Waterloo TCP

Programmers Guide
Erick Engelke
Release 1.20

Welcome to Waterloo TCP

Waterloo TCP is a TCP/IP stack which is freely available to anyone. There are no royalties on the software. The only restrictions are that the kernal libraries may not be sold by themselves and that the copyright of the authors is recognized.

A lot of work has gone into the library and the preparation of this documentation. The cost of manuals and software needed to keep extending Waterloo TCP comes from 'donations' cleverly disguised as purchases of this documentation.

Please note: this manual is copyrighted, you may not copy it without written permission by the author. If you are interested in using this manual as part of a course, please contact me for a per copy license fee for the manual.

If you have any requests or comments, please feel free to contact me by mail or E-mail.

If you would like to donate a program, especially in source, I would be pleased to add it to our distribution with full acknowledgement.

Erick Engelke
Waterloo TCP Architect
Erick@development.watstar.uwaterloo.ca

How To Read This

This guide is written in a structured format, not an alphabetical list of functions, that list is in the reference section. I was attempting to give some background before unleashing a disorganized list of functions. I will occasionally mention "see ...", that is for later reference, bear with me sequentially.

I am open to comments and suggestions for both the software and this manual. The first few releases of the manual will tend to be wordy and some sections redundant. With your comments, and more time to reflect, I hope to trim down and clarify the documentation.

Preliminaries

For any programs you write with Waterloo TCP, include **TCP.H**. It has all the necessary function headers and macros. I will continue to distribute new TCP.H files with each new release of the kernal, so you are recommended to not change this file for personal use.

The **WTCP_VER** flag will be incrementing, indicating things have changed. I will distribute **CHANGES.n** (eg. CHANGES.105 when version 1.05 is released) to describe the effects.

I hate typing in unsigned this and unsigned that. The convention I have used here is *byte, word*, and *longword* for the unsigned quantities.

During the design of the APIs, I eliminated every mechanism which would require more memory. The result is a kernel and applications which consume less memory than most kernels alone. That also meant a single stack, hence a single task, no callocs or mallocs in the kernal (since some programs will be TSR's) and efficiency over absolute cleanliness whenever possible. That's the way to do operating systems. The application programmers may do as they please, but you are reminded that the APIs described here will give fast and small code. Do not feel that my single task restricts you to a single task, the appendices give details on how to effectively use multitaskers with Waterloo TCP.

When you program, you must use my language for this project, Turbo C. Everything is currently small model. *You cannot use register variables* so set that in your make files or tcconfig.tc files right now! For TCC users, set the **-r-** flag. For integrated environment users, select Options/Compiler/Registers/Off.

Nutshell: How it works

The kernel is mostly a non-blocking system. It is also not tied to the timer tick for processing. You must explicitly call it occasionally to do all the tcp/ip house keeping.

To make Waterloo TCP a little more friendly, I added macros and routines which make the kernel look like a blocking, callable resource, basically like MS-DOS or UNIX. Since most programs end up waiting on events like data arriving, the whole thing looks like a blocking system.

Important Concepts

Sockets

Waterloo TCP uses what I call sockets. Enough other people call other things sockets that I don't mind adding yet another confusing definition. For the rest of this manual, what I call sockets will apply to my sockets, and not necessarily to any other programming environment.

A socket may be defined to be of type *tcp_Socket* or *udp_Socket*. You will have to use one of the functions **tcp_Open**, **udp_Open** or **tcp_Listen** to open the socket, but subsequently any routine of type **sock...** will work.

Note that I have made many functions non-blocking, this means you can do things between functions. That provides power but may sacrifice simplicity, so I have added blocking macros which make everything look a little more similar to UN*X or other systems. There are also blocking functions available for you to use. See **sock_...** functions for more details.

An optional *signal_handler* may be passed during the open or listen functions. Please use NULL until such time as this service is properly announced.

ASCII vs. BINARY

'C' programmers are used to ASCII and binary oriented file i/o. To simplify some situations, I have added a mode flag to the socket structure and several functions act upon it. If you intend to use ASCII oriented instructions, call **sock_mode** with the socket in question and the flag **TCP_MODE_ASCII**. You can return to binary mode at any time by calling **sock_mode** again with the flag **TCP_MODE_BINARY**. *Sockets always open in binary mode, so you must explicitly indicate to use ASCII mode.*

void **sock_mode(** *void* ***s,** *word* **mode);**

eg.
```
if ( !tcp_open( s, 0, host, port, NULL )) return(0);
sock_mode( s, TCP_MODE_ASCII );
sock_wait_established( s, sock_delay, NULL, &status );
```

The mode flag affects **sock_dataready** and **sock_wait_input** because they normally respond as soon as any data is available. In ASCII mode, these functions act as if no data is available until a complete string is ready to be transferred. In ASCII mode, the operation of **sock_puts** changes slightly as it appends a '\n' to the sent data, normally no '\n' is added. Remember, if you intend to use **sock_gets** or **sock_scanf**, you *must* set ASCII mode.

No other functions are affected.

Kernel Initialization

The kernel must be initialized before it is used, likewise it must be shut down before your application exits. Failure to shut down, for example, would cause you machine to hang the next time a packet arrived for your application.

The easiest mechanism is to use **sock_init()**. It reads the configuration file, installs a control break handler, sets up the automatic shutdown facility using **atexit()**, checks the packet driver, loads the Ethernet address for ARP stuff, and clears tables. Note that **sock_init** removes the need for you to call a routine before you exit *if* you use control break, **exit()**, or abort, retry, ignore. *(the abort, retry, ignore description is not correct at the date of publishing. Such an exit leave the system in an unknown state)* *void* **sock_init(** *void* **);**

For details on extending the configuration file, see the reference section under sock_init and usr_init.

Socket Functions

Opening Sockets

*Note, you cannot open a socket until after you have called **sock_init()**.*

Sockets are formed using local and possibly remote IP addresses and sessions. All references hereafter will use:

lport - local port number, must be defined for **tcp_listen**
port - remote port number, must be defined for **tcp_open**
ina - remote host internet address, necessary for **tcp_open**

In the *tcp* realm of thinking, there is usually one host who listens for a caller and one who calls although *tcp* inherently supports two mutual callers.

The listener must provide a local port on which he will be waiting and may or may not define the remote ip and/or remote port. Whenever the remote port or ina is not defined (i.e. 0), a matchall value is used and incoming sessions are not screened by those parameters. Remember, the completion of all necessary parameters does not constitute a session, the call must still be received.

For *udp*, the remote port and/or internet address may be defined. If the remote host is zero, a received packet will complete the socket, if the remote host is non-zero, the socket will transmit to the defined node or network upon the next sock_ writing function. **udp_open** uses ARP to find the correct network address.

tcp_open actively attempts to connect to the other machine and uses ARP to find the machine address.

tcp_open and **udp_open**, if called with an lport of 0, will dynamically assign a generic port value.

tcp and *udp* sessions are unrelated, separate sockets exist for each of the two even if the local port numbers are identical.

All open functions return zero on error such as being unable to resolve the other machine's hardware address. They also accept *signal_handler* as described under the topic "sockets". tcp_Listen accepts a timeout value after which inbound socket attempts will fail.

word **tcp_open(** *void* *s**,** *word* lport**,** *longword* ina**,** *word* *port**,** *int (**signal_handler*)***);**
word **tcp_listen(** *void* *s**,** *word* lport**,** *longword* ina**,** *word* *port**,** *int (**signal_handler*), word* **timeout);**
word **udp_open(** *void* *s**,** *word* lport**,** *longword* ina**,** *word* *port**,** *int (**signal_handler*)***);**

After a session open call (**tcp_open** or **tcp_listen**), you must wait until the connection is established. There is a minimum of three packet transmissions, so the connection is hardly immediate. You may either poll the connection (**sock_established**) or have the library perform the waiting (**sock_wait_established**) and also the error handling if a timeout or reset occurs, see sock_... functions for more details on **sock_wait_established**.

word **sock_established(** *void* ***s);**
void **sock_wait_established(** *void* * s**,** *word* seconds**,** *word (**optional_fn*)()*,**word* *optional_status_ptr**);**

Closing Sockets
Once a socket has been opened, you will wish to read and write data. Upon completion of your data exchange you should use **sock_close** to close the connection. For tcp sockets, you must either use **sock_wait_closed** or poll the connection as described under **sock_wait_closed**. Ctrl brk or **exit()** will also close connections. *I do not think that abort() keep() or _exit() will close connections.*

void **sock_close(** *void* * **s);**

Waterloo TCP does not use keep-alives. They are highly discouraged by the internet community. A socket is assumed alive until closed, remotely reset, the program interrupted, or a user determined timeout occurs.

Reading Data

> *All reading functions return -1 when an error is encountered although I have not repeated it each time below. You should use signed len quantities. See* **sock_wait_input()** *for error handling.*

The two basic socket read functions are **sock_read** and **sock_fastread**. They both take the same parameters, but **sock_read** waits until the user buffer is full while **sock_fastread** returns immediately. **sock_dataready** returns the number of bytes ready to be read or -1 on error. It may be polled while waiting for data, and in ASCII mode (as set by sock_mode), it waits until a complete line is in the *tcp* buffer.

int **sock_read(** *void* *s**,** *byte* *dp**,** *int* len **);**
int **sock_fastread(** *void* *s**,** *byte* *dp**,** *int* len **);**

sock_gets reads a string and removes the lf or cr. The length of the new string is returned. If the new string is len or longer, the string is NULL terminated at len bytes and the remainder is discarded. If you intend to use **sock_gets**, you must first set ASCII mode using **sock_mode**. See the section on ASCII vs. Binary.

int **sock_gets(** *void* *s**,** *char* *text**,** *int* len **);**

sock_getc reads the next char if there is one, returns -1 on error, or 0 if there were no characters waiting. The limitation is that **sock_getc** cannot be used to read \0 or \1 unless you first check for data presence and connection stability which can be done with **sock_dataready()**:

```
if (sock_dataready( &socket )) {
        ch = sock_getc( &socket );        /* know connection stable and data present */
        printf("Got %u \n\r", ch );
}
```

char **sock_getc(** *void* *s **);**

sock_scanf busy waits for data and returns -1 on error or the number of fields completed. This function first performs an intrinsic **sock_gets**, so data must arrive on a single line. See the notes under the ASCII vs. Binary section.

int **sock_scanf(** *void* *s**,** *char* *format**,** ... **);**

To handle some tricky situations, sock_PreRead works exactly like sock_FastRead except it does not clear the data from the buffers. This technique may be used to prevent double buffering of received data.

int **sock_preread(** *void* *s, *byte* *dp, *int* len **);**

Writing Data

Writing to a socket places the bytes on the queue, but they may not necessarily be transmitted immediately (unless udp_Open was used), or the data may be lost in the transmission, or the other system may not immediately pass the data on to the user (unless udp_Open was used).

