 487, 547
 - overhead per block 548
 - overwriting files 555, 556
 - partial files 584
 - receive files 551
 - received file name 536, 538, 539
 - reject file 495
 - retry settings 538

- RTS low during write 550
- sending files 551
- state code 492, 549
- state string 553
- status 487, 489, 548
- status event 548
- status interval 552
- status properties 490
- status window component 492, 552
- timeout settings 537, 553, 554
- transfer complete 487, 546
- transfer complete event 546
- turnaround delay 554
- type setting 550
- Xmodem overview 501
- Ymodem overview 504
- Zmodem overview 507
- protocol log
 - See also* protocol
 - determining file name 584, 585
 - file format 583
 - overview 583
 - updating file 585
- protocol status
 - See also* protocol
 - abstract status class 578
 - creating form 580
 - destroying form 580
 - form instance 580
 - protocol component 580
 - standard component 582
 - updating form 581
- proxy server
 - address 104, 105
 - password 104
 - port number 104
 - specifying user name or code 105
 - version 104

R

RAS

- active connections 380
- asynchronous dialing 377
- authenticate access 383, 386
- authentication 378
- callback number 374
- compression mode 374
- connecting 382
- connection handle 374
- connection status 374
- connection status dialog 381
- connection status text 379
- create phonebook entry 375
- delete phonebook entry 375
- device name 375
- device type 376
- dial 376
- dial dialog 377
- dial error 382
- dial mode 377
- dial options 378
- dial status 383
- dialer overview 372
- dialing parameters 379, 385
- disconnecting 383
- domain 378
- edit phonebook entry 378
- entry name 378
- entry names 381
- error text 379
- establish connection 376
- hangup 380
- override phonebook entry 384
- overview 371
- password 383
- phone number 384
- phonebook dialog 384
- phonebook path 384
- platform supported 385
- release resources 380

- speaker mode 385
- status component 387
- status dialog 386
- synchronous dialing 377
- terminate connection 380
- update dialing parameters 385
- user name 386
- Windows NT dial options 378
- WinNT connection 377
- raster line, reading 610
- README.TXT 4
- received data, checking for 95
- receiving
 - data 157
 - data packet 139
 - fax 698, 763
 - file 551
- references 115
- register components 6
- Remote Access Service *See* RAS
- remote device 926
- resending pager message 852
- resolution, specifying 621
- retrieving
 - cursor position 257
 - fax file name 762
 - job header record 773
 - keyboard map 257
 - next fax job 810
 - terminal buffer 256
- RI signal 86
- ring indicator (RI) 926
- rotation 651
- RS-232 926
- RS-485
 - base address 53
 - overview 31
 - RTS control 32
- RTS signal
 - checking 86
 - defined 926

S

- SAPI
 - components 309
 - overview 310
- scaling
 - fax printer 675
 - fax unpacker 626
 - fax viewer 649
- script
 - active 158
 - adding commands 162
 - cancelling 157
 - checking for syntax errors 161
 - command finish event 158
 - command start event 159
 - commands 146, 162
 - comport 157
 - debugging 154
 - defined 145
 - definition 145
 - determining protocol 162
 - determining terminal component 163
 - display event 159
 - displaying data 157
 - editing commands 162
 - encountering user function 161
 - error handling 160
 - example 155
 - executing 153, 158, 163
 - file 162
 - list of commands 162
 - options 149
 - parsing variable 160
 - preparing 161
 - protocol component 154
 - receiving data 157
 - required components 154
 - sending data 157
 - starting 163
 - syntax 146
 - syntax error handling 161

