TCP/IP Scripting Addition

version 1.1.2

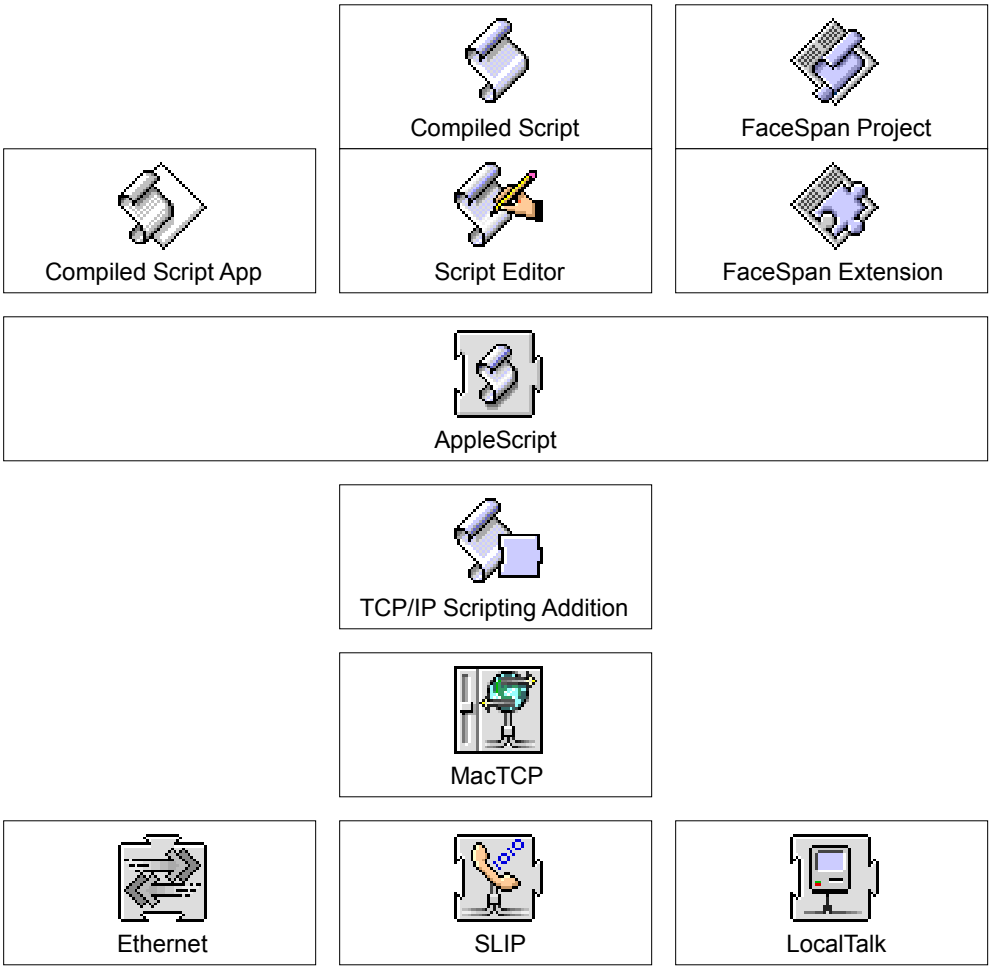
Mango Tree Software

Box 41119 • Providence, Rhode Island USA 02940
401–942–6882 • atul@netcom.com

Chapter 1

Introduction

The TCP/IP Scripting Addition allows Open Scripting Architecture (OSA) scripts — like AppleScript scripts — to execute commands related to TCP/IP. TCP/IP, which stands for Transmission Control Protocol/Internet Protocol, is the standard network communications method used on the Internet. Apple's implementation of TCP/IP on the Macintosh is MacTCP. The following diagram describes how the TCP/IP Scripting Addition works with AppleScript.



The TCP/IP Scripting Addition is called by AppleScript whenever a TCP/IP script command is executed. The TCP/IP Scripting Addition then uses MacTCP to conduct TCP/IP communications over Ethernet, EtherTalk, LocalTalk or Serial Line Internet Protocol (SLIP).