Most of the write functions do not transmit until a certain number of bytes are ready to send, or until a certain period of time has passed. Those values are either system constants, or determined using algorithms dependant upon the particular connection. All write functions except **sock_fastwrite** block until sufficient buffer space exists for the kernel to hold the remainder of the data. **sock_fastwrite** simply returns the number of bytes it was able to write and the user must loop through until all the data is written.

The basic socket writing function is **sock_write**. I have provided two additional functions which flush data buffers and instruct the remote computer to pass the data to the user. **sock_flush** ensures the data already written is transferred, and **sock_flushnext** affects the next data to be written. **sock_flushnext** is preferred because it is much more efficient.

int **sock_write(** *void* * s**,** *byte* *data**,** *int* len **);**
int **sock_fastwrite(** *void* * s**,** *byte* *data**,** *int* len **);**
void **sock_flush(** *void* * s **);**
void **sock_flushnext(** *void* * s **);**

sock_puts will put a null terminated string. It performs an intrinsic flush (sock_flushnext). If in binary (default) mode, the string is not changed, but if ASCII mode has been set with **sock_mode**, the string and a subsequent \n are written. In either case, length of the original string is returned.

int **sock_puts(** *void* *s**,** *char* *text **);**

sock_putc places a character on the queue. If the character is a CR or LF, the stream is intrinsically flushed. There is no character expansion performed. The character written is returned.

int **sock_putc(** *void* *s**,** *char* ch **);**

sock_printf writes the formatted string using **sprintf** and **sock_puts** and returns length.
sock_printf(*void* *s**,** *char* *format, ... **);**

Buffer Information

While the input and output buffers are not truly available to you, there are several routines which allow you to inspect the size of the buffers (sock_tbsize, sock_rbsize), the number of bytes currently used in the buffer (sock_tbused, sock_rbused), and the number of bytes not used (sock_tbleft, sock_rbleft). All values are < 32767 and in bytes. **sock_t**... functions refer to transmit buffers, while **sock_r**... functions refer to receive buffers. *udp* sockets will return 0 for all transmit buffer functions since they do not maintain a transmit buffer. Closed sockets should return 0 for all such functions.

int **sock_tbsize(** *void* ***socket);**
int **sock_tbused(** *void* ***socket);**
int **sock_tbleft(** *void* ***socket);**
int **sock_rbsize(** *void* ***socket);**
int **sock_rbused(** *void* ***socket);**
int **sock_rbleft(** *void* ***socket);**

Kernel Waiting

User applications may do polling to wait for data, but unless you are expecting real-time data (keystrokes) or handling multiple sessions, you will want to have simple mechanisms to do your busywaiting.

I have already written a few macros/functions to perform this task for most simple situations, they are macros called **sock_wait_...** These macros have several things in common, they accept a timeout value for the current operation and they routinely give the kernel a time slice, do a user defined yield if **sock_yield** was used, check the keyboard for ^C (remember we safely close sessions), and call an optional short user defined function repeatedly. They also accept a status pointer and handle errors so your input and output routines do not have to.

sock_wait_established is used after an open to wait until the connection is complete. For *udp* connections it returns immediately. **sock_wait_input** waits until some input arrives. **sock_wait_closed** is used after a **sock_close** operations, but is unnecessary after a **sock_abort**. **sock_tick** just performs a kernel slice, but replicates the error handling of the other **sock_...** functions.

sock_wait_established(*void* * s**,** *word* seconds**,** *word (**optional_fn*)**,** *word* *optional_status_ptr **);**
sock_wait_input(*void* * s**,** *word* seconds**,** *word (**optional_fn*)**,** *word* *optional_status_ptr **);**
sock_wait_closed(*void* * s**,** *word* seconds**,** *word (**optional_fn*)**,** *word* *optional_status_ptr **);**
sock_tick(*void* * s**,** *word* seconds**,** *word (**optional_fn*)**,** *word* *optional_status_ptr **);**

Although not considered pretty, (what systems programming is?) the sock_... functions branch on an error by setting *optional_status_ptr if it is non-NULL and doing a **goto sock_err;** The result is much simpler user code. Performing the same with **break**s and **returns** creates much larger modules. Using upcalls looses context sensitivity. Obviously the other two mentioned options exist, they are simply computer science imposed cleanliness. If you don't agree, look at the assembler code your compiler creates on a do while loop.

Name Server

The name server allows users to specify names rather than dotted ip address. Usually you will just need to use **resolve** and it will correctly handle names or dotted ip address formats.

longword **resolve(** *char* * **host_string);**

To save exe size, you may insist on dotted ip format only. Use inet_addr which returns an ip number in longword format, or 0 on error:
longword **inet_addr(** *char* ***dotted_ip_string);**

The kernel uses the following constants which are automatically filled in by sock_init. You may read def_nameserver and def2_nameserver, but writing to them is forbidden, particularly once the TSR version is released.
extern char ***def_domain;**
extern longword **def_nameserver, def2_nameserver;**

What You Have Learned

You have learned almost everything which you will need in order to program TCP/IP applications using Waterloo TCP. The remaining functions provide additional features, but are not really as important. You might like to look at some of the sample applications in the reference section or in the software distribution and see how they work.

When you are resonably familiar with their basic layout, return to this section for a brief description of the rest of Waterloo TCP.

Nagle Algorithm

The performance and bandwidth costs of TCP can often be improved by using a simple and interesting algorithm commonly referred to as the Nagle Algorithm.

The Nagle algorithm instructs us to not launch many small packets, but to store up new data until it can piggyback an acknowledgement, or it reaches a maximum size, or the previous ack timed out.

For slow lines, Nagle reduces the number of headers transmitted which in turn reduces the ratio of $^{headerbytes}/_{databytes}$, which improves channel efficiency and will free up the channel for other uses. Without Nagle, Telnet keystrokes invoke a flurry of small, wasteful packets. With Nagle, characters seem to 'bunch up', but the actual performance is identical.

To assume that Nagle is only helpful over slow media is actually quite incorrect. TCP can be costly over any network media, even Ethernet or FDDI where bandwidth may not be a problem. Consider the cost of having your UNIX box receive many small packets from user TELNET sessions. If twenty secretaries are typing, say each at 80 words per minute. Those 133 packets per second are a nusance because they use up your limited ethernet buffers as well as lots of processing time for interrupts, acknowledgements, tcp protocol updates, etc. Over a local Ethernet, Nagle only has an influence when response times are slow, namely when the network is congested or the server is very busy. Clearly those are exactly the times we wish to reduce redundant packets, so Nagle is again appreciated.

These two scenarios explain why Nagle is on by default, but Waterloo TCP allows you to disable it with a call. Some applications, such as X-Windows and real time data collection need responsiveness enough to warrent being traffic hogs. Neither application is suggested to be used over anything but local subnets if that gives you any idea of the importance.

sock_mode(*void* * **socket,** *const* **flag);**
eg. **sock_mode(s, TCP_MODE_NAGLE);**
sock_mode(s, TCP_MODE_NONAGLE);

You can turn it on or off as often as you like, thereby allowing you to gain the benefits of Nagle, while temporarily disabling it to permit truely urgent data to be handled immediately.

Note: The Nagle algorithm has no effect on UDP sockets.

UDP Checksums

udp does not necessarily require checksums. According to the specifications, *udp* checksums should be enabled by default. Many systems will only support *udp* checksums on for all connections or off for all connections. Waterloo TCP defaults to checksums but allows you to override that decision on a socket by socket basis.

Incomming data is always checked if the remote computer uses checksums. Disabling local checksums speeds up both local and remote processing of datagrams.

A good reason to disable checksums is if your program implements its own checksum or you *know* that the protocol is being run over networks which implement checksums or cyclic redundancy checks. Remember, if the remote computer is on a different subnet, data may be passing through networks with no implicit checksum or other error detection.

sock_mode(*void* * **socket,** *const* **flag);**

eg. **sock_mode(s, UDP_MODE_CHK);**
sock_mode(s, UDP_MODE_NOCHK);

You can turn checksums on or off as often as you like, thereby allowing you to gain the benefits of checksums for reliable data while gaining speed over unimportant data.

Note: The checksum flag has no effect on TCP sockets.

UDP Record Services

As discussed earlier, Waterloo TCP can work with *udp* connections just as easily as *tcp* connections, in fact they both seem to act identically. Of course this appearence is actually quite ficticious. The basic sock read functions (**sock_read**, **sock_fastread**, etc.) are not adequate for all your *udp* needs. The most basic limitation in their is inability to treat *udp* as a record service.

By record service I mean you will receive distinct datagrams which should be kept and distinct. You will wish to know the length of the received datagram and the sender (if you opened in broadcast mode). You may also receive the datagrams very quickly, so you must have a mechanism to buffer them.

Rather than add clumsy appendages to the standard socket reading functions to handle these new requirement, I elected to add new functions specifically for record service.

