TurboTCP

Version 2.0 • October 1994

Designed, coded, and documented by Eric Scouten

Note
This document was produced using Microsoft Word and the TrueType fonts Palatino, Helvetica, Courier, and Zapf Dingbats. It is 55 pages long. There are two versions of this document: `**README.word` which is a Microsoft Word 5 document, and `**README.standalone` which is a combined application/document created with Print2Pict. ❖

❖ **I M P O R T A N T**
TurboTCP is a limited shareware product. If you are using TurboTCP for profitable purposes, you may be obligated to pay a shareware fee. There is no shareware fee for using TurboTCP in freeware applications. Be sure to read the shareware notice on page 4 for more details. ❖

Contents

Getting Started

Chapter 1 Introduction to TurboTCP 1

Overview of TurboTCP 1
- System Requirements 2
- What's New 2
- Future Plans for TurboTCP 3

Shareware Notice 4
Redistribution of TurboTCP 4
Contacting the Author 5
Also by the Author 6
Acknowledgments 6

Chapter 2 Building a TurboTCP Application 7

Structure of a Typical TurboTCP Application 7
- Class Structure 7
- Flow of Control 9

Converting from TurboTCP 1.0 12
An Example: The MiniTelnet Application 13

Chapter 3 Class Library Reference: CTCPEndpoint 14

Introduction 14
Using CTCPEndpoint 14
Member Functions 15
Opening and Closing Sessions 16
Receiving Data 18
Sending Data 18
Configuration Methods 21
Selectors 23
State Change Notifications 24
Document and Window Naming 26
Error Handling 27
Notification Routines 27
Data Members 33
Configuration Fields 33

Chapter 4 Class Library Reference: UTurboTCP 34

Introduction 34
Member Functions 34
Opening/Closing TCP Services 34
Processing Network Events 35
IP Address Utilities 36

Chapter 5 TurboTCP Resources 37

Introduction 37
Alerts and Dialogs 37
Strings 40
Code Resources 40

Chapter 6 TurboTCP Internal Classes 41

Overview 41

Chapter 7 Optional Class Reference: CTCPApplication 42

Introduction 42
Using CTCPApplication 42
Background Application Operation 43
Member Functions 43
Application Shutdown 44
Network Event Processing 44

Chapter 8 CTCPSessionDoc 45

Introduction 45
Using CTCPSessionDoc 45
Member Functions 46
Closing Windows and Sessions 47
Window Titling 47

Chapter 9 References 48

Other TCP/IP Software Libraries 48
Programmer's Utilities 49
Networking Protocols 49
General TCP/IP RFCs 49
Protocol-Specific RFCs 50

Introduction to TurboTCP

Overview of TurboTCP

This document describes the TurboTCP class library, which integrates the Think Class Library with Apple's MacTCP driver. This library, now in its second major version, provides robust support for most TCP/IP applications on the Macintosh. Its major features include:

- **Truly asynchronous operation.** All MacTCP calls are made using asynchronous calls. TurboTCP does not use "polling loops;" instead, execution continues, freeing your application to perform other processing. TurboTCP provides callback routines to receive notification of I/O completion and asynchronous events; these events are relayed to your application code at event-loop time.
- **High-speed automatic data receiving mechanism.** TurboTCP can automatically issues one or more "no-copy" receive commands as needed so that MacTCP can constantly receive data for your application. On Ethernet-connected machines performance can reach MacTCP's limit of 400 KB/second.
- **Mix-in architecture (new to version 2.0).** The primary user-level TurboTCP class controlling interaction with MacTCP is designed as a C++ mix-in class. It can easily be merged with any of the subclasses of TCL's CDirector (such as CDocument or CDialogDirector).
- **Robust error recovery.** TurboTCP prevents attempts to dispose of MacTCP data structures prematurely. Such attempts are delayed until they can be completed safely. This operation is transparent to both programmer and user. Other MacTCP errors are handled in a manner consistent with the TCL error architecture.
- **Background friendly.** TurboTCP contains logic to allow any application to give time to other applications when running in the background. Performance is optimized for foreground operation.

This package **does not include** the Think Class Library itself, in either original or modified form. You must purchase a copy of Symantec C++ for Macintosh, version 7.0, in order to use the TCL and TurboTCP.

Note

Additional late-breaking information may be contained in a file named `*README.extra` in the root folder of the package. ❖

System Requirements

To make effective use of TurboTCP, you need the following items:

- Symantec C++ for Macintosh, version 7.0, and the Think Class Library version 2.0.x.
- At least 15MB disk space available on your hard drive.
- A Macintosh computer with a network connection which supports TCP protocols (ideally an Internet connection).
- MacTCP version 2.0.4.
- (Optional.) Metrowerks CodeWarrior Bronze, Silver, or Gold, DR/3 or newer. To use the CodeWarrior compilers, you will need a copy of my TCL 2.0 CodeWarrior Port Package, available on several FTP sites.

Applications built with TurboTCP will have the following minimal requirements:

- A Macintosh computer with a network connection which supports TCP protocols (ideally an Internet connection). If building with the CodeWarrior Silver or Gold compilers, PowerPC native execution is supported.
- Macintosh System 7.0 or newer.
- MacTCP version 2.0.4.

TurboTCP has not been tested with earlier versions of system software (6.0.x) or the MacTCP driver (1.1.x). There should be few compatibility issues.

Note

If these requirements do not fit your needs, you may which to the references chapter (at the end of this document) for pointers to other TCP/IP libraries for the Macintosh which are available as shareware and/or freeware. ❖

What's New

If you are **new to TurboTCP,** you should read all of Chapter 2 carefully to understand the design of the class library and its use in your applications.

If you have used previous versions of TurboTCP, please be aware of the following changes which were introduced in version 2.0:

- **Ported to TCL 2.0.x.** This is the most dramatic change in TurboTCP. Although TurboTCP's reliance on TCL classes is minimal, the library now requires the use of certain C++ constructs which are not available in the Think C compiler. (It is strongly discouraged to use TurboTCP with the TCL version 1.1.x.)
- **PowerPC native compilation.** TurboTCP has been extensively tested on a Power Macintosh. Issues relating to native callbacks and routine descriptors have been ironed out. (Please note that the testing has occurred **only** with the Metrowerks CodeWarrior compilers. I do not use the Symantec PowerPC Cross Development Kit.)

- **Modified inheritance tree.** Users of TurboTCP 1.0 often encountered difficulties with the old CTCPSessionDoc class – particularly in cases where a document was not an appropriate user-interface item. In version 2.0 a new class, CTCPEndpoint, was added. This is a mix-in class which provides most of the functionality of the old CTCPSessionDoc, but is not linked to any TCL class. CTCPSessionDoc remains in the class library; it is now multiply-inherited from CDocument and CTCPEndpoint.
- **New interface to CTCPStream and CTCPResolverCall.** In TurboTCP 1.0, CTCPStream and CTCPResolverCall passed notification to their "owners" by use of a CCollaborator link. In version 2.0 this link was removed. It is now required that you provide an object which is a descendant of CTCPEndpoint to receive notification of call completion and asynchronous events. CTCPStream and CTCPResolverCall access methods of CTCPEndpoint directly. This change was made to improve application performance. (Users of CTCPSessionDoc should see no difference in behavior.)
- **Better use of C++ access control.** Some of the classes of TurboTCP (CTCPDriver and CTCPAsyncCall, in particular) have been redefined as "internal classes." Access to all methods of these classes (including constructors and destructors) is now declared private, with certain TurboTCP classes named as friend classes.

Future Plans for TurboTCP

TurboTCP is a constantly evolving software library. The following are some ideas for enhancements I would like to make in future versions of the library. Please do not interpret this list as a guarantee of a future release, nor as any guarantee of a release date for a newer version.

- **Improved support for Telnet command-line protocols.** A new class, CTelnetCmdLine, may be added which will add support for parsing and responding to commands from such Telnet protocols as FTP, NNTP, and POP3. (Note that these will not be complete implementations of any such protocol, and emphasis will be given to protocol clients rather than servers.)
- **Support for Open Transport.** As the Open Transport specifications become available, I intend to build a version of TurboTCP which will have similar interfaces to the current version with explicit support for Open Transport.

Shareware Notice

The licensing agreement for TurboTCP 2.0 is driven by a single basic principle: If you profit from the use of TurboTCP, I should profit from it also. I welcome the use of TurboTCP by freeware authors and hobbyists constructing programs for their own use – for such purposes the library is provided free of charge.

A shareware fee of **US$35** applies in the following instances:

- You sell a product which includes TurboTCP and receive a gross income of US$250 or greater (over any time period) from sales of the product. Such products may include –but are not limited to– commercially-sold application programs and shareware programs. The shareware fee applies to any **single** product title, but includes rights to sell an unlimited number of copies of that title. (CD-ROM publishers: Please see the note about redistribution of TurboTCP.)
- You develop an application which includes TurboTCP for the exclusive use of a for-profit corporation. The shareware fee applies to an unlimited number of application generated for exclusive use of a single corporation.

In the following cases, TurboTCP is provided to you **free of charge:**

- You develop an application using TurboTCP for your own personal use, or for the use of a non-profit organization.
- You develop an application using TurboTCP which is distributed as freeware (i.e., distributed to the general public via the Internet at no charge).

Note
It is particularly expensive to accept foreign currency or foreign checks. If you are from outside the United States, please contact me to make special arrangements. ❖

Note
This shareware license agreement is different from the license agreement for version 1.0. If you have paid shareware fees under the 1.0 agreement, you are exempt from the new agreement. ❖

Redistribution of TurboTCP

I would like the TurboTCP library to be available to as many prospective software authors as possible. I periodically post library updates to the major FTP sites (mac.archive.umich.edu and sumex-aim.stanford.edu). The most recent public version can always be found on the TCL archive at **daemon.ncsa.uiuc.edu**. I encourage you to redistribute TurboTCP to local bulletin boards, campus file servers, and other FTP sites.

I must ask that you observe a few rules for redistributing TurboTCP:

- **Do not redistribute any version of TurboTCP that has been modified from the original release.**
- **Do not redistribute TurboTCP without this documentation file intact.**
- Neither you nor anyone else may sell or charge for the TurboTCP source library. Minimal download costs (including CompuServe) are acceptable.
- TurboTCP may be included on the Info-Mac CD, but not on any other CDs without my permission. In general, I am willing to grant permission for CD-ROM publishing with some compensation. Please contact me first!

Contacting the Author

I am interested to know how TurboTCP is being used. I invite you to contact me just to let me know what kinds of applications are being developed, and make suggestions for future versions. If you have questions about using TurboTCP, the TCL CodeWarrior Port Package, or about TCL 2.0 in general, you may contact me at the following addresses:

Internet	scouten@uiuc.edu
Telephone	+1 217 333 5214 (office) +1 217 344 1275 (home)
Fax	+1 815 836 3221
US mail	Eric Scouten 404 N Lake St Urbana, IL 61801

A mailing list has been set up for discussing TurboTCP issues. At present, it cannot support automatic subscription requests. If you wish to be added to the list, please contact Adam Treister <treister@forsythe.stanford.edu>.