Chapter 2

Installation

The TCP/IP Scripting Addition requires MacTCP 2.0.4 and AppleScript (preferably version 1.1). Script Editor, Scriptable Text Editor, and FaceSpan are recommended to take full advantage of the Scripting Addition.
Drag the TCP/IP Scripting Addition to the Scripting Additions folder inside your Extensions folder, which is in your System Folder. You do not need to restart your Macintosh.

Upgrading

If you are upgrading from a previous version of the TCP/IP Scripting Addition, you will need to edit and recompile your compiled scripts. If you open a compiled script with the new TCP/IP Scripting Addition in place, you will notice several of your commands have changed. Statements using the following commands will need to be edited:

tcp write	Parameter `text` changed to `data`
tcp connect **tcp wait for connect**	You will need to edit any statements using fields of a returned stream object: Change «`class stid`» back to `stream id` Change «`class drid`» back to `driver id` Change «`class buff`» back to `buffer address` Change «`class read`» back to `already read`
tcp status	You will need to edit any statements using fields of a returned stream status object: Change «`class wait`» back to `bytes waiting` Change «`class stat`» back to `connection status` Change «`class lhst`» back to `local host` Change «`class lprt`» back to `local port` Change «`class rhst`» back to `remote host` Change «`class rprt`» back to `remote port` You will need to edit any statements using connection status constants: Change «`constant stenstr1`» back to `No connection` Change «`constant stenstr2`» back to `Waiting` Change «`constant stenstr3`» back to `Establishing`

	Change «`constant stenstr4`» back to `Connected` Change «`constant stenstr5`» back to `Close sent` Change «`constant stenstr6`» back to `Close requested` Change «`constant stenstr7`» back to `Closed`
tcp send **tcp receive**	You will need to edit any statements using transfer type constants: Change «`constant ttyptyp1`» back to `ISO88591` Change «`constant ttyptyp2`» back to `Raw Data` Change «`constant ttyptyp3`» back to `Raw Data Fork` Change «`constant ttyptyp4`» back to `Raw Resource Fork` Change «`constant ttyptyp5`» back to `MacBinary`

The easiest way to make these changes is to copy your script into a text editor (like MPW Shell or BBEdit) and use its search and replace functions.

This editing is now necessary because the internal keywords for many field names and constants had to be changed to prevent conflicts with applications using the same internal keywords (for example, MacPGP uses the keyword `read` for one of its parameters and this was conflicting with `already read`). The new internal keywords should not conflict with any applications.

Scripts written using earlier versions of the TCP/IP Scripting Addition and saved as text will, for the most part, not need to be edited. **It is recommended that you save your existing scripts as text before upgrading to this version.**

Important note on registering

Please note that until you become a registered user of the TCP/IP Scripting Addition version 1.1.2 and install the registered version, you will occasionally be presented with reminders to register while using certain commands. In particular, you will reminded up to two times whenever you use any of the newer commands and parameters that were not present in version 1.0, such as MacBinary file transfers, the debugging commands, or the new options in `tcp read`. For more information on registering, please read the Licensing Information text file included in this package, or contact Mango Tree Software at the address on the title page for a licensing agreement.

Command Set

tcp connect

Creates a TCP/IP stream and connects to the specified remote host and port number

Usage **`tcp connect`**

`to host` (string) — Connect to hostname

`port` (integer) — Connect to port number

[`from port` (integer)] — Optional parameter: connect from port number (default: use any arbitrary port)

[`maximum seconds` (small integer)] — Optional parameter: try to connect for this number of seconds (default: 120 seconds)

[`buffer size` (integer)] — Optional parameter: size of receive buffer in bytes (minimum/default: 4096)

Result stream — Stream object used for the other commands in the suite

Notes
- This new stream must be closed before the script application quits.

Example
```
set command_stream to (tcp connect to host "nic.ddn.mil" ¬
      port 30)
```

tcp wait for connect

Creates a TCP/IP stream and waits for an incoming connection from a remote host

Usage **`tcp wait for connect`**

[`port` (integer)] — Optional parameter: listen on port number (default is to

use an unused port)

[`buffer size` (integer)] — Optional parameter: size of receive buffer in bytes (minimum/default 4096)

Result stream — Stream object used for the other commands in the suite

Notes
- Do not read or write to the stream until the `connection status` of the stream (found through the `tcp status` command) equals `Connected`.
- This new stream must be closed before the script application quits.