Once a socket has been openned with **udp_open**, you can use **sock_recv_init** to initialize that socket for the following functions. Note that **sock_recv** and related functions are *incompatible* with **sock_read**, **sock_fastread**, **sock_gets**, **sock_getc** and **sock_scanf**. Once you have used **sock_recv_init**, you can no longer use the older style calls.

sock_recv_init installs a large buffer area which gets segmented into smaller buffers. Whenever a *udp* datagram arrives, Waterloo TCP stuffs that datagram into one of these new buffers. The new functions scan those buffers. You must select the size of the buffer you submit to sock_recv_init, you should make it as large as possible, say 4, 8 or 16 Kbytes. **sock_recv_init** returns 0 on success.

sock_recv scans the buffers for any datagram received by that socket. If there is a datagram, the length is returned and the user buffer is filled. If no datagrams were found, sock_recv returns 0.

sock_recv_from works identically to **sock_recv** with the addition that it places the source *ip* address in a user location pointed to by *fromptr*. This allows you to write a server using *udp*. To reply, you must open a *udp* socket with the *udp_open* function. There is no facility in Waterloo TCP to simply send **udp** datagrams without a socket.

int **sock_recv_init(***void* ***socket,***byte* ***buffer,***int* **bufferlen);**
int **sock_recv(** *void* ***socket,** *byte* ***buffer,** *int* **maxlength);**
int **sock_recv_from(** *void* ***socket,** *byte* ***buffer,** *int* **maxlength,** *longword** **fromptr);**

```
#include <stdio.h>
#include <tcp.h>
#include <newstruct.h>

#define SAMPLE 401


udp_Socket data;
char bigbuf[ 8192 ];

main(int argc, char *argv[])
{
   word templen;
   char spare[ 1500 ];

   sock_init();
   if ( !udp_open( &data, SAMPLE, 0xffffffff, SAMPLE, NULL) ) {
        puts("Could not open broadcast socket");
        exit( 3 );
   }

   /* set large buffer mode */
   if ( sock_recv_init( &data, bigbuf, sizeof( bigbuf ))) {
        puts("Could not enable large buffers");
        exit( 3 );
   }

   sock_mode( &data, UDP_MODE_NOCHK );        /* turn off checksums */

   while (1) {
        tcp_tick( NULL );
```

```
            if ( templen = sock_recv( &data, spare, sizeof( spare ))) {
                /* something received */
                printf("Got %u byte packet\n", templen );
            }
            kbhit(); /* check for control breaks */
        }
    }
```

General User Functions

psocket will print a remote host and port to standard output
void **psocket(** *void* *__socket);__

sockerr returns an ASCII string containing the last error message found and believed to be the cause of the current problem.
char ***sockerr(** *void* ***socket);**

sockstate returns a brief ASCII string containing the current state of the connection. For *udp* sockets, the state is always "UDP Socket", since UDP is a connectionless protocol and has no state.
char ***sockstate(** *void* ***socket);**

getpeername will return the remote computer's IP address. **getsockname** returns the local port number. These are UNIX similar functions, see the reference section for more details and differences between these functions and their UNIX counterparts.
void **getpeername(** *void* ***socket,** *struct sock_addr ** **p,** *word* ***length);**
void **getsockname(** *void* ***socket,** *struct sock_addr ** **p,** *word* ***length);**

gethostid() returns the IP address of the machine, **sethostid()** sets the IP address.
longword **gethostid();**
longword **sethostid();**

getdomainname fills the user supplied string with the current domain name. A pointer to the string is also returned. **setdomainname()** sets the domain name to the passed string and returns the pointer to that string.
char ***getdomainname(** *char* ***user_buffer);**
char ***setdomainname(** *char* ***string);**

ntohs, **htons**, **ntohl** and **htonl** convert host to network byte ordering.
word **ntohs(** *word* **value);**
word **htons(** *word* **value);**
longword **ntohl(** *longword* **value);**
longword **htonl(** *longword* **value);**

rip will remove cr's and lf's from the string and return a pointer to the string.
*char ** **rip(** *char ** **string);**

To handle control breaks correctly, (remember we get passed interrupt calls) Waterloo TCP automatically installs a control break handler. You may change the mode of the control break handler with **tcp_cbreak** or you may write your own handler.

If you use **tcp_cbreak**, *mode* is a combination of 0x01 disallow ctrl breaks, and 0x10 display a message on ctrl breaks.
sock_init calls this with mode 0x10 which displays a message and then aborts the program.

void **tcp_cbreak(** *word* **mode);**

If you use your own control break handler, remember to call **sock_exit()** with no parameters to release the packet driver and to close all sessions.

Kernel Accessible Routines

If you are doing your own delaying functions, you will need to be able to tick the kernel. **tcp_tick** returns almost immediately and will return 0 if the optional socket passed has been closed, timed out, or reset by the other host. The full operations of **tcp_tick** are described in the "indepth understanding" chapter.

word **tcp_tick(** *void* ***s);**

The macro **sock_tick** calls **tcp_tick** and then acts similarly to the **sock_wait_...** functions except that it does not perform any waiting. See the earlier section on "Kernel Waiting" (page 9).

Relatively little in the way kernel debugging facilities is available in the basic system, but what is provided may be turned on using **set_debug_state**. However, TCPDBUG.EXE is a better form of debugger. See the chapter on debugging.

void **set_debug_state(** *word* **state);**

eg. **set_debug_state(0);** turns off debugging
set_debug_state(1); turns on basic debugging

If you are debugging your code and need to see some error termination message, call **sock_debugdump** with your socket. Also described in the debugging chapter.

void **sock_debugdump(** *void* ***s);**

eg. **sock_debugdump(&socket);**

A user-defined yield function will be called on any kernel waiting functions if you set the function with **sock_yield**. If parameter s is NULL, yield_fn will be inherited by all subsequent open or listen commands, otherwise s must point to a specific socket and yield_fn is applicable only to that socket. An obvious use for **sock_yield** is to add multitasking features. Hopefully, someone will release a public domain version of their tasker in this application. See the appendix on adding taskers.

void **sock_yield(** *void* ***s,** *void (****yield_fn*))***;**

eg. 1

```
fn() { printf("WAITING"); }
main()
{
    tcp_Socket socket;
    sock_init();
    sock_yield( NULL, fn );
    if ( !tcp_open( &socket, ...
```

eg. 2

```
finger( host, port )
{
```

```
tcp_Socket socket;
sock_init();
if ( !tcp_open( &socket, 0, host, port, NULL )) exit( 0 );
sock_yield( &socket, fn );
...
```

Notes: **sock_yield(NULL,** ... does not affect functions which are already open
sock_yield(s, NULL) or **sock_yield(NULL, NULL)** can be used to reset the appropriate sockets so that no yield function will be called.

There are two user accessible timer routines. All work under the same premise, you supply a value in seconds and save the new time, later you make successive calls to determine if the timer has timed out. Day wraps are handled and timeouts up to about 18 hours may be selected.

The **sock_wait...** timers use the following with 0 meaning never timeout
void **ip_timer_init(** *void* ***s,** *word* **seconds);** to set the timer
word **ip_timer_expired(** *void* ***s);** to check the timer, 0 on not timedout

You may use as many user defined timers as you wish since you will hold the timer values. *Kludge: if you must, you can adjust the timeouts by adding or subtracting constants to the value returned from set_timeout. Add 1L for each $^{1}/_{18}$ second. Eg. set_timeout(1) - 17 would specify a timeout of about 55 milliseconds.*

longword **set_timeout(** *word* **seconds);**
word **chk_timeout(** *longword* **timeouttime);**

eg.
```
longword timeout;
timeout = set_timeout( 20 );
while ( ! chk_timeout( timeout ))
        printf("not timed out yet");
```

ICMP and PING

ICMP (Internet Control Message Protocol) is the standard method for Internet hosts to send control messages. Although the ICMP protocol is technically sent using *ip* (much like *tcp* and *udp*), it is considered a required part of IP rather than another layer. IP uses ICMP to complain, query and respond at a level lower than *tcp* or *udp*. Since ICMP is usually implemented as a part of IP, it usually runs at the same high priority as IP and is not typically bogged down even when the *tcp* and *udp* protocols are very busy. One excellent use of this fact is the 'PING' program.

PING is the standard name for the standard method of determining whether another computer is alive, and if so, for computing the round-trip-time (RTT) between the two computers. PING works by sending an ICMP echo request packet to the other host. The remote computer responds, usually as fast as possible, and the original computer records the difference in time.

Waterloo TCP supports ping with two functions: **_ping** which generates the request and **_chk_ping** which should be used periodically to check the ping cache for a response from that particular host. The ping cache has length one and is overwritten on each incoming ICMP echo response (which posted during **tcp_tick** or **sock_tick**) and nullified on each successful **_chk_ping** request. **_chk_ping** returns 0xffffffff if the cache does not contain a new entry for that host, or the number of *ticks* between the transmission and response receipt of the currently returned sequence number. Failure to call **tcp_tick** (or its counterpart **sock_tick**) or **_chk_ping** very frequently will result in lost receipts due to the cache size.

word **_ping(** *longword* **host_ip,** *longword* **sequence_number);**
longword **_chk_ping(** *longword* **host_ip,** *longword* *****sequence_number);**

For more details on how to use the ping functions, look at the example program **PING.C**.

Undocumented Calls

Only the calls listed *in this manual* (and extensions in future revisions of the manual) are supported. Undocumented calls may have parameters change or functionality changes in future revisions. Likewise the *tcp_Socket* and *udp_Socket* structures will change. I intend to offer a service similar to setbuf to allow you to use larger *tcp* windows.

In the future, the kernel may be moved to a TSR which will support multiple EXE's. This support will help with WINDOWS, DESQVIEW, and simply when user's shell out of TELNET and want to run LPR or something. All undocumented functions would be totally unavailable at that time.

Distribution

Latest copies are always available from anonymous FTP to **SUNEE.UWaterloo.ca** under **/pub/wattcp**. You may place new files in **/pub/wattcp/uploads** and send me mail. The current address of sunee is [129.97.128.196], but it is subject to change within this year sometime.

A mailing list exists, namely **wattcp@development.watstar.uwaterloo.ca**. E-Mailing to my address shown below with an appropriate message will add you to the list.

Anyone is free to use the kernel and associated programs, that included companies, organizations, governments, institutions, etc. You may share programs with others and are encouraged to do so, but you may also sell the programs. I will gladly help distribute your free programs.

There is no risk with using this kernel, its free distribution will not change.

If you are interested in using this product for a course, please contact me.

Also, mail to me or the group with ideas.

June 12, 1991
Erick Engelke
Waterloo TCP Architect

E-mail: Erick@development.Watstar.UWaterloo.ca

c/o Watstar Development Group
Faculty of Engineering, University of Waterloo
200 University Rd., Waterloo, Ont.
Canada N2L 3G1

My home address is:
1010-130 Lincoln Rd.
Waterloo, Ont., Canada.
N2J-4N3

Phone (519) 886-0405 and talk to my answering machine. If you phone, leave a return phone number and an E-mail address. Reaching me at my office is a lost cause!

Indepth Understanding
In this section, I simply went through my source code and listed what I saw. By no means is this chapter essential for programming with Waterloo TCP. However, if my descriptions are sparse elsewhere, this might give you a feel for what is going on.

The three open commands, **tcp_open**, **udp_open**, and **tcp_listen** start by zapping your buffer space and putting it to a linked list. In the case of the **..._open** commands, the destination address is checked for accessibility. If it is a local broadcast, the hardware broadcast is selected, otherwise local tables or subsequently *ARP* is used to resolve the address. A failure on the resolution of the address at this point will mean the open function fails. **tcp_open** also sends the first datagram in an attempt to open the tcp connection.

As you may recall, **sock_yield** is used to specify a global yield function. If **sock_yield** has been used, all subsequent open and listen calls inherit that yield function by default. An explicit call to **sock_yield** may be used to remove or replace the yield function for a specific socket. **sock_yield** itself merely updates the pointer in a particular socket or the pointer in a system variable which is used on future opens and listens.