Please note that this package is not a commercial product. Although I am generally available (and interested) to talk about this package and any related topic, I may from time to time be unable to respond to requests for support of the package due to other commitments.

Also by the Author

- **TCL 2.0 port package to CodeWarrior.** A set of patch files which makes it possible to compile the entire Think Class Library with the current Metrowerks CodeWarrior compilers for both 680x0 and PowerPC Macintosh. Incorporates a number of bug fixes and workarounds for unimplemented features (most notably C++ templates). Available for FTP on **daemon.ncsa.uiuc.edu**.
- **TCL 2.0.x bug list.** A listing of approximately 30 known bugs in the Think Class Library version 2.0.x including their status (unverified, acknowledged by Symantec, fixed in a more recent version). Posted regularly to the Usenet group **comp.sys.mac.oop.tcl** and available by FTP on **daemon.ncsa.uiuc.edu**.
- **Terminal pane.** A simple ASCII terminal emulator package for the TCL. Included in the TurboTCP distribution.

Acknowledgments

No Internet manual would be complete without a message of thanks to helpful people. TurboTCP was heavily influenced by a number of its early users. Brian Bezanson (MGI) and Paul Ferguson (Kaleida Labs) provided several months of detailed testing of the 1.0 version of TurboTCP and made many suggestions which have improved my programming style. Thanks also to Dan Strnad (Apple) and Jim Browne (NCSA) for providing special information and advice. And, of course, Peter Lewis and Jon Wätte have been constantly available for bouncing ideas about and have been just fun guys to know.

Structure of a Typical TurboTCP Application

This chapter serves as an introduction to the TurboTCP class library. It describes the basic structure of a typical application which makes use of TurboTCP, and the ways in which it takes advantage of the TCL.

In the following sections, you will be shown the basic class structure of the TurboTCP library and how messages are passed back and forth between the various objects in a running application.

It is important to note that TurboTCP provides essentially no user interface behaviors and is only minimally concerned with the TCL's chain of command. TurboTCP provides its own event queuing mechanism for responding to networking events.

Class Structure

Figure 2-1 (on the next page) illustrates the class structure of TurboTCP. The class structure is conceptually divided into three sets of classes: classes which are the internal core of TurboTCP, classes which interact with your user classes, and two optional helper classes.

The **core classes** (CTCPDriver, CTCPAsyncCall, CTCPStream, and CTCPResolverCall) implement the low-level interface with the MacTCP driver. These classes are reserved for the internal use of TurboTCP; you should not access them yourself. (In future versions of TurboTCP, the interfaces for these classes may need to be changed to support Apple's future TCP/IP technology.)

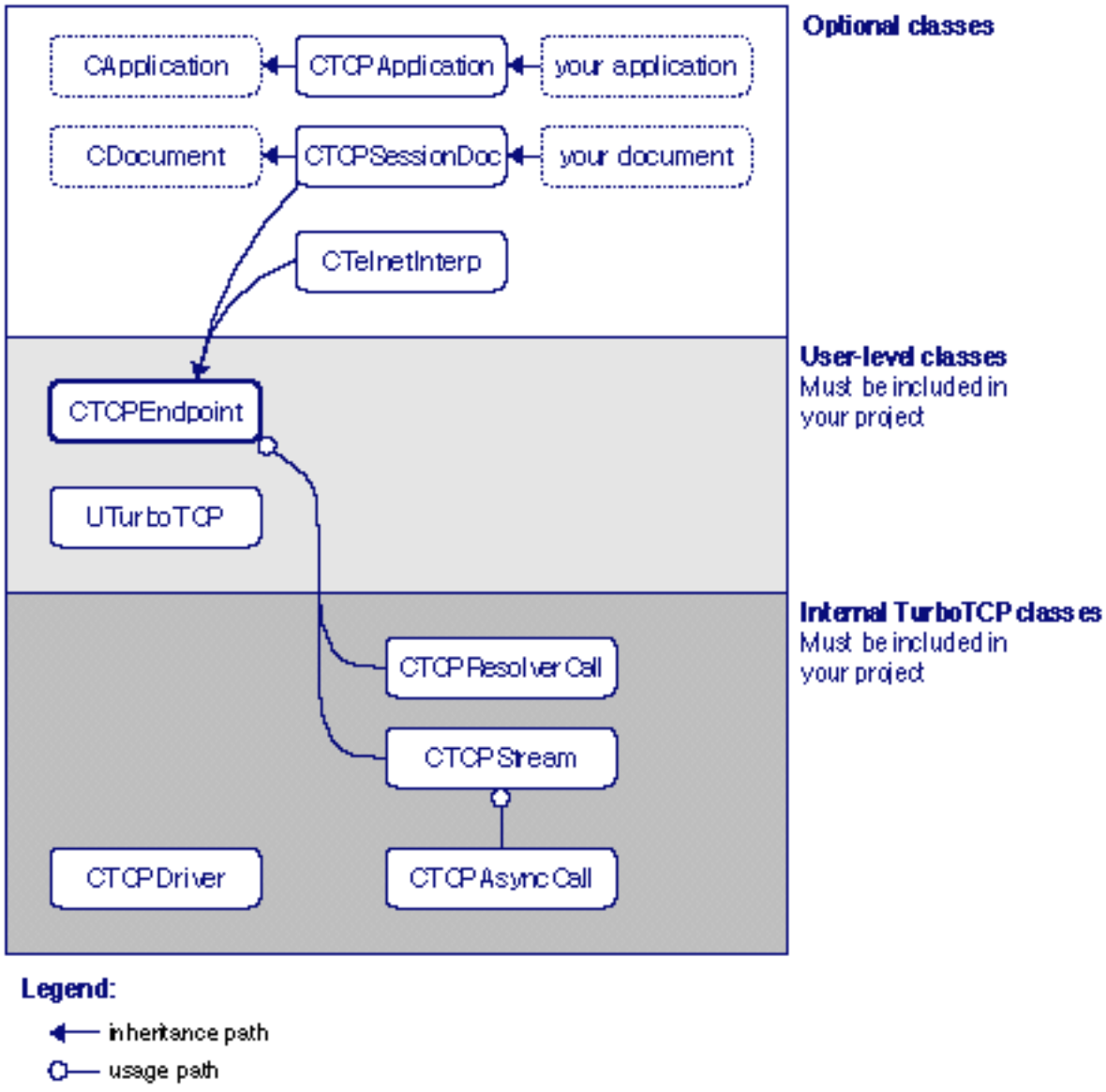
The **CTCPEndpoint** class is where your interaction as a programmer with TurboTCP begins. This mixin class handles the standard behaviors of starting and ending TCP sessions. You will need to create a subclass of CTCPEndpoint to provide behaviors specific to the TCP protocol you are using. The **CTelnetInterp** class, a descendant of CTCPEndpoint, provides additional behaviors specific to Telnet-based protocols. The **CTelnetCmdLine** class, which is not yet implemented, will provide additional behaviors for protocols which follow the Telnet-style command line interface (such as FTP, NNTP, and POP3).

The **UTurboTCP** class contains some utility routines that link the TurboTCP event-handling mechanism to the application's event loop.

Two helper classes are provided for your convenience. (These classes are not required in TurboTCP applications, and are provided largely for compatibility with TurboTCP 1.0.) If

you wish to treat TCP sessions as document objects (as is done in MiniTelnet), you may wish to use the **CTCPSessionDoc** class, which is multiply-inherited from CTCPEndpoint and CDocument. The **CTCPApplication** class (a subclass of CApplication) simply installs the TurboTCP event handling mechanism into the TCL main event loop. If you choose not to use the CTCPApplication class, you must provide the same behaviors of CTCPApplication in your application class. These are easy to mimic.

Figure 2-1 **TurboTCP class diagram**



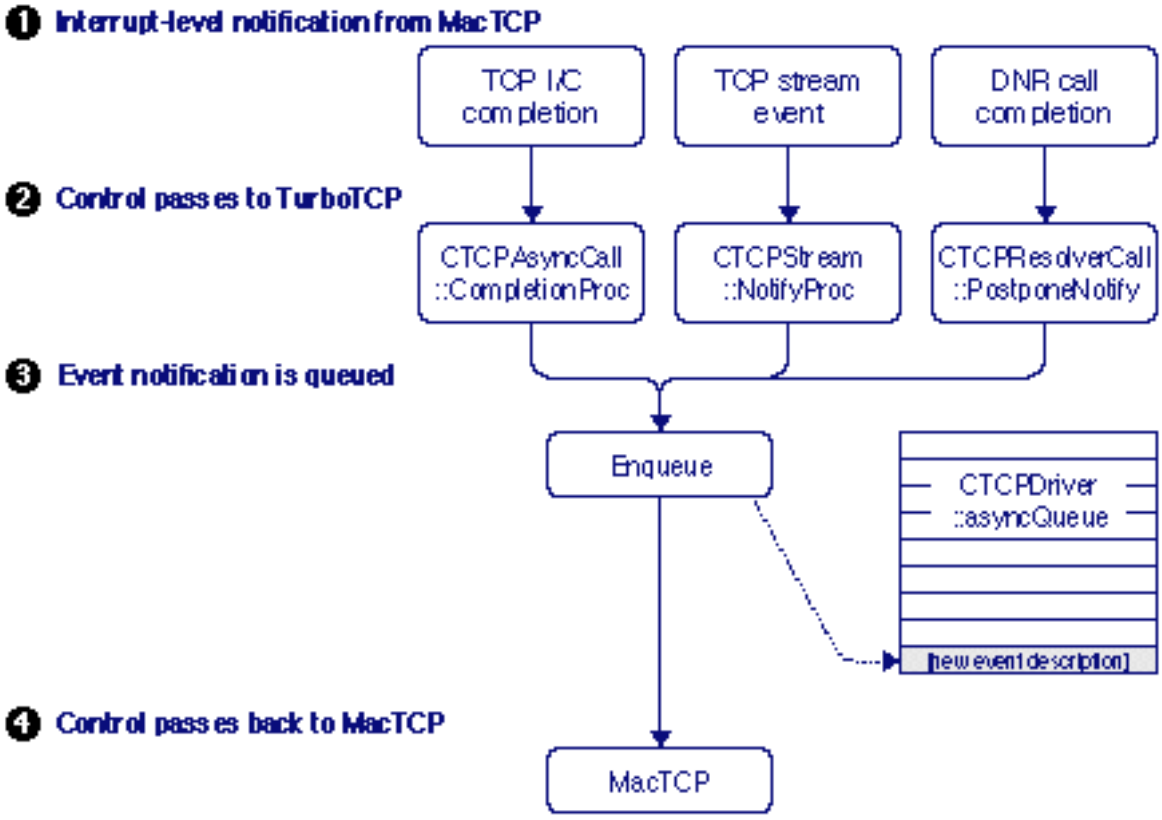
Flow of Control

There are two primary stages of the TurboTCP event loop which bear mentioning.
(A discussion of the creation of connections, etc., is included in the chapter on CTCPEndpoint.)

First is the mechanism for processing interrupt-level notifications from MacTCP. This process is described by Figure 2-2. As interrupt notifications are received from MacTCP, they are noted by TurboTCP and matched with existing data structures (the asynchronous call object, the TCP stream object, or the resolver call object).