- terminal component 154
- tracking progress 158, 159
- user functions 152
- user variables 153
- sending
 - data 81, 84, 157
 - DTMF tone 429
 - file 551
 - page 859
- serial data 926
- server 99, 926
- setting
 - device layers 58, 109
 - options 619
- Simple Network Paging Protocol *See* Pager
- Sliding Windows Control 518
- smart terminal 927
- SNPP server error 858
- socket
 - accepting connection 126
 - connecting 126
 - defined 927
 - disconnecting 126
 - error 127
 - overview 100
 - reading data 127
 - writing to 127
- specifying
 - fax phone number 752
 - font file name 609
 - font size 610
 - input image file name 608
 - input image file type 611
 - left margin width 613
 - output fax file name 620
 - resolution 621
 - tab size 622
 - top margin size 623
- speech
 - considerations for recognition 312
 - considerations for synthesis 312
 - recognition 311
 - requirements for recognition 312
 - requirements for synthesis 312
 - synthesis 310
 - synthesis tags 334
- S-registers 926
- start bit 927
- starting
 - data collection 143
 - data packet string 143
 - print job 836
 - script 159
- state
 - activating 184
 - defining bitmap 182
 - defining condition 182
 - defining string 185
 - determining color when not active 183
 - determining component color 181
 - determining number of glyph cells 183
 - determining whether active 181
 - satisfying conditions 184
 - terminating 185
- state machine
 - activating 176
 - cancelling 173
 - changing 175
 - collected data 174
 - component name 177
 - currently active 174
 - data size 174
 - determining serial port 173
 - error code 175
 - pointer 174
 - starting 176
 - terminating 176, 177
- station ID 621
- status
 - activating trigger 88
 - adding trigger 50
 - conversion operation 617
 - creating fax form 824
 - defining terminal display 259

- destroying fax form 824
- fax engine 806
- fax server 800
- FTP codes 570
- modem 68
- pager 851
- printer 687
- protocol 489
- RAS 386
- RAS dialing 383
- receiving fax 758
- TAPI 397
- TAPI dialing 425
- trigger 25
- unpack operation 636
- status lights
 - bitmaps 189
 - break light 191
 - CTS light 191
 - DCD light 191
 - DSR light 191
 - error light 191
 - lights to display 191, 193
 - lit color 189
 - lit state 189
 - overview 188
 - ring light 191
 - RXD light 191
 - serial port 193
 - start triggers 194
 - timeout settings 193, 194
 - TXD light 191
 - unlit state 190
- stop bits
 - defined 927
 - determining number 91
- streaming protocol 927
- success response 858
- SuperKermit 518
- supporting
 - error correction 795
 - fax error correction 726, 731

T

- tab, specifying size 622
- TAPI
 - answering calls 402
 - auto answer 410
 - bps rate 411
 - call info 415
 - caller ID 412
 - closing port 424
 - comport 413
 - configuration record 431
 - connect event 422
 - connect fail 423
 - connect fail event 423
 - connecting 422
 - create status display 441
 - destroy status display 441
 - determining an instance 735
 - device 441
 - device configuration 419, 430
 - device selection 429
 - dial 415, 416
 - dial attempts 409
 - dial retry 420
 - disconnect 413
 - disconnect check 413
 - display comports 431
 - DTMF 406
 - fax server device 807
 - log 424
 - log event 424
 - log file name 446
 - logging 400, 434, 446
 - logging overview 397
 - making calls 400
 - number of devices 415
 - opening port 425
 - overview 389
 - passthrough mode 414
 - phone number 421
 - port close event 424

- port open event 425
- property sheets 430
- retry wait 428
- rings 409
- service providers 404
- state 435
- status 425
- status display 397
- status display form 441
- status event 425
- status message 436
- status processing 394
- status window 432, 433
- translate address 437
- unimodem 404
- unimodem/v 404, 406
- update status display 442
- version 409
- wave file 428
- wave file support 405
- wave message length 420
- wave record 432
- TAPI integration 389
- technical support 931
- Telelocator Alphanumeric Protocol *See* Pager
- Telephony Application Programming Interface 389
- Telnet 106, 114, 927
- terminal
 - accepting serial events 276
 - adding character map 248
 - adding keyboard mapping instance 241
 - background color 201, 205, 278
 - background/foreground color 215
 - blinking text cycle 278
 - buffer overview 197, 201
 - capturing data to file 278
 - character height 280
 - character mapping 249
 - character mapping overview 245
 - character set 201, 205, 280
 - character set mapping table 199
 - character sets 215
 - character width 281
 - clearing buffer 282
 - clearing character mapping 249
 - clearing display 281
 - clearing keyboard mapping 241
 - client window 271
 - clipboard 273
 - column count 207, 282
 - column origin 218
 - comport component 269
 - control sequences 195, 224
 - copying displayed text 273
 - copying to clipboard 283
 - creating 283
 - creating buffer 207
 - cursor position 202, 216, 217, 218, 219
 - custom keyboard mapping 238
 - data source 246
 - default color 207
 - defining column cursor 206
 - delay repaint 286
 - deleting character 208
 - deleting line 208
 - design considerations 196
 - destroying 284
 - determining number of mappings 241
 - display 201
 - display characters 201
 - display options 213
 - display status 216
 - display view 203
 - draw command 250
 - draw script 249
 - emulation 195
 - emulation mode 195
 - erasing character 211
 - erasing display 212
 - erasing line 211
 - escape sequences 224
 - forced repaint 286