The **sock_close** function identifies the socket type. In the case of tcp, a *fin* is attempted and subsequent calls to **sock_wait_closed** or a similar service must be made so the close can be completed or timedout. In any case, once a socket is truly closed, (immediate for udp) the function removes the socket from the appropriate list and sets a value in the buffers so future i/o should fail.

tcp_tick is the basic call to the tcp kernel. It performs everything which can be done at the current time, then returns immediately.

First it checks the receive queue for packets and demultiplexes the datagrams among the tcp, udp, icmp and arp protocols and the handler for the protocol performs the necessary operation on the data. For tcp, the connection state is updated, the statistics are updated for the retransmission strategy, and an acknowledgement datagram is sent immediately. The ack piggybacks any new, untransmitted data and include the new window size available for the remote tcp to fill. In the case of udp, the new data is appended to the buffer if space permits. Arp receipts update the local arp table and generate a response if we are being queried. ICMP messages are displayed. If the message indicates an error, a pointer to the message is placed in the socket structure and an attempt to correct the situation is made. User applications may read the message by issuing a call to **sockerr**.

After all incoming packets are processed, the retransmitter is called. **chk_timeout** is used to ensure retransmissions for any socket are made not more than once per second. For each open tcp connection with pending data, it checks the last transmission time and recommended re-transmission time. If appropriate, a new datagram is sent. If the user previously called sock_close on a tcp socket, the socket is checked for a close timeout.

The read and write functions can be grouped as to blocking and non-blocking functions. The non-blocking functions obviously return immediately. The blocking functions repeatedly call **tcp_tick** to handle incoming datagrams, and the user defined yield function if one had been provided with **sock_yield**. Non-blocking functions may or may not yield.

All read functions transfer data then update the buffer to reflect the new space. In the case of tcp, reading data affects the available data window. Adjustments to the window size are placed in the socket structure but are not immediately transmitted. At the next sending of data, or after a small timeout period, the new window size will be automatically transmitted. *(At the date of publishing, the window size is immediately retransmitted which may lead to the well-documented silly-window syndrome.)*

The write functions are dependant upon the local transmit buffer size available and not the tcp window. An attempt is made to immediately write the data if the buffer size is sufficiently large or the user has specified to do so with **sock_flushnext** or **sock_flush**.

sock_printf and **sock_scanf** significantly increase the size of executable and may cause problems in TSRs.

The functions **sock_rbsize**, **sock_rbused**, **sock_rbleft**, **sock_tbsize**, **sock_tbused** and **sock_tbleft** merely check the socket type and return the values from the structures. No other processing is performed.

The name resolution functions, available through **resolve** will return immediately if the passed argument is a numeric dotted ip address. Otherwise the primary name domain server will be queried. Failing resolution, the secondary nameserver will be consulted. **resolve** blocks until the name is resolved. Normal packet processing continues since **tcp_tick** is also called constantly. This is one instance of the default yield function being over-ridden, **resolve** does not honour yield functions declared with **sock_yield** because it uses a static buffer space.

The address resolution protocol handler caches a table of arp related data. Currently the cache size is 20. Calls to **_arp_resolve(** *longword* **ip,** *void* ***hardware)** attempt to get a hardware address for the destination ip address. First the cache is checked, then, if the ip address is on our subnet, an arp request is made. Otherwise the gateway is assumed and the address is found using a recursive call to **_arp_resolve** with the gateway's ip address. **_arp_resolve** returns with zero if it times out, or a non-zero value if the address is found. Typically arps are fast. During the response delay, **tcp_tick** is repeatedly called to handle incoming packets and manage other sessions. The optional yield function set with **sock_yield** is not called because **_arp_resolve** is too low in the kernel, arps are rare, and the response should be fast enough to not be a problem.

Many socket functions must determine if the passed structure is a valid open socket. They do so by calling **_chk_socket(** *void* ***s)**. The returned values are 1 for udp, 2 for tcp, and 0 for failure. A text string indicating the status of a connection may be found using the **sockstate** function.

The bottom two levels of code are the packet driver and Ethernet handlers. The "Ethernet" code simply encapsulates outgoing packets and removes headers on received packets, it does not manage the ip to hardware translation, that is performed in the arp code and managed by ip. The packet code performs sends, receives, initializes the packet driver, resets the packet driver and reads the hardware address. All outgoing frames are built in a single frame buffer as they are needed. Incoming frames are placed into one of the buffers during the interrupt and processed inside **tcp_tick**. *The current implementation is limited to 5 buffers, each of length 1500 bytes.*

set_debug_state can be used to enable or disable the internal tcp debugger code. Most of the messages are to help me solve kernel problems rather than user-level debugging. Whether you find the messages useful or not is another question. In **PING.EXE**, for example, the debugger will transmit the destination ethernet address. For debugging your code, use the TSR, **TCPDBG.EXE** (release date, March 15, 1991).

The initialization code in sock_ini.c calls pcconfig.c to open the configuration file, read the contents, parse the lines and process them. Application programmers, of course, may use the **usr_init** hook to extend the configuration file.

If the attempt to open the config file failed, or if BOOTP was selected, functions in pcbootp.c get called. They open up a simple udp session, defaulting to broadcast unless otherwise configured, and pump out bootp requests with random durations between. The randomness is intended to allow several machines to all boot up at the same time, but to not synchronize their requests. The randomness is partially dependant upon the millisecond clock and the ethernet address of the network card.

The **sock_recv** feature is located in pcrecv.c. The udp socket structure's data buffer is trampled upon since the two sock_recv and normal input functions are incompatible. Each time incoming data is found during a tcp_tick(), the udp_handler function processes the packet and decides whether to place the data in the usual small buffer, or to do an upcall to the udp socket's datahandler function which **sock_recv_init** changed to be its own **_recv_daemon**. **_recv_daemon** scans the available buffers and places the packet in a free buffer if one exists.

Using Multitaskers with Waterloo TCP

Oddly enough, while the MIT tcp kernel started with an idea and a multitasker, the Waterloo TCP kernel does not have a resident tasker. For other projects I have written my own tasker and used other peoples as well. For this particular situation, however, I decided to leave the tasker up to the user. If you are familiar with Desqview or whatever, either check the distribution site or help writing whatever is necessary

All blocking read, write, and **sock_wait...** functions are programmed using loops which repeatedly call the kernel **tcp_tick** function. Also included in the loops are a call to the yield function for that particular socket. The yield function may be set on a per-socket basis, or on a system-wide basis with the **sock_yield** command.

void **sock_yield(** *void ****socket,** *void (****yield_fn***)***);**

To simplify a multitasker environment, you may call **sock_yield** immediately after **sock_init** with the socket parameter set to NULL and the function parameter set to your yield function. All subsequent opens and listens will inherit that yield function automatically. Note that **sock_yield** may be called with socket set to NULL at any time, but it only affects future opens and listens, not currently opened or listening sockets.

You may set the yield function for a particular socket by opening the socket with either an open or listen command, then calling **sock_yield** with a pointer to that socket and to the yield function to be called. This is useful to turn off the tasker for a specific socket, such as ones which use static data structures rather than stack based structures. The **resolve** function falls into this category though it could certainly be re-written. *Note: if you wish to not perform yields, you should call* ***sock_yield(&socket, NULL);*** *immediately after the open function.*

If the yield function is set to NULL, the kernel is instructed to not call any yield function.

eg.

```
#include <tcp.h>
void yield() { printf("WAITING,");}
main(int argc, char **argv)
{
    tcp_Socket socket;
    longword host;

    sock_init();
    sock_yield( NULL, yield );
    if (argc < 2 ) exit(3);
    host = resolve( argv[1] );
    if (! tcp_open( &socket, 0, host, 25, NULL)) {
         puts("unable to open socket");
         exit( 3 );
    }
    sock_yield( &socket, NULL );
   ...
```

The example above demonstrates how to set all sockets to use the function 'yield', then goes on to create a socket which will *not* use yield.

The kernel will not call the yield function at any critical areas as long as the tasker is not pre-emptive.

Configuration Files

sock_init() is the normal mechanism to start up the kernal, and this involves reading the configuration file.

The data in the configuration file is organized as

[*directive*= ["*data*" | *data*]] [#*comment*|;*comment*]

eg. **netmask=255.255.254.0**
netmask=255.255.254.0 # 23 bit subnet
domainlist=waterloo.edu

If a directive is not followed by data, the directive is ignored. Similarly, lines without directives are ignored. For all internal processing the directives are capitalized while the data is left case sensitive.

Data is grouped as quoted data or normal data with the former simply being surrounded by quote marks.

Quoted data may contain any character except quotes. If a line of quoted data does not contain a terminating quote, the newline character ends the quotation.

Normal data (non-quoted) has white space removed and will be immediately terminated by the start of a comment which is indicated by **#** or **;**.

To compare the two types of data, the PRINT directive (which simply print

You can add your own directives to the configuration file and Waterloo TCP will use upcalls to your routines with pointers to "directive" strings and "data" strings. Your program need merely compare the directive string and then

```
#  Watstar Configuration
#  Bugs to  Erick@Development.Watstar

#"NETMASK" must not be 255.255.255.255

netmask=255.255.254.0
nameserver=129.97.128.24
nameserver=129.97.128.196
gateway=129.97.176.1
domainslist="waterloo.edu"
```

Debugging Waterloo TCP Programs

If you find the program cannot link or gives many errors during compilation, check to see that you have TCP.H in your header include path, the libraries in your LIB include path, small model set on your compiler, and the libraries listed for linking in your .PRJ file or your make file. Did you spell the names correctly, did you capitalize correctly?

Programs crash. Check to see that you do not have register variables enabled. If you use TCC, make certain you have -r-. I use switches: -ms -r- -I\tc\include -I..\

Are you using a kernel which matches your manual. If not, download the latest kernel and documentation changes.

If you are using **sock_gets**, meaning you expect text strings to be returned, you must have previously called **sock_mode** with the flag **TCP_MODE_ASCII**. Later you can reset it to binary mode with **TCP_MODE_BINARY**.

If the program always crashes when you exit, check to see if you are trying to close a socket you already closed. Since the kernal uses a linked list of the socket structures, you could be causing it to go into an endless loop.

Waterloo TCP for UNIX Programmers

This section is intended for the experienced UNIX programmer who wishes to compare that development environment with Waterloo TCP. A rough translation strategy is given, but the programmer is reminded to consult the reference section which provides more detailed descriptions of individual functions.

Major Differences

If you are used to UNIX network programming, you will probably be relatively comfortable with Waterloo TCP. Many UNIX calls have almost identical counterparts under Waterloo TCP. The most major inconvenience is that there are typically subtle differences between the two, particularly since Waterloo TCP was designed to be very tight and suited to TSR's, not the elegant world of concurrent programs and virtual memory.