The completion routines provided by each of these classes then set any flags related to the event notification and add the object to the TurboTCP event queue. The Mac Toolbox routine **Enqueue** is used to implement the queue, thus guaranteeing that any manipulations of the queue are safe from interrupts and will be safe under any future Macintosh System versions. No significant processing takes place in any of the TurboTCP interrupt-level procedures. These routines are written to minimize time spent with interrupts disabled.

Figure 2-2 **Interrupt-level flow of control**



Processing of TCP and DNR notifications is delayed until application event-loop time. The event-loop processing is described in Figure 2-3 (on the next page). Your application must call the routine **UTurboTCP::ProcessNetEvents** each time through the event loop (or at some regular interval). The CTCPApplication class takes care of this for you.

Once control passes to UTurboTCP::ProcessNetEvents, the library determines how much time can be spent in the network event loop and the maximum number of TCP notifications to be processed. The private routine CTCPDriver::ProcessOneNetworkEvent is called repeatedly until the network event queue is emptied or the allotted time for the network event loop has elapsed.

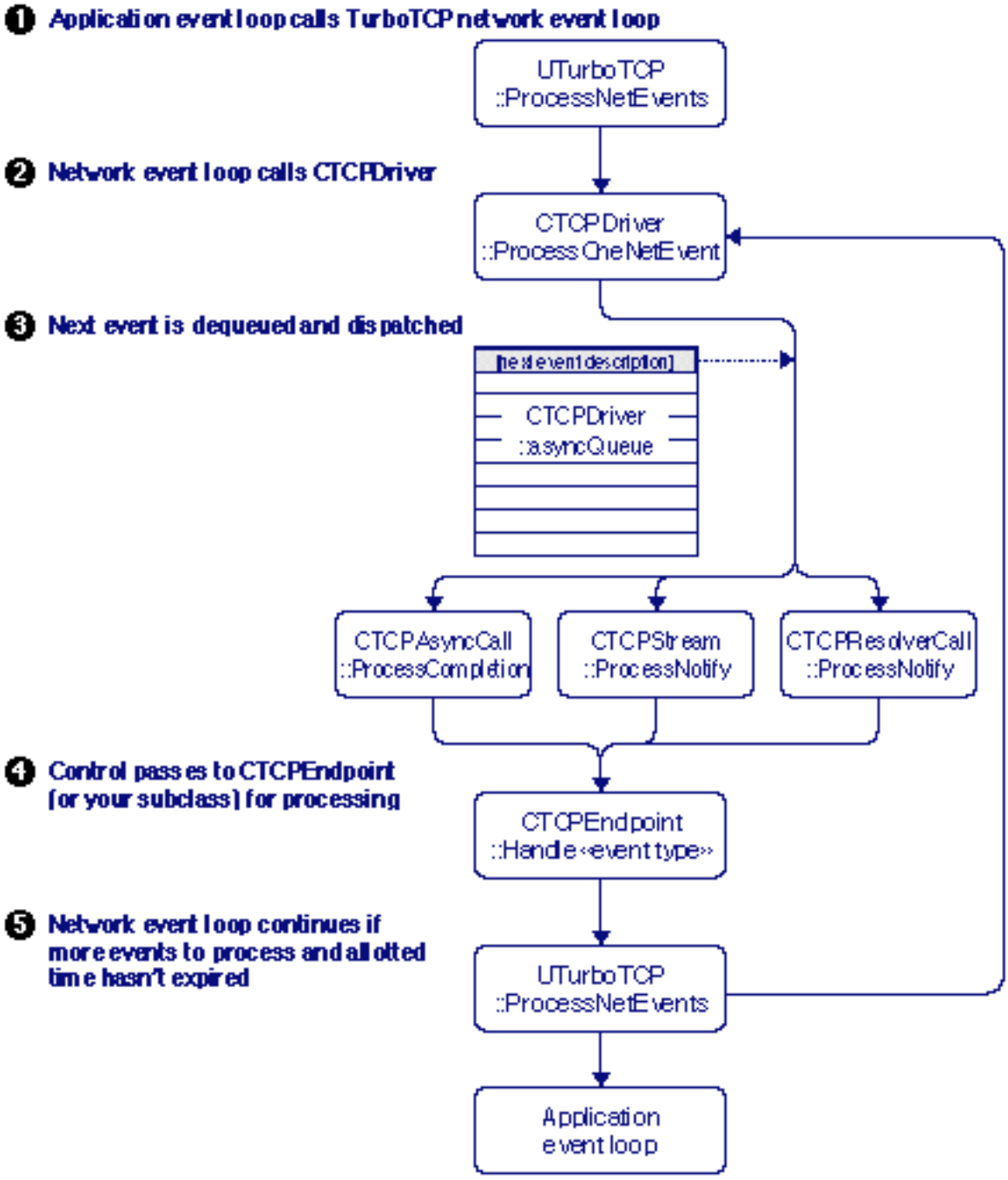
Each time CTCPDriver::ProcessOneNetworkEvent is called, a single event is removed from the TCP event queue and processed. Depending on the nature of the call, control is eventually passed to one of the methods of CTCPEndpoint named Handle…. The most commonly-used method from this set is CTCPEndpoint::HandleDataArrived.

The Handle… methods are your opportunity to perform the processing which is specific to the protocol that you are interpreting. Default behaviors are provided for most of these methods and you need not override all of them. You do need to override HandleDataArrived to interpret data which arrives from the remote host on the connection. (These methods are described in more detail in the chapter on CTCPEndpoint.)

Note

There is no provision for your application to do any processing at interrupt-level. You are strongly discouraged from modifying TurboTCP to gain access to the interrupt-level chain of command. These routines are highly likely to change in future versions of the library. ❖

Figure 2-3 **Event loop flow of control**



Converting from TurboTCP 1.0

The TurboTCP interface is similar to that in version 1.0. The primary changes are those related to the new mix-in architecture and preparation for Open Transport. Keep the following points in mind as you port your application from TurboTCP 1.0 to 2.0.

- **TCL 2.0 and true C++ now required.**
- **CTCPEndpoint mix-in class added.** This class provides most of the functionality of the old CTCPSessionDoc, but is not based on any TCL class. The next several bullets describe some of the implications of this change.
- **UTurboTCP class added.** This utility class encapsulates some TurboTCP behaviors which are not linked to any specific connections.
- **CTCPSessionDoc is now an optional class.** Some users of TurboTCP 1.0 chose to “roll their own” classes, copying many of the methods from CTCPSessionDoc to their own subclasses of other TCL objects (such as CDialogDirector). In such cases, you will need to rewrite your class to be multiply-inherited from CTCPEndpoint and the TCL object. (This should simplify your class definitions.) If you have been using the actual CTCPSessionDoc class, you will not need to make any changes.
- **Collaborator links removed from CTCPStream and CTCPResolverCall.** Both classes were descendants of CCollaborator in version 1.0. They are now standalone classes in 2.0 and the BroadcastChange references have been changed to direct references to the appropriate methods (Handle…) in CTCPEndpoint.
- **Interfaces changed for CTCPDriver, CTCPStream, and CTCPResolverCall.** In anticipation of Open Transport support, the entire interface to these three classes have been declared unsupported (though they will exist in large part in different classes in future versions of TurboTCP). These interfaces, by and large, correspond directly to MacTCP calls which will not be a part of the Open Transport programmer’s interface. If you have called methods of these classes directly, change these references to use the new CTCPEndpoint and UTurboTCP interfaces instead.
- **Two source files added.** You will need to add the following two files to your project: **CTCPEndpoint.cp** and **UTurboTCP.cp.**
- **New resource added (PowerPC builds only).** A new resource ('Ttcp' 23000) was added. This resource holds a 68K code resource version of the TCP completion routine to be used on PowerPC builds. It is included in the new **TurboTCP.rsrc** file. You need not include it in 68K-only builds.
- **Revised session closing mechanism.** The new interface to CTCPEndpoint includes methods RemoteClose and LocalClose which replace the old CTCPSessionDoc::Close. In TurboTCP 1.0, it was difficult to determine when the Close method should be called and under what circumstances it would be called by the library. This has been clarified by the changes to the interface.
- **Receive-bypass procedures removed.** The receive-bypass was a way of bypassing the inefficient method dispatch mechanism in Think C 6.0. As it turned out, it didn’t actually boost performance significantly – and it was an unsightly part of the library interface. All data-arrived notifications are now passed **only** through CTCPEndpoint::HandleDataArrived.

An Example: The MiniTelnet Application

The MiniTelnet is a limited but extensible application which implements the Internet standard Telnet protocol for terminal servers. This application was developed while testing the TurboTCP library, and serves as a good example of how to develop applications with the library.

Its interface should be fairly straightforward. Be aware that it does **not** provide any terminal emulations other than basic ASCII codes. (Specifically, it does not implement the VT100 protocol.)

Study its code closely to understand how TurboTCP is used in practice. Feel free to use its source code and extend it for your applications. (The MiniTelnet source is covered by the same license agreement as the rest of TurboTCP.)

Introduction

The CTCPEndpoint class is a base class for any class object which manages a single TCP connection session. It is designed with a mix-in architecture. You would typically mix it with a descendant of TCL's CDirector class (such as CDialogDirector or CDocument) to provide the user-interface behaviors.

Essentially no user-interface behavior is implemented in this class. (There is a method which provides error messages when TCP I/O errors occur.) It is the responsibility of your subclass to provide a suitable user interface.

Heritage

Base class:	none
Subclasses:	CTCPSessionDoc CTelnetInterp
Friend classes:	CTCPStream CTCPResolverCall
Friend of:	CTCPStream CTCPResolverCall

Using CTCPEndpoint

Each CTCPEndpoint object represents a **single** connection session between the user's Macintosh and a remote host. The constructor for CTCPEndpoint automatically creates the CTCPStream object, which in turn create MacTCP's stream structure, and the CTCPResolverCall object.

Deleting a CTCPEndpoint object results in an attempt to abort the TCP stream connection (if it is open) and cancel any DNR calls. The CTCPStream and CTCPResolverCall objects may persist in memory beyond the lifetime of the CTCPEndpoint object if the connection cannot be closed immediately. (They will delete themselves once the connection is properly terminated.) This behavior is transparent to the programmer and to the user.

CTCPEndpoint contains several abstract methods which can be overridden to provide common user-interface behaviors, including automatic window titling, and responding to local- and remote-close events.

You should subclass CTCPEndpoint to implement behaviors specific to the TCP protocol you are implementing, and to implement any user interface appropriate to that protocol.

Member Functions

constructor

public:
CTCPEndpoint

```
(unsigned short theDefaultPort,
unsigned long recBufferSize = recReceiveSize,
unsigned short autoReceiveSize = recAutoRecSize,
unsigned short autoReceiveNum = recAutoRecNum,
Boolean doUseCName = TRUE);
```

Initializes the endpoint object. Creates the CTCPStream and CTCPResolverCall objects; the constructor will throw an exception if there is not enough memory to construct these objects. The parameter **`theDefaultPort`** specifies the default remote host TCP port for your protocol (i.e., port 23 for Telnet). **`recBufferSize`** specifies the number of bytes in the MacTCP receive buffer for this connection. (Values can range from 4,096 to 131,072. Use smaller buffers for low-volume protocols such as Telnet; use larger numbers for higher-speed applications such as FTP.) **`autoReceiveSize`** specifies the number of entries in the RDS (read data structure) for the automatic "no-copy receive" calls that are issued by CTCPStream. (A typical value is 4.) **`autoReceiveNum`** specifies the number of TCPNoCopyRcv calls which will be issued simultaneously by CTCPStream. (Values can range from 4 for low-speed applications to 32 for high-speed applications.) **`doUseCName`** should be set true if you wish to have the object look up the "canonical name" for the remote host.