- foreground color 201
- inserting line 216
- keyboard mapping 238
- keyboard mapping overview 238
- keyboard mapping return value 242
- keyboard mapping sequence definition 238
- line attributes 214
- loading character map from file 250
- loading keyboard mapping 198
- loading keyboard mapping from file 239
- moving cursor 220
- moving cursor left 209, 217
- moving cursor right 210
- moving cursor to new line 222
- overview 195
- parser class 198
- removing tab stops 205
- removing vertical tab stops 206
- repainting 202, 214
- resetting defaults 219
- row count 219
- row origin 218
- scrollback buffer size 222
- scrollback view 203
- scrolling region 203, 221
- scrolling region active 223
- setting default 208
- setting tab stops 221
- setting text options 220
- storing character map 252
- storing keyboard mapping 244
- teletype mode 195
- text 215
- text color 212
- text defaults 221
- text options 201
- VT100 parsing 198
- wordwrap 222
- writing character 223
- writing string 223
- terminal buffer
 - control sequences 202
 - retrieving 256
- terminal component
 - defining 261
 - determining 163
 - displaying local data 286
 - emulator 269, 284
 - file name 279
 - font handling 272
 - hook keystrokes 290
 - lazy write mode 272, 289
 - line 287
 - overview 269
 - proportional fonts 272
 - row count 287
 - scrollback buffer size 289
 - scrollback mode 288
 - serial device attached 283
 - text color 285
 - text options 277
 - text size 272
 - window handle 284
 - write character 291
 - write string 291
- terminal parser
 - accessing 260
 - ancestor class 224
 - ANSI escape sequences 225
 - ANSI mode 225
 - argument 230, 235
 - argument count 230
 - Arguments property 225
 - clearing 225, 231
 - control characters 226
 - control sequence 233
 - creating 231
 - current command 231
 - escape sequences 226
 - one-byte control characters 226, 236
 - operations 224
 - overview 224
 - process character 232

- process single character 224
- sequence 225
- unicode character 233
- VT100 225
- VT100 escape sequences 235
- VT52 mode 225, 236
- TIFF files 595
- timer
 - clock tick 888
 - converting units 893
 - delay 890
 - detecting expired 893
 - elapsed time 890, 891
 - overview 888
 - remaining time 892
 - starting 891, 892
- timer trigger 25, 51, 79, 90
- top margin, specifying size 623
- tracing
 - adding entry 52
 - buffer size 94
 - control of 94
 - defined 33
 - file format 34
 - file name 93
 - hex format 93
 - overview 33
- transmitting fax 753
- trigger
 - activating 88, 90
 - adding 49, 50
 - data availability 71
 - defined 927
 - general event 69
 - line error 73
 - line status change 78
 - modem status change 74
 - output buffer space 75
 - output buffer space event 76
 - output request 77
 - overview 22, 23
 - removing 85

- status 25
- timer 79
- TTY
 - nil properties 262
 - overview 253

U

- UART 927
- unpacking
 - all pages 639
 - auto scaling 631
 - file name 634
 - number of pages 635
 - options 637
 - output file name 638
 - output line event 635
 - page 643
 - resolution 632
 - scaling 633, 634, 638, 646
 - status 636
 - status event 636
 - to APF 631
 - to bitmap 640, 644
 - to BMP 641, 644
 - to DCX 641, 645
 - to PCX 642, 645
 - to TIF 646
 - to TIFF 642
 - white space compression 647, 648
 - width 633
- updating printer log 696
- using fax error control 720