UNIX provides a very rich and powerful set of system calls for socket initialization which are simply not matched here. This rarely causes a problem since it is unnecessary for a program to require such services in most cases.

socket() This UNIX call allocates a socket from an internal table and returns a socket descriptor, often an index into the table. A call to socket() under Waterloo TCP is unnecessary since the application supplies a suitable chunk of memory and the functions tcp_open, tcp_listen, and udp_open provide the protocol type.

Certainly the data area could be allocated using malloc or calloc, but most applications will probably simply allocate the space with a static structure, or from the heap as a local variable.

close() UNIX sockets, being mapped into the filespace, may be closed in the same way a file would. The Waterloo TCP equivalent is sock_close.

bind() UNIX sockets are created without bindings to any local or destination protocol port, so bind is used to associate the socket to a particular local port. Attempting to bind to an already allocated port will cause bind to fail. Waterloo TCP is very different, the appropriate open or listen command accepts an optional local port number and the implicit bind always works. The ability to have several sockets associated with a particular local port is Waterloo TCP's answer to the UNIX listen command.

listen() Calling listen() with a port number and a queue length (say n) will instruct UNIX to allow n incoming sessions on that port. Upon an incoming session, UNIX creates a socket and passes the socket descriptor to the application through accept(). As discussed under the bind() topic above, Waterloo TCP requires n separate open or listen calls, probably done in a loop.

connect() Since sockets are created in an unconnected state, UNIX programmers use connect() to establish the connection if in fact they are using a connection oriented protocol, namely tcp. Waterloo TCP's **tcp_open** function actually starts the connect operation and wait_sock_established() waits until the connection is complete. In the case of udp connections, wait_sock_established() simply returns immediately.

read() is functionally replaced by sock_read(); write() is functionally replaced by sock_write();

readv() and **writev()** These functions currently have no counterpart under Waterloo TCP. If sock_readv() and sock_writev are added in a future revision, they will be almost identical to readv() and writev().

send() The UNIX send() function provides, among other things, the ability to send urgent data over tcp. This function is not available under Waterloo TCP.

sendto(), **sendmsg()**, **recvfrom()**, and **recvmsg()**, **sendto()** and **sendmsg()** allow a user to specify a destination for udp datagrams. recvfrom() and recvmsg() provide the receive end counterparts. None of these functions are supported under Waterloo TCP.

recv() Although recv() is not available under Waterloo TCP, sone of its features are. The the ability to read-ahead is available using the sock_Preread() function. The ability to read udp datagrams is available using the sock_recv function.

getpeername() and **getsockname()** These functions return the remote and local ip and port numbers respectively. Waterloo TCP replicates these functions with the same names.

getsockopt() and **setsockopt()** Unix uses one function to read each socket option and another to set each socket option. Waterloo TCP has separate commands to set each option. In brief they are:

- sock_mode()	- set binary or ascii mode
- sock_yield()	- set the yield function
	- local or systemwide
- sock_timeout()	- set the standard timeout
	- local or systemwide

Timeouts on tcp listens are set in the tcp_Listen command.

select() BSD 4.3 added a facility for simplifying server design in a single task. Currently there is no similar function on Waterloo TCP although once could easily provide a similar service by running through a loop as is shown in the LPD server program distributed with Waterloo TCP.

gethostid() UNIX and Waterloo TCP return the host ip address.

sethostid() Waterloo TCP and UNIX (given appropriate privileges), will set the host ip address.

gethostname() Unix will return the name of the host. Waterloo TCP does not CURRENTLY have the ability to read the host name unless it has been set with sethostname() or in the configuration file. Full name completion service is planned for future development.

sethostname() Unix (given appropriate privileges) and Waterloo TCP will set the host's name.

setdomainname(), getdomainname() Unix and Waterloo TCP allow applications to set or read the domain name.

inet_addr() UNIX and Waterloo TCP will return a (Waterloo TCP terminology) longword containing the ip address in host order when given a dotted ip string. If the address is not correctly resolved, 0 is returned.

inet_ntoa() UNIX and Waterloo TCP will convert a host ordered ip address to a dotted ip text string. Waterloo TCP uses a static buffer, so you must copy the string or use it before switching tasks or making another call to inet_ntoa().

inet_makeaddr(), **inet_netof()** and **inet_inaof()** These functions are not currently implemented under Waterloo TCP.

UNIX will provides name and host database searching. Waterloo TCP provides only resolve(), a service to find the host's ip address given either dotted-ip or text names.

UNIX allows access to databases of server and application protocol information. Waterloo TCP does not have such databases although they could certainly be added.

Technical Reference

---

getdomainname *return domain string*

---

char * **getdomainname(** *void* **);**

The returned value is a string to the current domain name. This string must not be modified.

```
main()
{
    sock_init();
    printf("Using %s for a domain\n", getdomainname());
}
```

---

gethostid *return ip address*

---

longword **gethostid(** *void* **);**

The returned value is the ip address in host format.

```
main()
{
    char buffer[ 512 ];
    sock_init();
    printf("My IP address is %s\n", inet_ntoa( buffer, gethostid()));
}
```

---

getpeername *return remote computer ip and port information*

---

see getsockname

---

getsockname *return local computer ip and port information*

---

void **getsockname(** *void* * **s,** *struct sockaddr* * **block,** *int* * **len);**
void **getpeername(** *void* * **s,** *struct sockaddr* * **block,** *int* * **len);**

getsockname returns the user's ip address and port information for a particular socket whereas getpeername returns the peer's ip address and port information for a particular socket.

If no information is available, the returned lenght is 0, otherwise it is the length of the completed structure or the maximum length permitted under *len.

Note, this is based on a UNIX function of the same name. The block should be 14 bytes long. Waterloo TCP will assume 14 bytes if you pass NULL for the pointer to len. The data structure is

```
typedef struct sockaddr {
        word     s_type;
        word     s_port;
        longword s_ip;
        byte     s_spares[6];  /* not used for tcp/ip connections */
};
```

---

htons, **htonl**, **ntohs**, **ntohl** *convert byte order*

---

word **ntohs(** *word* **value);**	convert network word to host word
word **htons(** *word* **value);**	convert host word to network word
longword **ntohl(** *longword* **value);**	convert network longword to host longword
longword **htonl(** *longword* **value);**	convert host longword to network longword

These functions are necessary since our hosts (MS-DOS comptuters) are based on the INTEL 80x8x line which do not use the same byte orderring as the standard for network byte orderring.

---

inet_addr *convert dotted ip to binary*

---

longword **inet_addr(** *char* * **dotted_ip_string);**

inet_addr simply converts an ip address from dotted ip format to host format (longword). No check is made as to the the validity of the address.

```
eg.
main( int argc, char *argv[] )
{
   longword host;

   if (argc == 2 ) {
      host = inet_addr( argv[1] );
```

```
        printf("Host set to %lx\n\r", host );
    } else
        printf("use: %s a.b.c.d\n\r", argv[0] );
}
```

see also inet_ntoa()

---

inet_ntoa *convert binary address to ASCII string*

---

char ***inet_ntoa(** *char* ***s,** *longword* **ip);**

inet_ntoa uses the user-supplied buffer to build an ascii string containing a dotted ip representation of the supplied ip address. A sufficient buffer size would be 16 bytes.

for an example program, see gethostid(). see also inet_addr.

---

ip_timer_init, ip_timer_expired *socket timeout functions*

---

void **ip_timer_init(** *void* * **s,** *word* **seconds);**
word **ip_timer_expired(** *void* * **s);**

These two functions are used in the sock_wait_... macros to provide timeouts. First ip_timer_init must be called with the number of seconds to permit, 0 indicating to never timeout. That function places a timer value into a portion of the socket structure. At later intervals, ip_timer_expired is called with a pointer to the function and returns 1 when the timer has expired.

The following is a simple example from **sock_wait_closed**. Please note that this is just an example, the kernal code is subject to modification. If you check out tcp.h, you will see that sock_wait_closed calls the function _ip_delay2, and here is that code.

```
_ip_delay2( void *s, int timeoutseconds, procref fn, int *statusptr)
{
    int status;

    ip_timer_init( s , timeoutseconds );        /* set timeout */
    do {
        kbhit();                                /* permit ^c */
        if ( !tcp_tick( s )) {
            status = 1;                         /* fully closed or reset */
            break;
        }
        if ( ip_timer_expired( s )) {           /* check for expiration */
            sock_abort( s );                    /* give up and use reset */
            status = -1;                        /* signal an error */
            break;
        }
        if (fn) {                               /* call optional user function */
            if (status = fn(s))
                break;
        }
```

```
        if ( s->tcp.usr_yield ) (*s->tcp.usr_yield)();        /* call yield */

    } while ( 1 );
    if (statusptr) *statusptr = status;
    return( status );
}
```

_ping, _chk_ping *provide ping services*

---

word **_ping(** *longword* **host_ip,** *longword* **sequence_number);**
longword **_chk_ping(** *longword* **host_ip,** *longword* * **sequence_number);**

The programmer's guide portion of this manual describes the ICMP and PING capabilities, this section describes only how to use these functions.

_ping must be called with a destination host address and a sequence number which will be embedded in the packet.

tcp_tick or sock_tick should be called very frequently as they are called to manage the timing of ping.

_chk_ping should be called frequently with a host ip address. If an appropriate packet is found from that host ip address, the sequence number is returned through *sequence_number, and the time difference between our request and their response is returned in PC ticks. There are approximately 18.2 ticks per second.

If _chk_ping returns 0xffffffffL, there were no ping receipts on this current call.

Take a look at the sample ping.c program available on the latest distribution for more details.

---

psocket *print a socket remote host and remote port value*

---

void **psocket(** *void* * **s);**

Given an open udp or tcp socket, the remote host is printed out in dotted ip format followed by a colon and the decimal port number on that machine. This routine can be useful for debugging your programs, simply place it after the sock_wait_established.

---

resolve *resolve an IP address*

---

longword **resolve(** *char* ***host_string);**

resolve will resolve a text string containing a host name or a dotted ip address into a longword containing the ip address. In the case of dotted ip, no check is made as the the validity of the address. This function currently blocks while it executes. To simply convert dotted ip to longword, see inet_addr.

for a sample program, see tcp_Open.

---

rip *remove carriage returns and line feeds*

---

char * **rip(** *char* * **string);**

Many situations arise where the programmer must strip off \n and \r characters from a string. rip performs this function for the Waterloo TCP kernal.