Note

The parameter sequence for this call is somewhat different from that of the CTCPSessionDoc::ITCPSessionDoc method in TurboTCP 1.0. ❖

destructor

public:
`virtual` **~CTCPEndpoint**`();`

Deletes the CTCPStream and CTCPResolverCall objects. These objects may linger in memory past the life of the endpoint object if transactions are still pending or sessions are still being closed. These objects will "self-destruct" when it is safe for them to do so. This behavior is handled transparently by the library.

Opening and Closing Sessions

OpenUserHost

public:

```
virtual void OpenUserHost
    (char* theHostName,
    unsigned short theDefaultPort,
    Boolean allowPortChange);
```

Opens a connection to a remote host. The string **theHostName** specifies the host to connect to. This string may be in dotted decimal format (e.g., "128.1.2.3") or in domain-name syntax (e.g., "host.somewhere.edu"). The parameter **theDefaultPort** specifies the default TCP port for your protocol (i.e., port 23 for Telnet). If **allowPortChange** is true, theHostName may (at the user's discretion) contain a port number which will override theDefaultPort. If the user wishes to provide such a port number, it must follow the host name with a single space then the port number in decimal format. No other characters are allowed. If allowPortChange is false, the port override format just described will not be detected and a bad DNR lookup will occur.

This routine calls the domain-name server (StrToAddr call) to resolve the host name which was provided. Upon successful completion of the DNR lookup, a connection is opened using the TCPActiveOpen call. If the connection is successfully opened, the method **HandleOpened()** is called. If for any reason the connection can be opened (including a bad DNR lookup), the method **HandleOpenFailed()** is called.

The remote host's name is copied to the data member **hostCName**. If the data member **useCName** is true, OpenUserHost will initiate a name lookup on the address which is returned – so that the "canonical name" of the host will be available in the hostCName field.

OpenHost

public:

```
virtual void OpenHost
    (unsigned long remoteHostIP,
    unsigned short remoteHostPort);
```

Opens a connection to a remote host without using the domain-name resolver. The parameter **remoteHostIP** contains the intended remote host's IP address, and **remoteHostPort** contains the remote TCP port number.

The remote host's name (in dotted decimal format) is copied to the data member **hostCName**. If the data member **useCName** is true, OpenHost will initiate a name lookup on the address which is returned – so that the "canonical name" of the host will be available in the hostCName field.

Listen

public:
`virtual void` **Listen**
`(unsigned long remoteHostIP,`
`unsigned short remoteHostPort);`

Waits for a remote host to initiate a connection. If you wish to specify the IP address of the remote host or port number, you may use the parameters **`remoteHostIP`** and **`remoteHostPort`**. If you wish to specify the local port number, you may use the method **`SetLocalHostPort`** prior to calling `Listen`. To specify a timeout for the open command, use the method **`SetOpenTimeout`** prior to calling `Listen`.

Once a connection is opened, if the data member **`useCName`** is true, `OpenHost` will initiate a name lookup on the remote host so that the "canonical name" of the host will be available in the `hostCName` field.

Note
There may be a bug with this routine when specifying a port number on the local end of the connection. This is being investigated. ❖

LocalClose

public:
`virtual Boolean` **LocalClose**`(Boolean quitting);`

Use this method when the program or user wishes to close the connection (i.e., a click is received in the close box, the user chooses Close or Quit from the menu bar, or the data transmission is completed). It initiates a graceful close of the TCP session (using TCPClose). If the session is currently being opened or closed, it is aborted (using TCPAbort).

Note that the signature for this routine is identical to that of `CDirector::Close()`. It is designed to be used in tandem with this method. For example, see the behavior in CTCPSessionDoc:

```
Boolean CTCPSessionDoc::Close(Boolean quitting)
{
    closeAndQuit = quitting;
    if (!ConfirmClose(quitting))
        return FALSE;
    if (itsFile)
        itsFile->Close();
    if (!LocalClose(quitting))
        return FALSE;
    return CDirector::Close(quitting);
}
```

SessionEstablished

public:
`virtual Boolean` **SessionEstablished** `();`
Use this method to determine if a valid connection is established. If the connection is established and ready for data transfer, the **`return`** value will be true; otherwise it will be false.

Receiving Data

By default, CTCPEndpoint objects will automatically issue the necessary MacTCP calls to receive data from an open connection. MacTCP performance seems to be optimized by issuing several TCPNoCopyRcv calls simultaneously. TurboTCP will perform this for you. The specified number of calls will be issued when the connection is opened. As each call is completed, it is reissued so data may be received continuously. The sequence of call completions is maintained properly.

The constructor for CTCPEndpoint takes as arguments the number of simultaneous receive calls, the number of entries in the RDS (read data structure) for each call, and the size of the receive buffer. Use the following table as a guide to configuring these parameters:

Application	recBufferSize	autoReceiveSize	autoReceiveNum
Minimum settings	4,096	1	1
Low-throughput (Telnet)	16,384	4	4
High-throughput (FTP)	110,592	16	32

Depending on the nature of your application, you may choose values anywhere between the minimum and high-throughput values listed above. Data received for this endpoint will be passed to **`HandleDataArrived`**. This method is described later in this chapter.

Note
At this time, there are no methods in CTCPEndpoint to support manually receiving data. ❖

Sending Data

The following member functions are responsible for sending data to the remote host on this connection. For applications where a large volume of data is being sent, you will need to take some care to ensure that available memory does not become an issue. (In other words, you will need to establish a mechanism for monitoring the amount of data sent at a single time.)

SendBfrCpy

public:

```
void SendBfrCpy
    (const void* theData,
    unsigned short theDataSize);
```

Make a copy of the data pointed to by **`theData`**, then sends this data to the remote host associated with this endpoint. The parameter **`theDataSize`** contains the number of bytes in the data. The copy of the data is disposed after the data are sent successfully. (Up to 65,535 bytes may be sent at a single time. You may wish to keep the data size to 1,024 bytes or less to improve performance.)

Use this method when you wish to send data which are held on the stack (i.e., local variables for a function) or in an object which could be disposed while the session is still active.

SendBfrNoCpy

public:

```
void SendBfrNoCpy
    (const void* theData,
    unsigned short theDataSize,
    Boolean disposeWhenDone,
    Boolean notifyWhenDone);
```

Sends the data pointed to by **`theData`** to the remote host associated with this endpoint. The parameter **`theDataSize`** contains the number of bytes in the data. (Up to 65,535 bytes may be sent at a single time. You may wish to keep the data size to 1,024 bytes or less to improve performance.)

Use this method when you wish to send data held on the stack (i.e., local variables for a function) or in an object which could be disposed while the session is still active.

If the parameter **`disposeWhenDone`** is true, the data will be disposed by calling `DisposPtr(theData)`. If the parameter is false, there will be no attempt to dispose the data.

If the parameter **`notifyWhenDone`** is true, the member function HandleDataSent will be called when this data block has been successfully sent to the remote host. (If such notification is not necessary, leave this parameter set to false. It will improve application performance.)

❖ **W A R N I N G**

Use this method **only** when you can be positive that the data block will remain until MacTCP has successfully sent the data. The data must be located on the heap in a structure which will not be disposed. ❖

❖ **W A R N I N G**

If you set `disposeWhenDone` to true, the data **must** have been allocated using NewPtr. Any other kind of pointer (handle dereference, member of structure, etc.) will almost certainly cause the Memory Manager to crash. ❖

SendChar

public:
`void` **SendChar**`(const char theChar);`
Send the single character **`theChar`** to the remote host associated with this endpoint. Uses the method `SendBfrCpy`.

SendCString

public:
`void` **SendCString**`(const char* theString);`
Send the C string pointed to by **`theString`** to the remote host associated with this endpoint. Uses the method `SendBfrCpy`.

SendPString

public:
`void` **SendPString**`(const char* theString);`
Send the Pascal string pointed to by **`theString`** to the remote host associated with this endpoint. Uses the method `SendBfrCpy`.

SetNextPush

public:
`void` **SetNextPush**`();`
Send the next data with the TCP “push” flag set. Applies only to the next single data packet sent.

SetNextUrgent

public:
`void` **SetNextUrgent**`();`
Send the next data with the TCP “urgent” flag set. Applies only to the next single data packet sent.

operator <<

friend
```
CTCPEndpoint& operator << (CTCPEndpoint& s, const char* v);
CTCPEndpoint& operator << (CTCPEndpoint& s, const unsigned char* v);
CTCPEndpoint& operator << (CTCPEndpoint& s, char v);

CTCPEndpoint& endl(CTCPEndpoint& ep);
CTCPEndpoint& local_IP(CTCPEndpoint& ep);
CTCPEndpoint& remote_IP(CTCPEndpoint& ep);
CTCPEndpoint& push(CTCPEndpoint& ep);
CTCPEndpoint& urgent(CTCPEndpoint& ep);
```

C++ iostreams-like interface to CTCPEndpoint. These methods are defined as inline functions which call the **`Send`**... methods described above. The manipulators provide special output: **`endl`** sends a carriage return (ASCII 0x0D), **`local_IP`** sends the local Macintosh's IP address in dotted decimal format, and **`remote_IP`** sends the remote host's IP address in dotted decimal format.

The manipulator **`push`** sends the next data with the TCP "push" flag set; **`urgent`** sends the next data with the TCP "urgent" flag set. Both of these manipulators apply only to the next single send operation. For example, the statement:

```
ep << urgent << "ABC" << "DEF";
```

will send only the data "ABC" as urgent data.

Note
There is no `operator >>` to receive data to CTCPEndpoint. ❖

Configuration Methods

SetDefaultPort

public:
```
void SetDefaultPort(unsigned short newDefaultPort);
```
Sets the default TCP port for the remote end of the connection to **`newDefaultPort`**. The value applies only to connections opened after this call.

SetLocalHostPort

public:
```
void SetLocalHostPort(unsigned short newLocalPort);
```
Sets the default TCP port for the local end of the connection to **`newLocalPort`**. The value applies only to connections opened after this call.

SetOpenTimeout

public:
`void` **SetOpenTimeout**`(unsigned short openMaxSeconds);`

Sets the maximum time to wait for a connection to be opened to **`openMaxSeconds`**. The value applies only to connections opened using `OpenUserHost` or `OpenHost` after this call. For subsequent `Listen` commands, the connection must be opened within `openMaxSeconds` **after** the initial connection request is received from the remote host. If you do not call this method (or if `openMaxSeconds` is 0), the default value of two minutes will be used.