Example

```
— Start waiting for a connection

set gopher_socket to (tcp wait for connect port 70)

— Loop until someone connects to us

repeat while (connection status of ¬
      (tcp status stream gopher_socket) ≠ Connected)
end repeat
```

tcp close

Closes and releases a TCP/IP stream

Usage **`tcp close`**

`stream` (stream) — Stream object

Result none

Notes
- Do not use the stream again after this command.

Example

```
tcp close stream command_stream
```

tcp shutdown

Partially closes one or both directions of a TCP/IP stream

Usage **`tcp shutdown`**

`stream` (stream) — Stream object

[`no further reads` (boolean)] — Optional parameter: closes the stream so that no further reading from the stream can take place (default is false)

[`no further writes` (boolean)] — Optional parameter: closes the stream so that no further writing to the stream can take place (default is false)

Result none

Notes
- You must still issue a `tcp close` on the stream when you are completely finished using the stream.
- Any `tcp read` issued on a stream after a `tcp shutdown no further reads` will return an empty string.

Example
```
tcp shutdown stream command_stream with no further reads
```

tcp status

Returns status of TCP/IP stream

Usage **`tcp status`**

`stream` (stream) — Stream object

Result stream status — Status of TCP/IP stream

Notes
- If the connection is actually closed or waiting for a close, but data remains in the internal buffer waiting for a `tcp read` or `tcp receive`, then `tcp status` will falsify a status of `Connected`.
- See Chapter 6 for more information on the internal buffers.

Example
```
set gopher_socket to (tcp wait for connect port 70)
tcp status stream gopher_socket

(* until there is a connection, this will return something like:

{class:stream status, bytes waiting:0, connection status:Waiting,
local host:"165.112.52.8", local port:70, remote host:"0.0.0.0",
remote port:0}
*)
```

tcp my address

Returns Internet address of this computer

Usage **`tcp my address`**

Result string — IP address for the given hostname

Notes
- This command returns the IP address for the local Macintosh. You do

not need any open streams for this to work.

Example
```
display dialog "My address is: " & (tcp my address)
```

tcp name to address

Returns IP address for the given hostname

Usage **`tcp name to address [string]`**

Result string — IP address for the given hostname

Notes
- This command returns the IP address for the given hostname. You do not need any open streams for this to work.

Example
```
display dialog "The IP address for nic.ddn.mil is " & (tcp name to address "nic.ddn.mil")
```

tcp address to name

Returns hostname for the given IP address

Usage **`tcp address to name [string]`**

Result string — Hostname for the given IP address

Notes
- This command returns the hostname for the given IP address. You do not need any open streams for this to work.

Example
```
display dialog "The hostname for 128.148.128.40 is " & (tcp name to address "128.148.128.40")
```

tcp ahead

Returns true if the specified characters are waiting ahead in the TCP/IP stream

Usage **`tcp ahead`**

`stream` (stream) — Stream object

`characters` (string) — Scan to see if this string of characters have been received

Result boolean — True if the specified characters are found ahead in the stream

Notes
- You will usually use this command before a `tcp read`, so that you

know what to read up to. See the `tcp read` command for more details.

- `tcp ahead` will return true only if the **entire** text to scan for is found waiting to be read.

- See Chapter 6 for more information on the internal buffers.

Example

```
--    This was wait until the connected user presses return
--    (which is followed by a linefeed)

set LF to (ASCII character of 10)

repeat until tcp ahead stream commandStream characters LF
end repeat
```

tcp debug dump buffer

Returns data waiting in the buffer to be read

Usage **`tcp debug dump buffer`**

`stream` (stream) — Stream object

Result string — Data waiting in buffer to be read

Notes

- `tcp debug dump buffer` returns the data waiting in the internal stream buffer. This data has been read by the Scripting Addition from the tcp stream, but has not yet been returned to the script.

- See Chapter 6 for more information on the internal buffers.

- You can use this command after a `tcp ahead` fails, so that you may see exactly what characters are waiting ahead in the stream.