After finding a newline (\n) or carriage return (\r) character, rip replaces that character with a zero. If there are multiple \n's or \r's or a combination, at least the first \n or \r character is replaced by a zero. The resultant string beond the 0 character is undefined.

rip also returns a pointer to this string so you can use it like:

```
setmode( s, TCP_MODE_ASCII );
...
sock_puts( s, rip( questionable_string ));
```

In the above example, the socket has been placed in ASCII mode, so sock_Puts automatically adds a \n newline character, so rip is used to make certain the string does not already have a newline character.

Remember, rip modifies the source string, it does not work with a copy!

Although this is not really a networking function, it seems useful and is linked in anyways, I have simply documented it for you.

---

setdomainname *set the domain string*

---

char ***setdomainname(** *char* ***string);**

The domain name returned by **getdomainname()** and used for **resolve()** is set to the value in the current string. Note that changing the contents of the string after a setdomainname() call may or may not change the value of the system domain string and is not recommended. You are recommended to dedicate a static location which may be dedicated to holding that name.

setdomainname(NULL) is an acceptable way to totally remove any domain name and subsequent resolves will not attempt to append a domain name.

The passed string pointer is always returned.

sethostid *set the host's ip address*

longword **sethostid(** *longword* **ip);**

This function sets the system's default ip address. Changing the ip address will turn existing TCP or UDP sessions into toast. You should close all sockets before calling this function. The passed ip address is always returned.

set_debug_state *toggle ths system debugging state*

void **set_debug_state(** *word* **state);**

Some very rudimentary debugging information is available if you set the socket debugging feature on.

set_debug_state(1);	enables debugging
set_debug_state(0);	disables debugging

sock_abort *abort a socket connection*

void **sock_abort(** *void* * **s);**

As described more fully under the sock_close section, sock_abort can be used to abort a connection. This service is not meaningful under udp and simply performs a close.

sock_close *close a socket*

void **sock_close(** *void* * **s);**

sock_close will attempt to close a socket and no more data may be sent out or received through that socket.

In the case of udp, the socket is immediately closed since udp is a connectionless protocol. tcp, however, is a connection oriented protocol so the close must be negotiated with the remote computer. sock_wait_closed or another mechanism must the used to ensure that a tcp connection is closed.

In emergency cases, it is possible to abort the tcp connection rather than close it. Although not correct for normal transactions, this service is available and used by all tcp/ip systems. See sock_abort for more details.

example programs are given under tcp_open, tcp_listen, and udp_open.

---

sockerr *return last error message*

---

char ***sockerr(*** *void* * **s);**

sockerr returns the last ASCII error message recorded for a particular socket. If no messages have been recorded, the returned value is NULL. The messages are read-only, you may not modify them!

eg.
```
char *p;
...
sock_err:
    if ( status == 1) puts("Closed normally");
    else if ( p = sockerr( s )) printf("Socket closed with error '%s'\n\r", p );
```

---

sock_established *affirm that a connection is established*

---

int **sock_established(** *void* ***s);**

tcp connections require a handshaked open to ensure that both sides recognize a connection. Whether we initiated the connection with tcp_Open or were called after we allocated a session with tcp_Listen, sock_Establish will contintute to return 0 until the connection is established at which time it will return 1.

Alternatively, you can use the sock_wait_established macro which will busywait for a connection to be established as well as offering a timeout service and multitasking capabilities. See sock_wait_established for more details.

udp is a connectionless protocol, since udp_Open checks the validity of the remote host, the connection is assumed to work, and hence sock_Established alway sreturns 1 for udp sockets.

You only need to use one of sock_wait_Established and sock_wait_established

---

sock_delay *system timeout value*

---

extern word **sock_delay;**

The sock_wait_... functions allow you to specify a timeout value in seconds. In many cases, it is not advantageous to impose your own delay values. sock_delay is a system constant which contains a resonable timeout value. Additionally, users may override the timeout value I selected by placing the line

sockdelay=*n*
eg. **sockdelay=5**
in their WATTCP.CFG file. To use this system timeout value, either substituting it for the seconds parameter of the sock_wait_... functions, or use a *seconds* value of -1. Remember, a *seconds* value of 0 means never timeout.
Note: using -1 for seconds has not yet been implemented. It will be mentioned in the distribution notes when it has been added. For now you must specify sock_delay explicitly.

---

sock_fastread *attempt to read data from a stream*

---

int **sock_fastread(** *void* ***s,** *byte* ***dp,** *int* **len);**

sock_fastread attempts to read data from a socket. If possible, the buffer is filled, otherwise, only the number of bytes read is returned. A return value of -1 indicates a socket error. If sock_wait_input is used to wait for data, -1 will never be returned since sock_wait_input handles the error.

A sample application using sock_fastread is listed under sock_read

---

sock_fastwrite *attempt to write data to a stream*

---

int **sock_fastwrite(** *void* ***s,** *byte* ***data,** *int* **len);**

sock_fastwrite attempts to place the data onto the streams outgoing buffer and transmit the data. If the data will not fit, the number of bytes transmitted will be returned.

For udp, sock_fastwrite will send one record. For tcp, the new data may be transmitted if enough data is in the buffer or sufficient time has expired or the user has explicitly used sock_flushnext to indicate this data should be flushed immediately. In either case, no guarentee of acceptance at the other end is guarenteed.

---

sock_flush *flush a socket*

---

void **sock_flush(** *void* ***s);**

sock_flush will flush the unwritten portion of the tcp buffer to the network. No guarentee is given of delivery at the remote computer.

In the case of a udp socket, no action is taken.

sock_flushnext is recommended over sock_flush.

---

sock_flushnext *flush the next item sent*

---

void **sock_flushnext(** *void* ***s);**

Writing to tcp sockets does not guarentee that the data is actually transmitted or that the remote computer will pass that data to the other client in a timely fashion. Using a flush function, certain other functions will guarentee that Waterloo TCP places the data onto the network. No guantee is made that the remote client will receive that data.

sock_flushnext is the most efficient of the flush functions, it indicates that the next function to be used to send data to a particular socket will also force that socket to be flushed meaning the data will be transmitted immediately.

Several functions imply a flush and do not require an additional flush, namely sock_puts sock_printf and sometimes sock_putc (when passed a \n).

---

sock_getc *read a single character from a socket*

---

char **sock_getc(** *void* ***s);**

If another character is available from the stream, the character is returned. Other possible values are -1 for a socket error or 0 if no character is available. No ASCII translation is performed, so the 0 character can only be verified if data was known to exist.

A sample application using sock_getc is listed under sock_read.

---

sock_gets *read a line of input from a socket*

---

int **sock_gets(** *void* ***s,** *char* ***text,** *int* **len);**

sock_gets reads a string from a socket and removes the carriage return or linefeed. The length of the new string is returned. If the new string is longer than the buffer size descibed in **len**, the string is null terminated at the given lengths and the remainder is discarded. If you intend to use sock_Gets, you must first set ASCII mode using sock_mode.

A sample application using sock_Gets is listed under sock_Read.

---

sock_init *initialize Waterloo TCP*

---

void **sock_init();**

The Waterloo TCP system must be initialized to function correctly and sock_init is the easiest method to ensure sufficent measures have been taken to make the system operational. sock_init attempts to read the system configuration file. The configuration file may be extended as is shown under the section usr_init. The location of the executable is checked first, failing that, the current subdirectory is checked. The file is openned using _open and all calls avoid using memory consuming functions. After configuration, a control break handler is installed using the sock_cbk function. See sock_cbrk for more details.

Sample applications in sock_read and usr_init all show how sock_init must be used.

---

sock_mode *set the socket's mode or features*

---

void **sock_mode(** *void* ***s,** *word* **mode);**

There are several mode features:
TCP_MODE_ASCII
TCP_MODE_BINARY

tcp and udp sockets are usually in binary mode, meaning that tcp is treated as byte stream and udp is treated as records filled with bytes.

Setting the mode to ASCII for a particular socket causes the appropriate protocol to act as an ASCII stream of characters with record boundaries on the newline (\n) character for certain functions. The new behavior means sock_dataready will only return true when a complete, \n terminated string is resident in the buffer, or if the buffer is full without the occurance of a \n character. sock_puts is modified in that it appends a \n to any line printed. sock_gets (which should only be used in ASCII mode) strips of the \n character, so the functions effortly pass data. sock_printf in ASCII mode appends a \n since it uses sock_puts to perform the transfer. sock_scanf uses sock_gets to read the line and hence requires the \n and must only be used in ASCII mode.

see sock_read for a sample application

TCP_MODE_NAGLE
TCP_MODE_NONAGLE

The Nagle algorithm may substantially network traffic with little negative effect on a user application. In some situations, the Nagle improves application performance. By default, Nagle is selected. If you are doing X-Windows or real time data collection, you may switch Nagle off by selecting TCP_MODE_NONAGLE. The Nagle flags only affect *tcp* connections.

UDP_MODE_CHK
UDP_MODE_NOCHK

Checksums are required under *tcp* but not necessarily so under *udp*. For example, a *udp* based program may use its own checksum scheme and the *udp* internal checksum is redundant. If you do not wish to compute checksums for data you transmit on a *udp* socket, select UDP_MODE_NOCHK. Note, you do not control whether the remote computer will send checksums. If that computer does checksum its outbound data, Waterloo TCP will check the received packet's checksum.

---

sock_preread *get advance copy of received data*

---

int **sock_preread(** *void* ***s,** *byte* ***dp,** *int* **len);**

Some situations arise where it would be nice to read data without causing it to disappear from the socket's buffer. Usually that means dubble buffering. sock_preread works exactly like sock_fastread, except it does not remove the read data from the data buffers. The returned value is the number of bytes transferred, 0 for no data waiting, or -1 on a socket error.

---

sock_putc *write a character to a socket*

---

char **sock_putc(** *void* ***s,** *char* **c);**

A single character is placed on the output buffer. In the case of a \n character, the buffer is flushed as described under sock_flushnext. No other ASCII character expansion is performed.

The returned value is the character **c**.

NOTE: sock_putc uses sock_write and thus may block of the output buffer is full. See sock_write for more details.

---

sock_puts *write a string to a socket*

---

char **sock_puts(** *void* ***s,** *char* ***text);**

A string is placed on the output buffer and flushed as described under sock_flushnext. If the socket is in ASCII mode, a \n character is placed on the stream following the string. No other ASCII character expansion is performed. The returned value is a pointer to the original stream.

NOTE: sock_puts uses sock_write and thus may block of the output buffer is full. See sock_write for more details.

---

sock_r... *return information on receive buffers*

---

see sock_t...