SetListenTimeout

public:
`void` **SetListenTimeout**`(unsigned short openMaxSeconds);`

Sets the maximum time to wait for a connection to be opened to **`openMaxSeconds`**. The value applies only to connections opened using **`Listen`** after this call. If you do not call this method (or if `openMaxSeconds` is 0), `Listen` will wait indefinitely for a connection. A `Listen` command may be canceled by disposing the `CTCPEndpoint` object.

SetTypeOfService

public:
```
void SetTypeOfService(Boolean lowDelay,
    Boolean highThroughput,
    Boolean highReliability);
```

Sets the type of service requested for upcoming connections opened on this endpoint. The values apply only to connections opened after this call.

SetPrecedence

public:
`void` **SetPrecedence**`(TCPPrecedence newPrecedence);`

Sets the precedence requested for upcoming connections opened on this endpoint to **`newPrecedence`**. The value applies only to connections opened after this call. Values for `newPrecedence` are defined in **CTCPEndpoint.h**. If you do not call this routine, the default of `precRoutine` will be used. This should suffice for most circumstances.

SetDontFragment

public:
`void` **SetDontFragment**`(Boolean newDontFragment);`

Sets the don't fragment flag for upcoming connections opened on this endpoint to **`newDontFragment`**. The value applies only to connections opened after this call. Setting the `newDontFragment` flag may result in non-delivery of some larger data packets. The default value is false.

SetTimeToLive

public:
`void` **SetTimeToLive**`(unsigned short newTimeToLive);`
Sets the maximum “hop count” (or time-to-live) for upcoming connections opened on this endpoint to **`newTimeToLive`**. The value applies only to connections opened after this call. The default value is 60.

SetSecurity

public:
`void` **SetSecurity**`(unsigned short` **newSecurity**`);`
If the **`newSecurity`** flag is non-zero, enables the MacTCP default security parameter on upcoming connections opened on this endpoint to **`newTimeToLive`**. The value applies only to connections opened after this call. The default value is 0.

Selectors

GetRemoteHostName

public:
`void` **GetRemoteHostName**`(char* hostStringBfr);`
Returns the domain name of the remote host on this endpoint to the string **`hostStringBfr`**. The string buffer must be at least 256 characters long.

GetRemoteHostIP

public:
`unsigned long` **GetRemoteHostIP**`();`
Returns the IP address of the remote host on this endpoint.

GetRemoteHostPort

public:
`unsigned short` **GetRemoteHostPort**`();`
Returns the TCP port of the remote host on this endpoint.

GetLocalHostIP

public:
`unsigned long` **GetLocalHostIP**`();`
Returns the IP address of the local host on this endpoint.

GetLocalHostPort

public:
`unsigned short` **GetLocalHostPort**`();`

Returns the TCP port of the local host on this endpoint. For a `Listen` command, this method can be used to obtain the randomly-assigned port number once `Listen` has been called.

GetDefaultPort

public:
`unsigned short` **GetDefaultPort**`();`

Returns the default remote host TCP port for this endpoint. The default port is typically set by the subclass of CTCPEndpoint.

State Change Notifications

RemoteClose

protected:
`virtual void` **RemoteClose**`();`

This method is called whenever the remote host initiates a termination of the session (either a graceful close or abort). The method, as provided, responds to the closing event properly. Your method should override this method to close windows or destroy objects as appropriate. **Be sure to call the inherited method.** For example, see this behavior in CTCPSessionDoc:

```
void CTCPSessionDoc::RemoteClose()
{
    if (goAwayOnClose)
        Close(FALSE);
    CTCPEndpoint::RemoteClose();
}
```

StateChanged

protected:
`virtual void` **StateChanged**`();`

Called by other methods of CTCPEndpoint whenever the status of the connection changes. You may override this method to provide any other behaviors; calling the inherited method is not necessary.

The default method builds a new title for the window and calls `SetWindowTitle()`. If the data member **`showFileName`** is true, it calls the method `GetFileName()` to obtain the name of the file associated with the connection. If the data member **`showHostName`** is true, the name of the host (as represented in **`hostCName`**) is added to the window title. If both flags are true, a colon separates these two strings. If the connection is not ready, the host name is bracketed by parenthesis.

The default title for an active connection looks something like this:

```
file_name : host_name
```

For an active session on a non-standard port, the title looks like this:

```
file_name : host_name port#
```

For the same session (when `showFileName` is false), the title looks like this:

```
host_name port#
```

For an inactive session, the title looks like this:

```
file_name : (host_name)
```

The characters used for these strings may be customized. They are loaded from the `'STR#'` 23000 resource. Six strings are defined in this resource:

string 1	**Empty session.** Used when no host name is given. Default is “No session”.
string 2	**Separator.** Used when both filename and hostname are displayed. Default is a single colon surrounded by a space on either side.
string 3	**Not ready prefix.** Precedes the host name when a session is not active. Default is a left parenthesis.
string 4	**Not ready suffix.** Follows the host name when a session is not active. Default is a right parenthesis.
string 5	**Untitled window string.** Used when `showFileName` is true, but `GetFileName()` returns an empty string (i.e., the document is untitled). A number assigned to the document by the global CDecorator object is added to this string. Default is “Untitled-”.
string 6	**Port delimiter.** Used when the connection is on a different port number than the default port number specified when the constructor for CTCPEndpoint was called. When this occurs, this string (followed by the actual port number in decimal) is attached to the host name in. Default is a single space.

Document and Window Naming

CTCPEndpoint provides this set of routines to support the process of titling windows for the connection sessions. This behavior is optional; you may ignore all of these methods and title windows using your own convention.

SetWindowTitle

protected:
`virtual void` **SetWindowTitle**`(Str255 newTitle);`

Called by `StateChanged()` to set a new window title. If your subclass of CTCPEndpoint is also a subclass of CDirector, you can use the following code:

```
CYourClass::SetWindowTitle(Str255 newTitle)
{
    if (itsWindow)
        itsWindow->SetTitle(newTitle);
}
```

The default method does nothing. You need not call it.

GetFileName

protected:
`virtual void` **GetFileName**`(Str255 theName);`

Called by `StateChanged()` to get the name of the file associated with this connection.

If your subclass of CTCPEndpoint doesn't associate a file with the connection (i.e., it isn't a subclass of CDocument), you should leave the data member **`showFileName`** set to false (the default) and not override this method.

If you do implement file behaviors –and wish to have the filename included in the window title– you should set **`showFileName`** to true and override this method as follows:

```
CYourClass::GetFileName(Str255 theName)
{
    if (itsFile)
        itsFile->GetName(theName);
    else
        theName[0] = '\0';
}
```

The default method does nothing. You do not need to call it.

Error Handling

TCPErrorAlert

protected:
`virtual short` **TCPErrorAlert**
`(OSErr err, long message, short alertID, short parm3);`

Displays a customized error message to indicate that an error has occurred on this connection. Use this method instead of TCL's `ErrorAlert()` routine to provide more specific information about the connection (i.e., the remote host name). The standard TCL mechanism is used to convert error number **`err`** to a string error message. The parameter **`message`** has the same meaning as in the TCL. The dialog resource of type `'ALRT'` and number **`alertID`** is displayed; the parameter **`parm3`** is converted to a decimal number string and substituted for ^3 wherever it appears in the dialog. (This parameter is typically used to represent the TCP operation code.) The **`return`** value is the item number in the dialog which causes the alert to be completed (usually 1 for "OK").

Four custom error alerts (`'ALRT'` 23000…23003) are defined for special error conditions. These dialogs include special text and ParamText slots related to TCP errors. These are included in the **TurboTCP.rsrc** file and described in the chapter on TurboTCP resources.

Notification Routines

The following routines are called when various TCP events are called. They are called by CTCPStream or CTCPResolverCall. These methods are called at event-loop time (or whenever `UTurboTCP::ProcessNetEvents()` is called). You may override any or all of these methods to provide special handling when specific network events occur. You need override only the `HandleDataArrived()` method.

IMPORTANT
In no case are any of these routines called at interrupt time. This means you are free to allocate memory or make any Toolbox call from any of these routines. ❖

HandleDataArrived

protected:
`virtual void` **HandleDataArrived**
`(void* theData, unsigned short theDataSize,`
`Boolean isUrgent) = 0;`

Called when data arrives for this connection. **`theData`** points to the data. There will be **`theDataSize`** bytes of data. If the data arrived with the TCP urgent flag set to true, the **`isUrgent`** parameter will be set to true.

IMPORTANT
This method is declared pure virtual. You **must** override it. ❖

HandleClosed

protected:
`virtual void` **HandleClosed**`();`
Called when the session is actually closed. Default method does nothing. You need not call it.

HandleClosing

protected:
`virtual void` **HandleClosing**`(Boolean remoteClosing);`
Called when either the local or remote host initiates a closing of the connection. If the remote host is initiating the close, the **`remoteClosing`** parameter will be set to true; otherwise it will be false.

The default method calls `StateChanged()` to cause a retitling of the window and calls the `RemoteClose()` method if `remoteClosing` is true.

If you override `HandleClosing()`, you **must** call the inherited method.

HandleDataSent

protected:
```
virtual void HandleDataSent(void* theData,
    unsigned short theDataSize);
```
Called when data is successfully sent to the remote host. The **`theData`** parameter points to the original data (of length **`theDataSize`**). Default method does nothing. You need not call it. (The CTCPStream class takes care of disposing of the memory used for sending data automatically.)

❖ **I M P O R T A N T**
This method is called **only** for data sent through the **`SendBfrNoCpy`** method with the `notifyWhenSent` parameter set to true. It is not called for data send through the `SendBfrCpy` method (and any of the helper methods which call `SendBfrCpy`). ❖

HandleICMP

protected:
```
virtual void HandleICMP(ICMPType reportType,
    unsigned short optionalAddlInfo,
    void* optionalAddlInfoPtr);
```
Called when an ICMP report is passed to the connection stream. The message received is described by the parameter **`reportType`**. Some message types may contain additional information; if so, it will be contained in the parameters **`optionalAddlInfo`** and **`optionalAddlInfoPtr`**. (A full description of the ICMP messages is contained in the MacTCP Developer's Guide, and is beyond the scope of this manual.)

The default method does nothing. You need not call it.

HandleOpened

protected:
`virtual void` **HandleOpened**`();`

Called when a connection is successfully established. Default method records the remote TCP port number of the connection and calls `StateChanged()`. If you override this method, you should call the inherited method.

HandleOpenFailed

protected:
`virtual void` **HandleOpenFailed**`(OSErr theResultCode);`

Called when an attempt to open a connection fails. Failure may occur at the DNR lookup stage if `OpenUserHost()` was used or at the TCPActiveOpen or TCPPassiveOpen call. When called, the **`theResultCode`** will contain the error code (Macintosh OS error number) describing the reason for failure.

The default method calls `TCPErrorAlert()` to display an error message, then calls `RemoteClose()` to simulate a session closing. It also signals failure with a kSilentError parameter (to prevent future error messages).

HandleSendFailed

protected:
```
virtual void HandleSendFailed
    (void* theData, unsigned short theDataSize,
    OSErr theResultCode);
```

Called when a TCPSend operation fails. The **`theData`** parameter points to the original data (of length **`theDataSize`**). **`theResultCode`** will contain the error code (Macintosh OS error number) describing the reason for failure. The default method calls `TCPErrorAlert()` to display an error message. You need not call it.