Example

```
set found to (tcp ahead stream s characters "word:")
tcp debug dump buffer stream s
```

tcp read

Reads from a TCP/IP stream

Usage **`tcp read`**

`stream` (stream) — Stream object

[`maximum bytes` (integer)] — Optional parameter: maximum number of bytes to read (default: no limit)

[strip characters (string)] — Optional parameter: filter out any of these characters before returning the string (default: strip no characters)

[until characters (string)] — Optional parameter: read until these characters are received (default: read whatever is incoming)

[using ISO88591/Raw Data] — Optional parameter: translates characters using the specified method after receiving (default is Raw Data)

[as (anything)] — Optional parameter: read the incoming data in this form (default is text)

Result anything — Data returned

Notes

- To read up until a linefeed character, use:

```
tcp read stream commandStream until characters ¬
      (ASCII character of 10)
```

- To do the same, but also stripping linefeeds after carriage returns, use:

```
tcp read stream commandStream until characters ¬
      (ASCII character of 10) using ISO88591
```

- A tcp read with none of the optional parameters will return all the characters currently in the internal stream buffer as a string; it will not read indefinitely.

- The using parameter translates the characters after the until characters, maximum bytes, and strip characters parameters have been dealt with, but before assigning a type of the data to be returned to you. You may only use ISO88591, which uses the ISO 8859–1 translation table (including stripping linefeeds), or Raw Data, which receives the data without translation. You may **not** use any other translation types with tcp read.

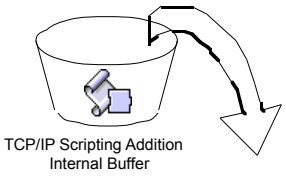
- If maximum bytes is specified, no more than that number of characters is returned. If you are also stripping or translating characters, then the number of characters returned may be less than the maximum you specified.

- The until characters parameter can be more than one character. For example, to read data up to “abcdefg” and return, you might use:

```
tcp read stream commandStream until characters "abcdefg" & return
```

- See Chapter 6 for more information on the internal buffers.

- The following figure the order of parameter processing in the tcp read command



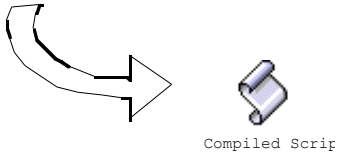
Check if until charactersparameter fou

Do not take more thaximum bytesparamete

Process with strip charactersparameter and ISO88591, if pres

Process with ISO88591 option, if pre

Return data with the type specified as parameter



• The `tcp read` attempts to read all the incoming characters waiting in the stream before trying to find the `until characters` string. If the last few incoming characters start to match the `until characters` string, but there are no more characters following, then the condition is treated as if the entire `until characters` string was found. Continuing the above example, if the data stream had only “abcd” before the `tcp read` was executed, the `tcp read` will match sucessfully and will return the characters up to and including “abcd”.

• It is best not to use `tcp read` unless you know that the number of characters waiting in the stream (found with `tcp status`) is greater than zero, or if the characters you are looking for (found with `tcp ahead`) are waiting to be read.

• If the `until characters` parameter is not found in the incoming characters, then **an empty string or an empty list** is returned (depending on whether a list was requested in the `as` parameter). This is very important!

• You may specify an AppleScript type in the `as` parameter to use for returning the data. For example, if you specify `as string`, `tcp read` will return the data as a string (which is the default behavior).

• Specify `as {string}` to get the data as a list of single characters.

- Specify `as integer` to get a 4 byte integer (read from the stream as most significant byte to least significant byte, or the XDR standard).

- Specify `as {integer}` to get a list with one 4 byte integer (read from the stream as most significant byte to least significant byte, or the XDR standard). If you specified `maximum bytes 3`, then the first 3 bytes will be used to make an integer. If you specified `maximum bytes` with more than 4 bytes, then every 4 bytes will be used to make an integer and any remaining bytes will be used to make a final integer.

- Specify `as data` to get the entire incoming data as a single data object.

- Specify `as {data}` to get a list of 1 byte data objects.

- Specify `as "PICT"` to get the entire incoming data as a single data with the type specified in quotes (or in this example, PICT).

- Specify `as {"PICT"}` to get a list of 1 byte data objects with the type specified in quotes (or in this example, PICT).