---

sock_read *read a complete buffer from a stream*

---

int **sock_read(** *void* ***s,** *byte* ***dp,** *int* **len);**

sock_read will busywait until the buffer is completely full or a socket error exists. If multitasking has been enabled with sock_yield, sock_read will honour the yield function in a tight loop.

The returned value is the number of bytes read or -1 on an error.

The following program shows how the four main input functions may be used to read a text stream. Note that sock_fastread and sock_read do not necessarily return a complete or single line, they return blocks of bytes. In comparison, sock_getc returns a single byte at a time and yielded shabby performance.

```
/*
 * This is a sample FINGER program which compares sock_fastread, sock_read,
 * sock_gets, and sock_getc for handling ASCII data.
 *
```

```
 * Note that the sock_fastread, sock_read, and sock_getc functions do NOT
 * return single line strings, they simply return orderred bytes.
 *
 * sock_getc looks the simplest, but it has the highest overhead both in
 * terms of Waterloo TCP, and especially in terms of the output through
 * putch().
 *
 * FINGER [user]@host mode
 *
 * where mode is 0, 1, 2 or 3 to indicate using sock_fastread, sock_read
 * sock_getc or sock_gets.  All modes returned identical output to the screen.
 */


#include <stdio.h>
#include <tcp.h>

#define FINGER_PORT 79

finger(userid, host, hoststring, method)
char *userid, *hoststring;
longword host;
int method;
{
    tcp_Socket fingersock;
    tcp_Socket *s;
    char buffer[ 513 ];             /* space for 512 plus zero terminator */
    int status;
    int len;


    s = &fingersock;
    if (!tcp_open( s, 0, host, FINGER_PORT, NULL )) {
        puts("Sorry, unable to connect to that machine right now!");
        return;
    }

    printf("waiting...\r");
    sock_wait_established(s, sock_delay , NULL, &status);

    if (*userid)
        printf("'%s' is looking for '%s'...\n\r\n\n", hoststring, userid);

    strcpy( buffer, userid );
    rip( buffer );                  /* kill all \n and \r's */
    strcat( buffer , "\n");

    sock_puts( s, buffer );

    switch (method) {
/*********************************************************************
```

```
*               using sock_FastRead                                    *
**********************************************************************/
        case 0 :
                while ( 1 ) {
                    sock_wait_input( s, 30, NULL, &status );

                    len = sock_fastread( s, buffer, 512 );
                    buffer[ len ] = 0;              /* must terminate it */
                    printf( "%s", buffer );
                }
                break;
/**********************************************************************
*               using sock_read                                        *
**********************************************************************/
        case 1 :
                while( 1 ) {
                    sock_wait_input( s, 30, NULL, &status );
                    len = sock_dataready( s );
                    if ( len > sizeof( buffer ))
                          len = sizeof( buffer );

                    sock_read( s, buffer, len );
                    buffer[ len ] = 0;
                    printf( "%s", buffer );
                }
                break;
/**********************************************************************
*                    using sock_getc                                   *
**********************************************************************/
        case 2 :
                while ( 1 ){
                    sock_wait_input( s, 30, NULL, &status );
                    putch( sock_getc( s ) );
                }
                break;
/**********************************************************************
*                    using sock_gets                                   *
**********************************************************************/
        case 3 :
                sock_mode( s, TCP_MODE_ASCII );
                while ( 1 ) {
                    sock_wait_input( s, 30, NULL, &status );
                    len = sock_gets( s, buffer, 512 );
                    puts( buffer );
                }
                break;
/************************************************************************/
    }

sock_err:
    switch (status) {
```

```
        case 1 : /* foreign host closed */
                 break;
        case -1: /* timeout */
                 printf("\n\rConnection timed out!");
                 break;
    }
    sock_close( s );
    printf("\n\r");
}

char *meth[] = { "sock_fastread", "sock_read", "sock_getc", "sock_gets" };


main(int argc, char **argv )
{
    char *user,*server;
    longword host;
    int status;
    word method;

    sock_init();

    /* process args */
    do {
        if (argc == 3) {
            method = atoi( argv[2] );
            user = argv[1];
            if (server = strchr( user, '@'))
                break;
        }
        puts("   FINGER  [userid]@server method");
        exit( 3 );
    } while ( 0 );

    *server ++ = 0;

    if ( method > 3 ) {
       puts("only values 0 through 3 are valid");
       exit( 2 );
    }

    printf("Using method %s\n\r", meth[ method ]);

    if (host = resolve( server )) {
        status = finger( user, host, server, method);
    } else {
        printf("Could not resolve host '%s'\n\r", server );
        exit( 3 );
    }
    exit( status );
}
```

---

sockstate *return state message*

---

char ***sockstate(*** *void* * **s);**

sockstate returns an ASCII message indicating the current state of a particular socket. For udp sockets which are stateless, the returned message is always "UDP Socket". If no message is available, this function may return NULL. The messages are read-only, you may not modify them!

eg.
```
char *p;
...
#ifdef DEBUG
if ( p = sockstate( s ))
    printf("Socket state is '%s'\n\r", p );
#endif DEBUG
```

---

sock_t... *return information about the transmit buffers*

Some applications need intimite details concerning the amount of memory available in transmit and receive buffers. Waterloo TCP will provide the size, used space and remaining space in either buffer with the following functions which all return the number of bytes:

	transmit buffer	receive buffer
buffer size	*int* **sock_tbsize(** *void* ***s);**	*int* **sock_rbsize(** *void* ***s);**
buffer bytes in use	*int* **sock_tbused(** *void* ***s);**	*int* **sock_rbused(** *void* ***s);**
buffer bytes free	*int* **sock_tbleft(** *void* ***s);**	*int* **sock_rbleft(** *void* ***s);**

These values are determined at the time of the function call and are accurate.

If you are in a TSR, for example, and you do not wish to block, you may first query how much space is available for writing before generating data which must be transmitted. This removes the need for your application to also buffer data.

eg.
```
if ( sock_tbsize( s ) > 10 ) {
    /* we know we can send up to 10 bytes without blocking or overflowing */
    ....
}
```

Another use might be in the addition of a multitasker function. See sock_yield for more information.

---

sock_tick *perform basic socket management*

---

sock_tick(*void* * **s,** *int* * **optional_status_ptr);**

sock_tick calls tcp_tick to quickly check incomming and outgoing data and to manage all the open sockets. If our particular socket, **s**, is either closed or made inoperative due to an error condition, sock_tick sets the value of *optional_status_ptr (if the pointer is non-null) to a value idetical to those described in sock_wait_... and then performs a **goto sock_err:** exactly like the sock_wait_... routines do.

---

sock_wait_... *functions which wait for events or timeouts*

Although Waterloo TCP does not normally block on i/o, user applications are easier to write for systems which do block. The following is a list of macros commonly used to make Waterloo TCP work like a blocking system.

These macros have an additional capability besides waiting, under some condition where either the remote or local host closes or resets the connection, these macros jump to a section of code to handle the error. For the current time, they perform a **goto sock_err;** in the current function. This mechanism is not truly pretty and may be avoided by inspecting the **tcp.h** header file and creating new macros with similar functionality. The sock_wait functions have been chosen because they work correctly in tsrs and consume little space as compared to the alternatives.

All sock_wait_... functions take the same parameters:

void ***s,** *word* **seconds,** *word (****optional_fn*)*,** *word* ***optional_status_ptr**

The first parameter is the socket in question. The **seconds** parameter is the timeout value in seconds before the particular socket gets tired and resets the connection and socket. A value of zero indicates the macro should never timeout. **optional_fn** is the pointer to an option 'C' function which will be called repeatedly inside the loop. If the optional function returns anything other than zero, the wait macro will terminal and return the value which was returned by the optional function. Typically the optional_fn parameter will be NULL to indicate no function. You might like to have an optional function which queries the keyboard and aborts the operation based upon a user demand. **optional_status_ptr** is a pointer to the status word. If this parameter is NULL, no status number will be available, but the macro will otherwise function as normal. Typically the pointer will point to a local signed integer which is used only for status. Following a **sock_err:**, the status value may be tested to determine how the socket was ended. A value of 1 means a proper close while a -1 value indicates a reset or abort.

A good value for the timeout value is **sock_delay**, a system variable set on configuration. Typically sock_delay would be about 20 seconds. You may find this values too low and may set them to a better value using the WATTCP.CFG configuration file. See sock_delay for more details.

The sock_wait_... macros also honour the user defined yield function for multitasking if one has been set with sock_yield.

sock_wait_established	-	wait until a connection is established for tcp sockets or return immediately for udp sockets.
sock_wait_input	-	wait until input exists for functions such as sock_read, sock_gets. As described under sock_mode, if in ASCII mode, sock_wait_input only returns when a complete string exists or the buffer is full.
sock_wait_closed	-	wait until a connection is fully closed or return immediately for udp sockets.

For examples, see tcp_Open, tcp_Listen, and sock_Read

---

sock_write *write a complete buffer to a stream*

---

int **sock_write(** *void* ***s,** *byte* ***data,** *int* **len);**

sock_write will busywait until the buffer is completely written or a socket error occurs. If multitasking has been enabled with sock_yield, sock_write will honour the yield function in a tight loop.

The returned value is the number of bytes written or -1 on an error.

For udp, sock_write will send one (or more) records. For tcp, the new data may be transmitted if enough data is in the buffer or sufficient time has expired or the user has explicitly used sock_FlushNext to indicate this data should be flushed immediately. In either case, no guarentee of acceptance at the other end is guarenteed.

---

sock_yield *Sets the yield function*

---

void **sock_yield(** *void* ***s,** *void (****yield_fn***)())***);**

Many complex programs are more easily handled with a multitasker which switches between threads as they each wait for data to be received or written. sock_yield provides kernal access to users wishing to use the multitasking features. The discussion for this topic is handled in one of the appendicies.

---

tcp_cbrk *control break handler*

---

void **tcp_cbrk(** *word* **mode);**

You will not normally need to call this function.

Waterloo TCP application programs cannot allow users to stop program execution with control break because we will often continue to receive packet interrupts after the program has executed. Instead, a thoughtout solution must be used. The current possibilities are:

a) Prevent control breaks or ignore them

b) Upon a control break, close all sessions and release packet driver and interrupt vectors

Waterloo TCP allows you to do either. It also provides an option for you to enable or disable a text message telling the user which action is to be taken.