You should not attempt to send the data again as this notification represents a permanent inability of the network to transmit the data. Also, you should not display an error message in this method, as the `HandleTCPError` method is called once for send failure.

❖ **I M P O R T A N T**
This method is called **only** for data sent through the `SendBfrNoCpy` method with the `notifyWhenSent` parameter set to true. It is not called for data send through the **`SendBfrCpy`** method (and any of the helper methods which call `SendBfrCpy`). ❖

HandleTCPError

protected:
`virtual void` **HandleTCPError**
`(OSErr theResultCode, short theCsCode);`

Called when MacTCP reports an error on the connection that is not related to a connection opening or send operation. **`theResultCode`** will contain the error code (Macintosh OS error number) describing the reason for failure. **`theCsCode`** contains the operation code of the MacTCP call which failed. The default method calls `TCPErrorAlert()` to display an error message. You need not call it.

HandleTerminated

protected:
`virtual void` **HandleTerminated**
`(TCPTerminReason terminReason, Boolean aboutToDispose);`

Called when a session is terminated (for any reason). When called **`terminReason`** contains the MacTCP reason code for the termination (see below). **`aboutToDispose`** is true if the stream is about to automatically dispose of itself. The default method displays an error alert using `TCPErrorAlert()` if the reason for termination is abnormal. It then signals that the state has changed by calling `StateChanged()` and `RemoteClose()`. If you override this method, you must call the inherited method.

The list of termination reasons is as follows:

`tcpTermRemoteAbort`	Remote host initiated an abort.
`tcpTermNetFailure`	Network failure.
`tcpTermSecurityMismatch`	Security/precedence mismatch.
`tcpTermTimeout`	Timeout.
`tcpTermAbort`	User abort. No error message is displayed.
`tcpTermClose`	User close. No error message is displayed.
`tcpTermServiceFailure`	Service failure.

HandleTimeout

protected:
`virtual void` **HandleTimeout**`();`

Called when an asynchronous notification of a timeout event occurs. This is not a serious error condition. If an error occurs related to a specific operation, it will be reported via the `HandleOpenFailed()`, `HandleSendFailed()`, or `HandleTCPError()` methods. Default method does nothing; you need not call it.

HandleUnexpectedData

protected:
`virtual void` **HandleUnexpectedData** `();`

Called when data arrives and no receive command has been issued to accept the data. If you are using auto-receive, this method should never be called. If you have disabled auto-receive, you should override this method to issue a new receive call. Default method does nothing; you need not call it.

HandleUrgentBegin

protected:
`virtual void` **HandleUrgentBegin** `();`

Called when data arrives with the TCP urgent flag set. If several data packets with urgent flags arrive in a row, this method will be called for only the first packet. Note that HandleDataArrived also gets notification of urgent data by way of its `isUrgent` parameter. Default method does nothing; you need not call it.

HandleStrToAddr

protected:
`virtual void` **HandleStrToAddr** `(struct hostInfo* theHostInfo);`

Called when the MacTCP DNR operation StrToAddr is completed. **`theHostInfo`** points to the "host information" record which includes the host's address.

Default method checks to see if the StrToAddr operation was called as part of an `OpenUserHost()` call. If so (and the DNR lookup was successful), it then initiates a TCPActiveOpen call to open the connection. If the data member **`useCName`** is true, it initiates an AddrToName call to obtain the canonical name of the host. If the DNR lookup failed, the method `HandleOpenFailed()` is called. If `OpenUserHost()` wasn't called, the default method does nothing.

If you override this method, be sure to call the inherited method.

❖ **W A R N I N G**
This method depends on a MacTCP-specific interface structure. It may be removed or replaced in a future version of TurboTCP. ❖

HandleAddrToName

protected:
`virtual void` **HandleAddrToName**`(struct hostInfo* theHostInfo);`

Called when the MacTCP DNR operation StrToAddr is completed. **`theHostInfo`** points to the "host information" record which includes the host's canonical name.

Default method checks to see if the StrToAddr operation was called as part of an `OpenUserHost()` call. If so (and the DNR lookup was successful), it then copies the host name which was provided to the data member **`hostCName`**. If the DNR lookup failed or if `OpenUserHost()` wasn't called, the default method does nothing.

If you override this method, be sure to call the inherited method.

❖ **W A R N I N G**
This method depends on a MacTCP-specific interface structure. It may be removed or replaced in a future version of TurboTCP. ❖

HandleHInfo

protected:
`virtual void` **HandleHInfo**`(struct returnRec* theHInfo);`

Called when the MacTCP DNR operation HInfo is completed. **`theHInfo`** points to the "host information" record which includes the host's CPU type and operating system (if available). Default method does nothing; you need not call it.

❖ **W A R N I N G**
This method depends on a MacTCP-specific interface structure. It may be removed or replaced in a future version of TurboTCP. ❖

HandleMXInfo

protected:
`virtual void` **HandleMXInfo**`(struct returnRec* theHInfo);`

Called when the MacTCP DNR operation HInfo is completed. **`theHInfo`** points to the "host information" record which includes the host's mail-exchange information (if available). Default method does nothing; you need not call it.

❖ **W A R N I N G**
This method depends on a MacTCP-specific interface structure. It may be removed or replaced in a future version of TurboTCP. ❖

Data Members

Configuration Fields

The following fields may be altered by your subclass of CTCPEndpoint to change its behaviors. The default values are listed here.

protected:

`Boolean` **useCName** `= true;`
If true, `OpenUserHost()` should look up host's canonical name.

`Boolean` **goAwayOnClose** `= false;`
Close window when session closes.

`Boolean` **showFileName** `= true;`
Show filename in window titles. See description for `StateChanged()`.

`Boolean` **showHostName** `= true;`
Show host name in window titles. See description for `StateChanged()`.

`short` **untitledNumber***;*
The number assigned to this window if there is no filename available. Used only if `showFileName` = true and the filename returned by `GetFileName()` is empty. See description for `StateChanged()`.

Class Library Reference:
UTurboTCP

Introduction

The UTurboTCP class contains a handful of utility functions for TCP services. All methods of UTurboTCP are static functions. They are grouped in UTurboTCP to reduce global namespace pollution.

There is no reason to create an instance of UTurboTCP.

Heritage

Base class:	none
Subclasses:	none
Friend classes:	none
Friend of:	CTCPDriver CTCPResolverCall

Member Functions

Opening/Closing TCP Services

The following routines are used to initialize and shut down the TurboTCP. They are called automatically by the CTCPApplication class. If you use CTCPApplication, you need not pay attention to these two routines. If you are designing your own application class, you will need to use these functions.

InitTCP

public:
`static void` **InitTCP**`();`

This routine should be called once at application start-up time. It creates the CTCPDriver object. In a future version of TurboTCP, it will detect the presence of Open Transport and adapt the program accordingly.

Note that the MacTCP driver is not opened at this time. This is to make TurboTCP applications more friendly to users with LAPs which dial out immediately when the MacTCP driver is opened (such as SLIP and PPP). (The MacTCP driver is opened only when a connection request is initiated through `CTCPEndpoint`.)

CloseTCP

public:
`static void` **CloseTCP**`();`

This routine should be called once at application quit time. It destroys the CTCPDriver object. If any TCP streams remain open, it will wait until they can be closed safely before returning. (This will prevent future applications from overwriting the TCP stream structures and crashing.)

Processing Network Events

ProcessNetEvents

public:

```
static void ProcessNetEvents
    (short maxEventsToProcess,
    long maxTicks);
```

Process one or more “network events.” Network events may include completion of TCP I/O operations, asynchronous stream notifications, or completion of domain-name resolver calls. These events are received at interrupt time and queued until they can be processed at this routine (free of interrupt-level constraints).

If you wish to limit the number of events processed at a single event loop, you may use the **`maxEventsToProcess`** parameter to specify the maximum number of network events processed. (If there are fewer events, the routine will stop when the queue is empty.) The value 0 will yield a default of 100 network events.

If you wish to limit the amount to time spend in the network event loop, you may use the **`maxTicks`** parameter to specify the maximum number of clock ticks (1/60 second) to spend processing network events. If `maxTicks` is 0, the routine will supply a default maximum time based on other CPU activity and whether the application is in the foreground or background.

IP Address Utilities

GetLocalIPAddr

public:
`static unsigned long` **GetLocalIPAddr();**
Returns the Macintosh's assigned IP address.

DoAddrToStr

public:
`static void` **DoAddrToStr**
`(unsigned long theIPaddr, char* theString);`
Converts the IP address stored in **`theIPaddr`** to a dotted decimal string in the buffer **`theString`**. (You must allow for up to 16 characters.)

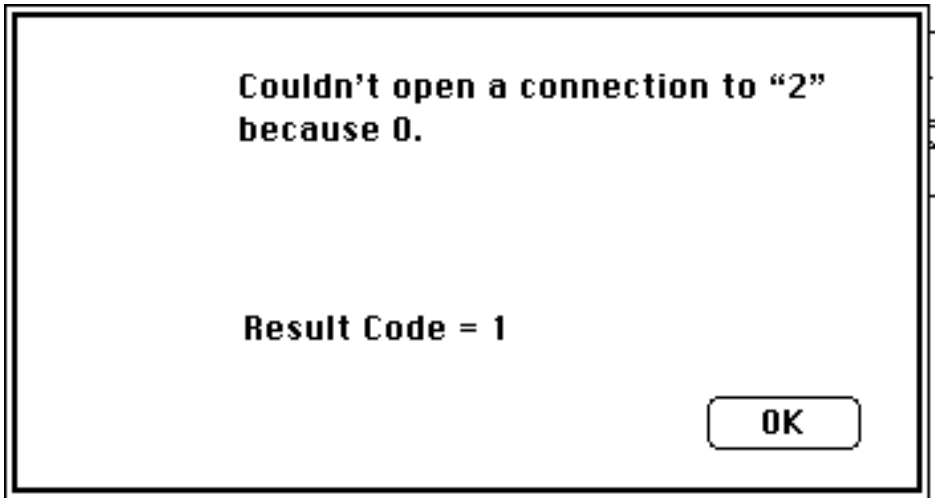
Introduction

This chapter describes a handful of resources which must be included in any TurboTCP application. These resources can be found in the file **TurboTCP.rsrc**. (Note that the MiniTelnet application has some additional resources. These resources are not described in this manual.)