Example

```
--    This will read a line of text up until a linefeed, and will
--    return only the non-return and non-linefeed characters in the
--    line. If the linefeed is not present, it returns the empty
--    string.

set LF to (ASCII character of 10)

set inLine to (tcp read stream s until characters LF ¬
      strip characters return & LF)
--    Here are several examples using the as paramter
--    Assume the incoming data is the sequence of bytes
--    (in hexadecimal) 40 41 42 43 44
tcp read stream s as integer
--    will return 1.078018627E+9 (or $40414243 in decimal)
tcp read stream s as {integer}
--    will return {1.078018627E+9} (or $40414243 in decimal)
tcp read stream s as {integer} maximum bytes 10
--    will return {1.078018627E+9, 69} (or $40414243, $44 in
--          decimal)
tcp read stream s as string
--    will return "ABCDE"
--    same as tcp read stream s as text, or just tcp read stream
tcp read stream s as {string}
--    will return {"A", "B", "C", "D", "E"}
--    same as tcp read stream s as {text}
tcp read stream s as short
--    will return 16706 (or $4041 in decimal)
tcp read stream s as {short} maximum bytes 10
--    will return {16706, 17220, 69}
tcp read stream s as {short} maximum bytes 10 until characters "D"
--    will return {16706, 17220}
--    Do you understand why it returned only 2 numbers?
--    All the bytes up to and including the "D" ($44) were used.
```

```
tcp read stream s as "PICT"
--    will return «data PICT4142434445»
tcp read stream s as "PICT" maximum bytes 4
--    will return «data PICT41424344»
tcp read stream s as data
--    will return «data rdat41424344445»
tcp read stream s as {data}
--    will return {«data rdat41», «data rdat42», «data rdat43»
--          «data rdat44», «data rdat45»}
--    Now assume the incoming data is the sequence of bytes
--    (in hexadecimal) 40 41 0D 0A (or "AB", carriage return,
--    linefeed)
tcp read stream s as integer
--    will return 1.09484775E+9 (or $40410D0A in decimal)
tcp read stream s as integer using ISO88591
--    will return 4276749 (or $40410D in decimal)
--    Do you see why? The last 0A byte was removed because of the
--    ISO 8859-1 translation.
```

tcp receive

Receives a file from a TCP/IP stream

Usage **`tcp receive`**

`stream` (stream) — Stream object

`destination` (anything) — File, alias, or file access reference number

[`maximum bytes` (integer)] — Optional parameter: maximum number of bytes to read (default: no limit)

[`strip characters` (string)] — Optional parameter: filter out these characters before writing to the file (default: strip no characters)

[`until characters` (string)] — Optional parameter: read until these characters are received (default: read until stream is closed)

[`maximum seconds` (integer)] — Optional parameter: read until this many seconds have passed waiting for characters (default: no time limit)

[`using ISO88591/Raw Data/Raw Data Fork/Raw Resource Fork/MacBinary`] — Optional parameter: translates characters using the specified method after receiving (default is Raw Data)

[`append` (boolean)] — Optional parameter: if an existing destination file is specified, append to the end of that file instead of replacing that file (default is false — replace the file if it exists)

Result none

Notes

- This is similar to the `tcp read` command, except the input is placed into a file. `tcp receive` has the same methods of specifying how much to read, plus it has a `maximum seconds` parameter to specify how long to wait when no new characters are being received.

- If `maximum bytes` is specified, no more than that number of characters is placed in the file. If you are also stripping or translating characters, then the number of characters in the file may be less than the maximum you specified.

- If the stream is closed by the remote host before the `until characters` text is found, then the remaining characters are read and written into the file and the command returns without an error.

- The `until characters` parameter works differently than the way it works in `tcp read`. With `tcp receive`, characters are read until the **entire** `until characters` parameter is found.

- The countdown timer specified in the `maximum seconds` parameter starts after the last character is read. If more characters arrive, the timer is reset. `tcp receive` stops waiting for characters only when the countdown timer expires.

- The `destination` parameter can be a file (e.g. `file "Macintosh HD:test"`), alias (e.g. `alias "Macintosh HD:test"`), or an open file reference number (e.g. obtained with `open for access alias "Macintosh HD:test"`). You can get file reference numbers by using the Read/Write Commands Scripting Addition, included with AppleScript 1.1.

- If you specify a open file reference number in the `destination` parameter, the received data is placed at the file mark (i.e. right after where you last read or wrote to the file). However, if you use the `append` option, received data is placed at the end of the open file.

- Specifying an existing file in a file or alias object as the `destination` parameter overwrites that file, unless you use the `append` parameter, in which case the incoming data is appended to the end of the file.