V

- V.17 928
- V.21 928
- V.22 928
- V.22bis 928
- V.25bis 928

- V.27 928
- V.29 928
- V.32 928
- V.42 928
- variable parsing 160
- viewer
 - active page 656
 - auto scaling 657
 - background color 658
 - bitmaps 666
 - copying to clipboard 659
 - cursor 658
 - displaying first page 660
 - displaying last page 662
 - drag and drop 656
 - dropping file 664
 - error 665
 - file name 660
 - foreground color 659
 - loading entire fax 662
 - next page 663
 - number of pages 663
 - page change 665
 - page height 667
 - page width 667
 - previous page 667
 - rotation 668
 - scaling 661, 669, 670, 671
 - scrolling 661, 671
 - selection 669, 670
 - updating scaling properties 657, 659
 - white space compression 671, 672, 673
- voice telephony 311
- VT100 927
 - See also* terminal
 - ASCII value 202
 - escape sequence responses 264
 - parser 225
 - standard 196

W

- wave file
 - current state 439
 - interrupting 420
 - name 438
 - playing 427
 - sound card 438
 - status 426, 427
 - stop recording 434
 - stopping 434
 - supported 417
- wave, average relative amplitude 410
- white space compression
 - fax unpacker 628
 - fax viewer 651
- Windows printer driver 830
- Winsock
 - disconnecting 111
 - error 111
 - overview 99
- Winsock port
 - connecting to server 109
 - definition of 106
 - device type 109
 - disconnecting 111
 - error 111
 - establishing connection 110
 - mode 113
 - network address 112
 - opening 112
 - port number or name 114
 - telnet 114
- Winsock socket
 - attaching client to listening socket 117
 - bind 117
 - close 118
 - connecting 126
 - creating 119
 - disconnecting 126
 - DLL description 119
 - DLL loaded 118

- DLL status 129
- DLL version 119, 129
- error 127
- error code 120
- establishing network connection 118
- handle 119
- host byte order 125
- listening for connect 121
- local address 121
- local name 121
- lookup address 122
- lookup name 122
- lookup port 123
- lookup service 124
- network address 124, 128
- network byte order 120
- number of sockets 124
- read data 127
- read event 127
- send data 129
- send notification 128
- writing to 127

X

- Xmodem protocol
 - See also* protocol

- extensions 503
- overview 501
- timeout settings 555
- XOff character 97, 924
- XOn character 97, 924

Y

- Ymodem protocol
 - See also* protocol
 - extensions 506
 - overview 504

Z

- Zmodem protocol
 - See also* protocol
 - block size control 511
 - control character escaping 508
 - file management options 510
 - long blocks 512, 556
 - overview 507
 - overwriting files 556, 557, 558
 - protocol options 509
 - resume transfer 510, 558
 - retry settings 557



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



Try the full range of
TurboPower products.
Download free Trial-Run Editions
from our Web site.

www.turbopower.com

For over fifteen years you've depended on TurboPower to provide the best tools and libraries for your development tasks. Now try SysTools 3 and XMLPartner Professional—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.

SYSTOOLS 3™

SysTools 3 means never having to write the same old routines again. That's because it's the only library with more than 1000 reliable, optimized, time-tested routines you'll use in virtually every project you build. For everything from low-level system access to high-level financial calculations, SysTools is a product that will easily pay for itself the first time you use it.

XMLPARTNER PROFESSIONAL™



TurboPower's newest cross-platform toolkit for XML (supporting Linux as well as Windows) provides all the reading, writing, and editing power of our entry-level product, XMLPartner, then adds advanced manipulation, transformation, and presentation components—all in a single package! Our new XSL Processor with XPath support, filter set, and EXMLPro utility make it easy to add XML capabilities to your applications.

Async Professional 4 requires Microsoft Windows (9x, Me, NT, 2000 or XP) and Borland Delphi 3 and above, or C++ Builder 3 and above

©2001, TurboPower Software Company

 **TURBOPOWER®**
Software Company