Alerts and Dialogs

Open Failed Alert (ALRT, DITL 23000)

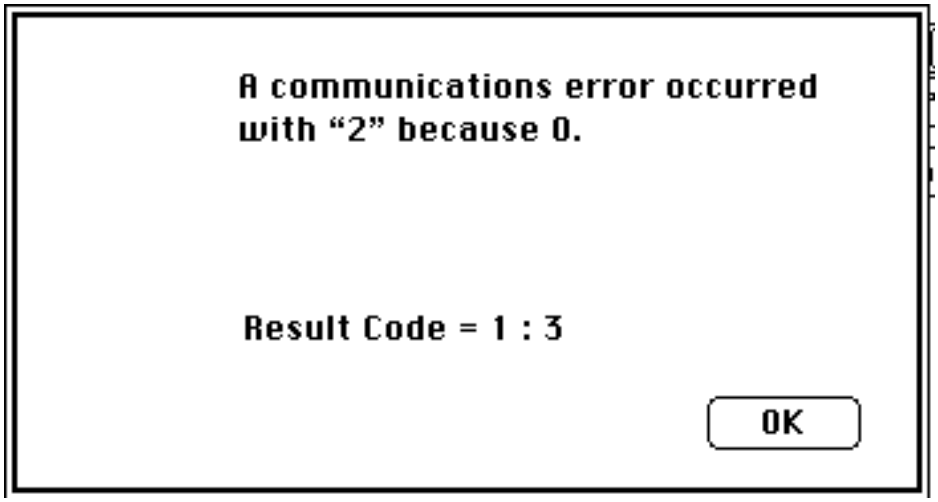
This dialog is displayed by **CTCPEndpoint::HandleOpenFailed** whenever an attempt to establish a TCP connection fails for any reason.



The parameter 0 is the reason for the failure; parameter 1 holds the Mac Toolbox error number; and parameter 2 is the name of the host to which a connection was attempted.

Unexpected Error Alert (ALRT, DITL 23001)

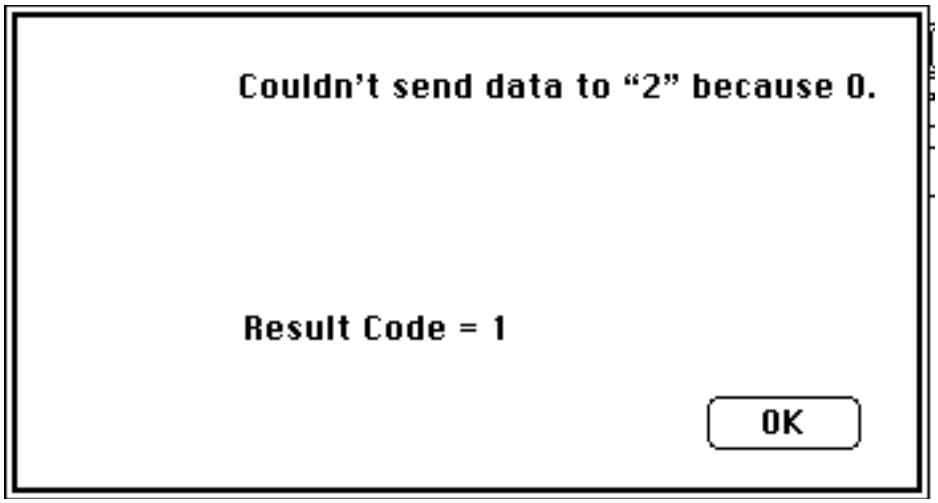
This dialog is displayed by **CTCPEndpoint::HandleTCPError** whenever MacTCP reports an error condition on a connection which has already been opened.



The parameter 0 is the reason for the error; parameter 1 holds the Mac Toolbox error number; and parameter 2 is the name of the remote host with which the error occurred; parameter 3 is the number of the TCP operation which failed.

Send Data Failed Alert (ALRT, DITL 23002)

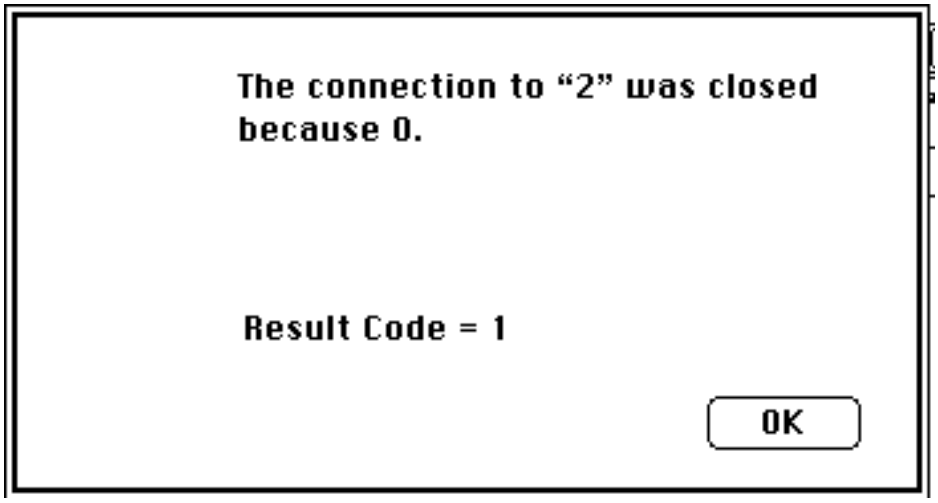
This dialog is displayed by **CTCPEndpoint::HandleTCPError** whenever MacTCP reports that data could not be sent to a remote host for any reason.



The parameter 0 is the reason for the error; parameter 1 holds the Mac Toolbox error number; and parameter 2 is the name of the remote host with which the error occurred.

Session Terminated Alert (ALRT, DITL 23003)

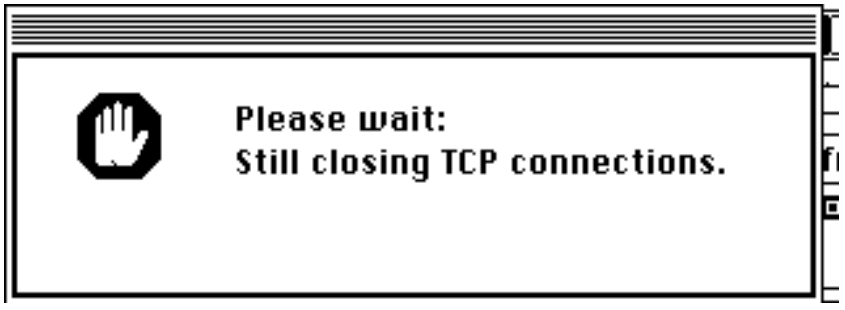
This dialog is displayed by **CTCPEndpoint::HandleTerminated** whenever MacTCP reports that a session was terminated for any unexpected reason (i.e. not a graceful close or user abort).



The parameter 0 is the reason for the session termination; parameter 1 holds the reason code for the termination; and parameter 2 is the name of the remote host with which the connection was lost.

Delayed Quit Dialog (DLOG, DITL 23010)

This dialog is displayed by **UTurboTCP::CloseTCP** whenever there are TCP sessions still open and there is a prolonged delay in terminating these sessions (i.e. two seconds or longer). The application can be in the background while this dialog is presented, but the user will not be allowed to perform any actions with the TurboTCP application.



Strings

Error Strings (Estr -23048 through +23018)

The Think Class Library uses error string ('Estr') resources to hold the text of error messages for common Macintosh error conditions. TurboTCP provides a set of additional error strings to describe common TCP errors. The negative resource numbers match the MacTCP error numbers, positive resource numbers correspond to additional error conditions defined by the TurboTCP library.

Window Title Strings (STR# 23000)

These strings are used by CTCPEndpoint::StateChanged to implement the automatic window titles. These strings are described below, and may be customized to the programmer's preferences:

string 1	**Empty session.** Used when no host name is given. Default is "No session".
string 2	**Separator.** Used when both filename and hostname are displayed. Default is a single colon surrounded by a space on either side.
string 3	**Not ready prefix.** Precedes the host name when a session is not active. Default is a left parenthesis.
string 4	**Not ready suffix.** Follows the host name when a session is not active. Default is a right parenthesis.
string 5	**Untitled window string.** Used when `showFileName` is true, but `GetFileName()` returns an empty string (i.e., the document is untitled). A number assigned to the document by the global CDecorator object is added to this string. Default is "Untitled-".
string 6	**Port delimiter.** Used when the connection is on a different port number than the default port number specified when the constructor for CTCPEndpoint was called. When this occurs, this string (followed by the actual port number in decimal) is attached to the host name in. Default is a single space.

Code Resources

Completion Procedure for PowerPC Builds (TTcp 23000)

This resource holds a 68K code resource version of the TCP completion routine. The PowerPC version of the routine CTCPAsyncCall::GetCompletionProc loads this routine and builds a "fat" routine descriptor for the completion procedure. The resource is ignored by the 68K version of the library.

Overview

TurboTCP contains a number of classes which are used to assist in the functioning of the TurboTCP user classes CTCPEndpoint and UTurboTCP. They will be briefly described in this chapter.

❖ **W A R N I N G**
These classes are for the internal use of TurboTCP only. They are likely to be changed in any future revision to TurboTCP (minor or major revisions). They classes are highly dependent on MacTCP data structures, and many changes are likely to be needed as support for Open Transport is added. ❖

The **CTCPStream** class encapsulates the stream object of MacTCP. It manages all interaction between CTCPEndpoint and MacTCP. The methods of CTCPStream include methods to directly call each of the MacTCP operations, configure upcoming calls, and provide notifications to the CTCPEndpoint object of I/O completions and asynchronous events.

The **CTCPAsyncCall** class encapsulates a single asynchronous call to MacTCP and allows the call to persist beyond the stack frame of any given method in CTCPStream. The methods of CTCPAsyncCall respond to notification that a call has completed and takes care of miscellaneous housekeeping following the completion (returning buffers, disposing of pointers, etc.)

The **CTCPResolverCall** class encapsulates the MacTCP DNR (domain name resolver). Its methods include methods to directly call each of the DNR operations, load and initialize the DNR code resource, and provide notifications to the CTCPEndpoint object of DNR completions.

The global **CTCPDriver** object acts as a referee, managing contact with the MacTCP driver. It maintains the queue of TCP and DNR events received at interrupt time, and dispatches the processing of these events at event-loop time. It also keeps a list of all open streams and resolver calls. This list is used at application shut-down to ensure that all MacTCP data structures are properly closed before the application quits.

Optional Class Reference:
CTCPApplication

Introduction

The CTCPApplication class is a subclass of TCL's CApplication class which links the TurboTCP library to the event loop. You would normally make your application object a subclass of CTCPApplication instead of CApplication.

You do not need to use the CTCPApplication class; it is provided as a convenience to you. If you choose not to use CTCPApplication, be sure to read the chapter on UTurboTCP.

Essentially no user-interface behavior is implemented in this class. It is the responsibility of your subclass to provide a suitable user interface.

Heritage

Base class:	CApplication
Subclasses:	(your application class)
Friend classes:	none
Friend of:	none

Using CTCPApplication

Incorporating the CTCPApplication class requires no special effort. The interface to CTCPApplication is identical to that of CApplication. No methods need to be overridden, other than those required by CApplication.

Background Application Operation

TurboTCP applications are designed to operate in the background. You need to configure the project to support this operation. Be sure that the “Background Null Events” flag in the application’s SIZE resource is set. (In Symantec C++, use the Set Project Type… command in the Project menu. In Metrowerks CodeWarrior, use the Project panel of the Preferences… dialog.)

CTCPApplication sets the maximum sleep time to 90 ticks (1.5 seconds). You may change this behavior to suit your application’s needs. I recommend somewhat lower values for high-throughput applications.

The utility function **`UTurboTCP::ProcessNetEvents`** controls the amount of time spent in the network event loop. Its decisions are based on the time elapsed since the previous time through the event loop; this behavior allows it to give more time when another application is consuming CPU time and to be more efficient when there is little competing CPU activity.