- The `destination` file name is ignored if the `MacBinary` option is used and the stream data is correctly interpretable as a MacBinary file. However, the location of the file specified by `destination` is used (e.g. for `destination file "Macintosh HD:folder 1:myfile"`, the new file is placed in `"Macintosh HD:folder 1"` with the original name specified in the MacBinary data). If a file already exists at that location with the name prescribed in the MacBinary header data, the data is written out into a file with the `destination` file name and an Duplicate File error is returned.

• The `using` parameter translates the characters after the `until characters`, `maximum bytes`, `maximum seconds`, and `strip characters` parameters have been dealt with, and immediately before saving the characters in the file. Specifying `ISO88591` uses the ISO 8859–1 translation table (including stripping linefeeds). Specifying `Raw Data` and `Raw Data Fork` sends the data fork of a file. Specifying `Raw Resource Fork` sends the resource fork of a file. Specifying `MacBinary` sends both forks of a file as a MacBinary II formatted file.

• Why is there both a `Raw Data` and `Raw Data Fork` option? Because you cannot use the `Raw Data Fork`, `Raw Resource Fork`, or `MacBinary` options if you specify a file reference number in the `destination` parameter (it does not make sense to specify `Raw Resource Fork`, when the file's data fork given in the file reference number is already open). Your only two options when using file reference numbers are `ISO88591` and `Raw Data`.

Example

```
set LF to (ASCII character of 10)

tcp receive stream s destination file "Macintosh HD:inFile" ¬
      until characters return & "." & return ¬
      maximum seconds 30 using ISO88591
```

tcp write

Writes text to a TCP/IP stream

Usage **`tcp write`**

`stream` (stream) — Stream from tcp connect

`data` (anything) — Data to send to host

[`using ISO88591/Raw Data`] — Optional parameter: translates characters using the specified method before sending. (default is Raw Data)

Result none

Notes • This command sends the specified data out through the specified TCP/IP stream.

• The `data` parameter used to be called `text`. Be sure to change your older scripts.

• The `using` parameter translates the characters before sending them. You may only specify `ISO88591`, which uses the ISO 8859–1 translation table (including adding linefeeds after carriage returns), or `Raw Data` which sends

the data as is.

- The data parameter can be a string, an integer, or a list of such elements. In the case of a list, the elements are concatenated before sending. Using `data {"Howdy ", "doody"}` is the same as `data "Howdy doody"`.

- Specifying `data 1` sends the integer 1 as a 4 byte integer, from most significant byte to least significant byte. Specifying `data (1 as string)` sends the ASCII character "1".

Example

```
set LF to (ASCII character of 10)
set CR to return
set CRLF to CR & LF

tcp write stream data_stream data {"USER ", ftp_user, return} ¬
      using ISO88591
```

tcp send

Sends a file through a TCP/IP stream

Usage **`tcp send`**

`stream` (stream) — Stream object

`source` (anything) — File, alias, or file access reference number

[`using ISO88591/Raw Data/Raw Data Fork/Raw Resource Fork/MacBinary`] — Optional parameter: translates characters using the specified method before sending. (default is Raw Data)

Result none

Notes

- This command simply sends the specified file out through the specified TCP/IP stream.

- The `using` parameter translates the characters before sending them. Specifying `ISO88591` uses the ISO 8859–1 translation table (including adding linefeeds after carriage returns). Specifying `Raw Data` and `Raw Data Fork` sends the data fork of a file. Specifying `Raw Resource Fork` sends the resource fork of a file. Specifying `MacBinary` sends both forks of a file as a MacBinary II formatted file.

- Why is there both a `Raw Data` and `Raw Data Fork` option? Because you cannot use the `Raw Data Fork`, `Raw Resource Fork`, or `MacBinary` options if you specify a file reference number in the `destination` parameter (it does not make sense to specify `Raw Resource Fork`, when the file's data

fork given in the file reference number is already open). Your only two options when using file reference numbers are `ISO88591` and `Raw Data`.

- Specifying an open file reference number in the `source` parameter sends the file from the current file mark (right after where you last wrote to or read from the file).

Example

```
tcp send stream s source (file "HD:sample file") using ISO88591
```

Classes Used

class stream

TCP/IP stream (internal use only; format will definitely change in the future). You should not need to access any of these fields.