Member Functions

constructors

public:
CTCPApplication `();`

public:
CTCPApplication
```
    (short extraMasters,
    Size aRainyDayFund,
    Size aCriticalBalance,
    Size aToolboxBalance);
```

public:
`void` **ITCPApplication**
```
    (short extraMasters,
    Size aRainyDayFund,
    Size aCriticalBalance,
    Size aToolboxBalance);
```

These constructors are identical in parameters and operation to the constructors for CApplication.

MakeHelpers

public:
`virtual void` **MakeHelpers**`();`
Overrides `CApplication::MakeHelpers` to initialize the TurboTCP drivers. (Calls `UTurboTCP::InitTCP`).

Application Shutdown

Quit

public:
`virtual Boolean` **Quit**`();`
Overrides `CApplication::Quit` to shut down the TurboTCP driver. (Calls `UTurboTCP::CloseTCP`).

Network Event Processing

ProcessNetEvents

public:
`virtual void` **Process1Event**`();`
Overrides `CApplication::Quit` to process network events collected at interrupt time. (Calls `UTurboTCP::ProcessNetEvents`).

Introduction

The CTCPSessionDoc class is a subclass of TCL's CDocument class and TurboTCP's CTCPEndpoint class. This class is a handy way of associating the TCL user-interface behaviors. View this as a "connection session" document, the point at which user interaction with the TCP session is handled.

This class does not implement any specific TCP protocol. You will need to provide a subclass to do so. (As an example, see the CTelnetInterp class for interpreting the Telnet protocol.)

Heritage

Base classes:	CDocument, CTCPEndpoint
Subclasses:	(your document class)
Friend classes:	none
Friend of:	none

Using CTCPSessionDoc

View the CTCPSessionDoc as the appropriate class for any data which is collected or sent through the network and displayed in a window. Like TCL's CDocument, you must provide behaviors to manage any windows and files associated with the connection.

You must override the **`HandleDataArrived`** method to respond to notification of incoming data and display or store it in the appropriate manner for the protocol you are implementing.

By default, the window title will be set to the name of the remote host to which the endpoint is connected. This behavior can be altered by changing the member fields **`showFileName`** and **`showHostName`**, or by overriding the member function **`SetWindowTitle`**. (See the discussion under **CTCPEndpoint::StateChanged**.)

Member Functions

constructors

public:
CTCPSessionDoc

```
    (unsigned short theDefaultPort,
    unsigned long recBufferSize = recReceiveSize,
    unsigned short autoReceiveSize = recAutoRecSize,
    unsigned short autoReceiveNum = recAutoRecNum,
    Boolean doUseCName = TRUE,
    Boolean printable = TRUE);
```

Initializes the endpoint object. Creates the CTCPStream and CTCPResolver calls; the constructor will throw an exception if there is not enough memory to construct these objects. The parameter **`theDefaultPort`** specifies the default remote host TCP port for your protocol (i.e., port 23 for Telnet). **`recBufferSize`** specifies the number of bytes in the MacTCP receive buffer for this connection. (Values can range from 4,096 to 131,072. Use smaller buffers for low-volume protocols such as Telnet; use larger numbers for higher-speed applications such as FTP.) **`autoReceiveSize`** specifies the number of entries in the RDS (read data structure) for the automatic "no-copy receive" calls that are issued by CTCPStream. (A typical value is 4.) **`autoReceiveNum`** specifies the number of TCPNoCopyRcv calls which will be issued simultaneously by CTCPStream. (Values can range from 4 for low-speed applications to 32 for high-speed applications.) **`doUseCName`** should be set true if you wish to have the object look up the "canonical name" for the remote host.

Note

The parameter sequence for this call is somewhat different from that of the CTCPSessionDoc::ITCPSessionDoc method in TurboTCP 1.0. ❖

ITCPSessionDoc

public:
`void` **ITCPSessionDoc**

```
    (CApplication* aSupervisor,
    Boolean printable);
```

A TCL-1.1.x style constructor retained for compatibility only. If possible, use the new constructor instead.

Closing Windows and Sessions

Close

public:
`virtual Boolean` **Close**`(Boolean quitting);`

Use this method when the program or user wishes to close the connection (i.e., a click is received in the close box, the user chooses Close or Quit from the menu bar, or the data transmission is completed). It initiates a graceful close of the TCP session (using TCPClose). If the session is currently being opened or closed, it is aborted (using TCPAbort).

RemoteClose

public:
`virtual void` **RemoteClose**`();`

This method is called whenever the remote host initiates a termination of the session (either a graceful close or abort). The method, as provided, responds to the closing event properly. The CTCPSessionDoc version closes the window if the `goAwayOnClose` flag is set to true.

Window Titling

SetWindowTitle

public:
`virtual void` **SetWindowTitle**`(Str255 newTitle);`

Called by `StateChanged()` to set a new window title, based on the new status (open or closed) of the connection. If a window object is associated with this document, the window's title is updated.

GetFileName

protected:
`virtual void` **GetFileName**`(Str255 theName);`

Called by `StateChanged()` to get the name of the file associated with this connection. If a file object is associated with this document, the file's name is returned.

Other TCP/IP Software Libraries

The following software libraries are available to the general public via the Internet. They offer different approaches to providing access to the TCP protocol and the MacTCP driver.

- **GUSI.** GUSI is an extension and partial replacement of the MPW runtime library. Its main objective is to provide a more or less simple and consistent interface across the following communications domains: files, AppleTalk, System 7 PPC Toolbox, Internet/TCP, and Printer Access Protocol (typically networked PostScript printers). Requires MPW C 3.2 or later.

  Written by Matthias Neeracher <neeri@iis.ee.ethz.ch>. Available for FTP from nic.switch.ch in the directory software/mac/src/mpw_c (or any common archive site).

- **SK.** A port of GUSI to Symantec C++. Written by Stephan Deibel. Available for FTP from dsg.harvard.edu in the directory /pub/src/SK.

- **TCPOOExample.** These units implement a relatively high-level interface to MacTCP, and allow fairly easy writing of TCP applications in Pascal using event driven, object oriented programming. Requires Think Pascal 4.0.

  Written by Peter Lewis <peter.lewis@info.curtin.edu.au>. Available for FTP from redback.cs.uwa.edu.au in the directory /others/peterlewis (or any common archive site).

- **UTCP.** A fairy simple C++ library for accessing MacTCP. Consists mostly of wrapper functions for the MacTCP calls. Uses automatic-receive mechanism similar to that of TurboTCP. Requires MPW C++ 3.3 or later.

  Written by David Peterson (not sure of an e-mail address). Available for FTP from sumex-aim.stanford.edu in the directory /info-mac/dev/src (or on common mirror sites).

- **Source code for common MacTCP applications.** The full source code to several of the popular TCP applications (NewsWatcher, old versions of Eudora, NCSA Telnet, etc.) can be found on the network. Most of these applications were developed in C and fairly well documented.

- **Apple TCP snippets.** Tiny applications (with source code) which demonstrate particular aspects of MacTCP programming. Not particularly useful as a code base for a full-fledged application, however.

  Written by Apple's DTS support staff. Available for FTP from ftp.apple.com.

Programmer's Utilities

Two utilities written for MacTCP programmers bear mentioning here. They have been especially helpful to me as I have developed and updated TurboTCP.

- **MacTCP Watcher.** This application shows a list of all current MacTCP streams. Detailed information on each can be obtained by double-clicking on any given stream. Very handy and accurate information.

  Written by Peter Lewis <peter.lewis@info.curtin.edu.au>. Available for FTP from redback.cs.uwa.edu.au in the directory /others/peterlewis (or any common archive site).

- **ZapTCP.** An extension to protect programmers from system crashes caused by quitting applications which have MacTCP streams open. (This often happens when debugging such applications.)

  Written by Steve Falkenburg of Apple's Macintosh DTS. Available for FTP from sumex-aim.stanford.edu in the directory /info-mac/comm/tcp (or any common archive site).

Networking Protocols

Most of the networking protocols used in MacTCP are described in RFC documents (Request For Comments). These documents are available for FTP from the host nic.ddn.mil. They are also typically mirrored on campus archive sites across the nation (and probably across the world). Look to a local server before bothering the major site.

The list below is by no means comprehensive; it just represents an effort to visit the high points of network programming. These RFCs are current as of June 1993.

General TCP/IP RFCs

- **Internet Assigned Numbers.** Describes the assigned TCP port numbers for various protocols, Telnet option numbers, etc. RFC 1340.
- **Internet Control Message Protocol (ICMP).** RFC 792, 950.
- **Internet Protocol (IP).** RFC 791, 1349.
- **TCP/IP tutorial.** RFC 1180.
- **Transmission Control Protocol (TCP).** RFC 793.

Protocol-Specific RFCs

The following RFCs define common protocols “spoken” by TCP/IP hosts.

- **File Transfer Protocol (FTP).** RFC 959.
- **Finger user information protocol.** RFC 1288.
- **Internet Gopher protocol.** RFC 1436.
- **Line Printer Daemon (LPD) protocol.** RFC 1179.
- **Network News Transfer Protocol (NNTP).** RFC 977.
- **Network Time Protocol (NTP),** version 3. RFC 1305.
- **Post Office Protocol (POP3),** version 3. RFC 1225.
- **Remote Procedure Call protocol (RPC),** version 2. RFC 1057.
- **Simple Mail Transfer Protocol (SMTP).** RFC 823.
- **Simple Network Time protocol.** RFC 1361.
- **Telnet protocol specifications.** RFC 854.
- **Telnet options specifications.** RFC 855.
- **Trivial File Transfer Protocol (TFTP),** version 2. RFC 1350.

Telnet Options Specifications

A large number of options for the Telnet protocol have been defined. These are listed in order of the option number.

- **0: Transmit binary.** RFC 856.
- **1: Echo.** RFC 857.
- **3: Suppress go-ahead.** RFC 858.
- **5: Status.** RFC 859.
- **6: Timing mark.** RFC 860.
- **7: Remote controlled transmission and echoing (RCTE).** RFC 726.
- **18: Logout.** RFC 727.
- **19: Byte macro.** RFC 735.
- **20: Data entry terminal (DET).** RFC 1043.
- **21: SUPDUP.** RFC 736.
- **22: SUPDUP output.** RFC 749.
- **23: Send location.** RFC 779.
- **24: Terminal type negotiation.** RFC 1091.

- **25: End of record.** RFC 885.
- **26: TACACS user identification.** RFC 927.
- **27: Output marking (security) option.** RFC 933.
- **28: Terminal location.** RFC 946.
- **29: Telnet 3270 emulation regime.** RFC 1041.
- **30: X-3 pad.** RFC 1053.
- **31: Window size negotiation.** RFC 1073.
- **32: Terminal speed.** RFC 1079.
- **33: Remote flow control.** RFC 1372.
- **34: Linemode negotiation.** RFC 1184.
- **35: X windows display location.** RFC 1096.
- **36: Environment communications.** RFC 1408.
- **37: Authentication.** RFC 1416.
- **255: Extended options list.** RFC 861.
- **IBM 5250 terminal interface.** RFC 1205.