Properties

`driver id` (small integer) [r/o] — MacTCP Driver ID

`stream id` (integer) [r/o] — Stream ID for this stream

`buffer address` (integer) [r/o] — Address of internal MacTCP buffer for this stream

`already read` (integer) [r/o] — Bytes already read from MacTCP

class stream status

TCP/IP stream status, returned by `tcp status`.

Properties

`bytes waiting` (integer) [r/o] — Number of bytes waiting to be read

`connection status` (No connection/ Waiting/ Establishing/ Connected/ Close Sent/ Close requested/ Closed) [r/o] — Connection status of the stream

`local host` (string) [r/o] — Internet address of this computer

`local port` (integer) [r/o] — Port number of this stream

`remote host` (string) [r/o] — Internet address of the remote computer

`remote port` (integer) [r/o] — Port of the remote computer

Chapter 5

Miscellaneous tips on using the TCP/IP Scripting Addition

• While debugging your scripts, I strongly recommend using the ZapTCP extension, by Steven Falkenburg. This extension causes open TCP streams to be closed when an application unexpectedly quits, and is especially useful when you forget to close streams you have left opened.

• You may find the Regular Expression matching commands in Script Tools, by Mark Alldritt, very helpful. For example, to parse a line like:

```
250 test... Sender ok
```

you can use the commands:

```
set lline to "250 test... Sender ok"

set expr to (compile regular expression "([0-9]*)(.)(.*)")
set scan to match regular expression expr to lline
```

and the results are stored as:

```
scan = {matched:true, match string:"250 test... Sender ok", match 1:"250", match
2:" ", match 3:"test... Sender ok"}
```

This way, you can easily pick apart the content of messages to get error numbers, error messages, and other useful items.

• After you use a `tcp connect` command, the statements following the connect up until the `tcp close` should be enclosed in a `try/on error` construct, like:

```
set sss to (tcp connect to host "…" port 70)
try
      -- Read, write, and otherwise use the stream
on error msg number num from obj partial result pr
      tcp close stream sss
      error msg number num from obj partial result pr
end try

tcp close stream sss
```

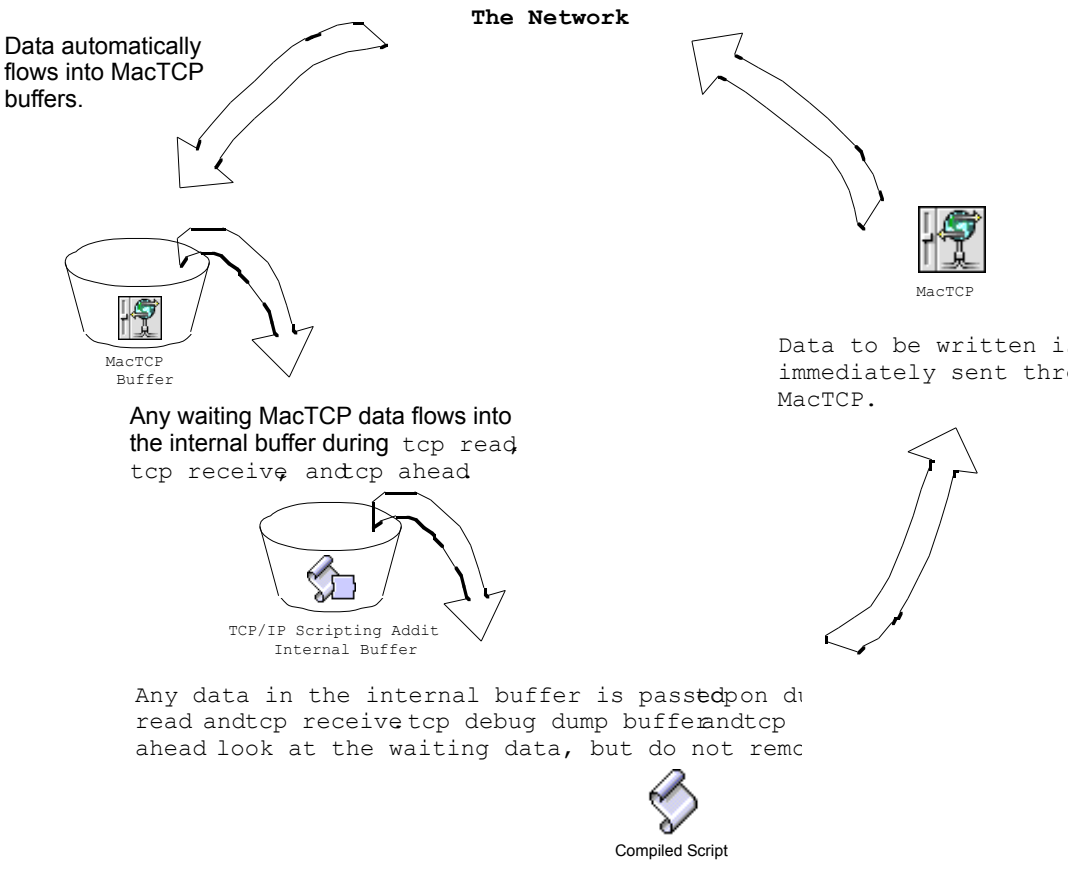
This way, streams will not be left open accidently if script errors are encountered.

• This extension has been tested with AppleScript 1.1 and with MacTCP 2.0.4. It works

under System 7.5 as well.

Internal Details

The following diagram explains the flow of data from a TCP/IP network to an AppleScript script.



From this diagram, you will note that MacTCP and the TCP/IP Scripting Addition have their own buffers (or buckets of data). Incoming TCP/IP data automatically flows into MacTCP's buffer. The size of this MacTCP buffer is set during the `tcp connect` command.

Data flows into the TCP/IP Scripting Addition internal buffer when the commands `tcp read`, `tcp receive`, and `tcp ahead` are executed. Note that `tcp read` and `tcp ahead` take all the data that is immediately waiting in the MacTCP buffer, whereas `tcp receive` will continue to read until the stopping criteria are met. The size of this internal buffer is variable; this buffer automatically expands and contracts to hold as much data as is present.

Finally, the AppleScript script gets the incoming data with `tcp read`. This data can be placed directly into a file with `tcp receive`. The script can scan the data in the internal buffer to see if certain characters are present using `tcp ahead`. The script writer can peek at the contents of the internal buffer with `tcp debug dump buffer`.

When `tcp shutdown no further reads` is executed, all TCP/IP Scripting Addition stream commands watch for arriving data in the MacTCP buffer. If any is found, the data is discarded and is not brought into the TCP/IP Scripting Addition internal buffer.

Note that data sent with `tcp send` and `tcp write` is sent immediately through MacTCP and does not accumulate in any buffers.

When `tcp shutdown no further writes` is executed, the TCP/IP Scripting Addition performs a half–close on the stream so that the remote host is signalled that no further data will be arriving.

Learning More About Common Internet Protocols

If you would like to explore further uses of the TCP/IP Scripting Addition, I suggest first browsing through the sample scripts included with this release.
Request for Comments (RFC) are the best references for the various Internet protocols. Here is a list of some of the more common Internet protocols, and the RFCs that describe them:

RFC 959	File Transfer Protocol (FTP)
RFC 1436	Gopher
RFC 1179	Line Printer Daemon (LPD)
RFC 821	Simple Mail Transfer Protocol (SMTP)
RFC 854 RFC 857 RFC 859 RFC 1073 RFC 1079 RFC 1080 RFC 1184 RFC 1408	Telnet and the various options to negotiate

You can download RFCs by anonymous FTP to nic.ddn.mil.

I also strongly recommend the text *TCP/IP Illustrated, Volume 1* by W. Richard Stevens, published by Addison–Wesley Publishing Company, 1994 (ISBN 0–201–63346–9). This book has the clearest examples of all the above protocols, plus many others. I would not have been able to write the Baby Telnet sample without it.

Acknowledgments

Thanks to Peter Lewis (author of many excellent products including Script Daemon, Finger daemon, and FTPd) for providing the code for ISO 8859–1 translation.

Thanks to Mark Alldritt for allowing me to use an early version of his upcoming Script Debugger. This was incredibly valuable in debugging the sample scripts.

Thanks to Roel Vertegaal for providing suggestions for improving the Send Mail sample script.

And of course, thanks to the users who have paid to license the TCP/IP Scripting Addition! Without your support, I would not be still working on it!