The controlling mechanism for these features is the tcp_cbrk function which is called with two bitmasks which may be logically ORed together. Mode values containing 0x01 enable control breaks, and mode values containing 0x10 will display a message.

mode value	features
0x00	allow control breaks and use no special message

0x01	disallow control breaks and do not display a message
0x10	allow control breaks and indicate the program was terminated
0x11	disallow control breaks and inform the user upon a control break

Waterloo TCP automatically calls tcp_cbrk with the value 0x10 during the sock_init function. You may override this default setting by calling tcp_cbrk with a different value at any point after the sock_init function.

---

tcp_listen *open a TCP port for an incomming session*

---

int **tcp_listen(** *void* ***s,** *word* **lport,** *longword* **ina,** *word* **port,** *int (****signal_handler***)***,** *word* **timeout);**

tcp_listen is used to tell Waterloo TCP that an incomming session for a particular port will be accepted. The parameter **s** is a pointer to a buffer space at least the size of *tcp_Socket* and **lport** is a mandatory value which indicates on which port an incomming socket will be accepted.

The values of the remote host's addres (*ina*) or the remote port (*port*) are possibly zero, in which case they are taken as wild cards for those particular values. Specifying a particular host and/or port will limit the incomming session to the desired host and/or port address.

The parameter for signal_handler is the pointer to a function which will be called when the socket is either closed or reset. *Some details for implementation of this service have not been finalised and the user is recommended to insert a value of NULL for the present time.*

The **timeout** parameter defines the number of seconds acceptable before sock_wait_established will time out. This value is typically set to 0 to indicate there is no timeout. A value of five would indicate the connection would have to be established in five seconds or less.

After making a call to tcp_listen you must either continually poll the connection or have the computer poll the connection until a session is established. For more details, see the sample application and functions **sock_established** and **sock_wait_established**.

The signal_handler is not well
Multiple tcp_listens to the same local port (*lport*) are acceptable and constitute the Waterloo TCP mechanism for supporting multiple incomming connections to the same local port. Each time another host attempts to open a session on that particular port, another one of the listens will be consumed until such time as all listens have become established sessions and subsequent remote host attempts will receive a reset.

```
#include <tcp.h>
#include <stdio.h>

#define TELNET_PORT 23

static tcp_Socket *s;
char *userid;

telnets(port)
int port;
```

```
{
    tcp_Socket telnetsock;
    char buffer[ 512 ];
    int status;
    int len;

    s = &telnetsock;
    tcp_listen( s, port, 0L, 0, NULL, 0);

    sock_wait_established(s, 0, NULL, &status);

    puts("Receiving incomming connection");
    sock_mode( s, TCP_MODE_ASCII );
    sock_puts( s, "Welcome to a sample telnet server.");
    sock_puts( s, "Each line you type will be printed on this screen once you hit return.");

    /* other guy closes connection except if we timeout .... */
    while ( 1 ) {
        sock_wait_input( s , 0, NULL, &status);
        sock_gets( s, buffer, 512 );
        puts( buffer );
    }

sock_err:
    switch (status) {
        case 1 : /* foreign host closed */
                puts("User closed session");
                return;
        case -1: /* timeout */
                printf("\n\rConnection timed out!");
                return;
    }
}


main(int argc, char **argv )
{

    sock_init();

    telnets( TELNET_PORT);

    exit( 0 );
}
```

sock_recv *receive a udp datagram*

int **sock_recv(** *void* ***socket,** *byte* ***buffer,** *int* **maxlength);**

After a *udp* socket has been initialized with **udp_open** and **sock_recv_init**, **sock_recv** scans the buffers for any datagram received by that socket. If there is a datagram, the length is returned and the user buffer is filled. If no datagrams were found, **sock_recv** returns 0.

```
#include <stdio.h>
#include <tcp.h>
#include <newstruct.h>

#define SAMPLE 401


udp_Socket data;
char bigbuf[ 8192 ];

main(int argc, char *argv[])
{
   word templen;
   char spare[ 1500 ];

   sock_init();
   if ( !udp_open( &data, SAMPLE, 0xffffffff, SAMPLE, NULL) ) {
         puts("Could not open broadcast socket");
         exit( 3 );
   }

   /* set large buffer mode */
   if ( sock_recv_init( &data, bigbuf, sizeof( bigbuf ))) {
         puts("Could not enable large buffers");
         exit( 3 );
   }

   sock_mode( &data, UDP_MODE_NOCHK );          /* turn off checksums */

   while (1) {
         tcp_tick( NULL );

         if ( templen = sock_recv( &data, spare, sizeof( spare ))) {
             /* something received */
             printf("Got %u byte packet\n", templen );
         }
         kbhit(); /* check for control breaks */
   }
}
```

---

sock_recv_from *receive a udp datagram and its source ip*

---

int **sock_recvfrom(** *void* ***socket,** *byte* ***buffer,** *int* **maxlength);**

After a *udp* socket has been initialized with **udp_open** and **sock_recv_init**, **sock_recv_from** scans the buffers for any datagram received by that socket. If there is a datagram, the length is returned, the user buffer is filled, and the source address is placed in *fromptr*. If no datagrams were found, **sock_recv_from** returns 0.

see sock_recv and sock_recv_init for more details. A sample program may be found under sock_recv.

---

sock_recv_init *initialize udp datagram service*

---

int **sock_recv_init(***void* ***socket,***byte* ***buffer,***int* **bufferlen);**

The basic sock read functions (**sock_read**, **sock_fastread**, etc.) are not adequate for all your *udp* needs. The most basic limitation is their is inability to treat *udp* as a record service.

A record service must receive distinct datagrams and pass them to the user program as datagrams. You will wish to know the length of the received datagram and the sender (if you opened in broadcast mode). You may also receive the datagrams very quickly, so you must have a mechanism to buffer them.

Once a socket has been openned with **udp_open**, you can use **sock_recv_init** to initialize that socket for the **sock_recv** and **sock_recv_from** functions. Note that **sock_recv** and related functions are *incompatible* with **sock_read**, **sock_fastread**, **sock_gets**, **sock_getc** and **sock_scanf**. Once you have used **sock_recv_init**, you can no longer use the older style calls.

sock_recv_init installs a large buffer area which gets segmented into smaller buffers. Whenever a *udp* datagram arrives, Waterloo TCP stuffs that datagram into one of these new buffers. The new functions scan those buffers. You must select the size of the buffer you submit to sock_recv_init, you should make it as large as possible, say 4, 8 or 16 Kbytes. **sock_recv_init** returns 0 on success.

See sock_recv and sock_recv_from for more details. A sample application is listed under sock_recv.

tcp_open *open an outbound TCP session*

int **tcp_open(** *void* ***s,** *word* **lport,** *longword* **ina,** *word* **port,** *int (****signal_handler*))**;**

tcp_open is used actively create sessions with other computers. The parameter **s** is a pointer to a buffer space at least the size of *tcp_Socket*.

The values of the remote host's addres (*ina*) or the remote port (*port*) are possibly zero, in which case they are taken as wild cards for those particular values. Specifying a particular host and/or port will limit the incomming session to the desired host and/or port address.

Multiple tcp_listens to the same local port (*lport*) are acceptable. Each time another host attempts to open a session on that particular port, another one of the listens will be consumed until such time as all listens have become established sessions and new remote host attempts will receive a reset.

After making a call to tcp_open you must either continually poll the connection or have the computer poll the connection until a session is fully established. For more details, see the sample application and functions **sock_established** and **sock_wait_established**.

tcp_open will fail and return 0 is the connection could not be opened due to routing difficulties such as the inability to resolve the remote computer's hardware address with ARP.

```
/* Sample using tcp_open */
#include <stdio.h>
#include <tcp.h>


main(int argc, char **argv)
{
   word status;
   word port;
   longword host;
   tcp_Socket tsock;

   sock_init();

   if (argc != 3 ) {
        printf("%s host port\n\r", argv[0] );
        exit( 3 );
   }

   if (!(host = resolve( argv[1]))) {
        puts("Could not resolve host");
        exit( 3 );
   }

   port = atoi( argv[2] );

   printf("Attempting to open '%s' on port %u\n\r", argv[1], port );
```

```
    if ( !tcp_open( &tsock, 0, host, port , NULL )) {
        puts("Unable to open TCP session");
        exit( 3 );
    }

    /* use sock_delay from WATTCP.CFG or default to 10 */
    printf("Waiting a maximum of %u seconds for connection to be established\n\r", sock_delay );

    sock_wait_established( &tsock, sock_delay, NULL, &status );
    puts("Socket is established");

    sock_close( &tsock );
    sock_wait_closed( &tsock, sock_delay, NULL, &status );

sock_err:
    switch ( status ) {
      case 1 :  puts("Connection closed normally");
                break;
      case 2 :  puts("Problem occurred...");
                sockerr( &tsock );
                break;
    }
    exit( (status == 1 ) ? 0 : 1 );
}

/* the following is the results from running 'test sunee 25'

        Attempting to open 'sunee' on port 25
        Waiting a maximum of 10 seconds for connection to be established
        Socket is established
        Connection closed normally
*/
```

---

tcp_tick *perform system processing*

---

word **tcp_tick(** *void* ***s);**

This is a low level proceedure which few programmers will need. Higher level functionality may be attained by using the sock_tick function, sock_wait_... macros, or in some cases the blocking read/write calls.

Waterloo TCP must constantly be doing little jobs like checking for new packets, processing those packets, and performing retransmissions on lost data. On most other computer systems and other kernals, the job of performing these required operations in the background is often performed by doing a task switch. As stated elsewhere, Waterloo TCP does not use a tasker for its basic operation although it can adopt a tasker for the user level services. Instead a single kernal routine is designed to quickly do this processing and return as soon as possible. That basic routine is **tcp_tick**.

The detailed explanation chapter gives insight on the inner workings of tcp_tick; this section aims to describe its use to an application programmer.

tcp_tick takes a parameter, namely a socket pointer. If the socket pointer is NULL, the returned value is undefined. If the socket pointer points to a valid open socket, tcp_tick returns one of two values

0	- connection reset or closed by other host
not 0	- connection is fine

Although you may ignore the returned value of tcp_tick, it is the easiest method to determine if the socket is inoperative.

tcp_tick performs processing on all sockets upon each invocation, the pointer is simply a convienence measure for determining the status of the current connection.

see also sock_tick, sock_wait_...

---

udp_open *open a udp port*

---

int **udp_open(** *void* ***s,** *word* **lport,** *longword* **ina,** *word* **port,** *int (****signal_handler*))**;**

udp sockets are used for connectionless data transfers. Despite the connectionless nature, which is protocol dependant, Waterloo TCP imposes a socket mechanism which requires a destination address (*ina* in the above function). As described under the UDP Datagram Service chapter, you may elect to use datagram oriented features. Refer to that chapter for an explanation and an example.

If the remote host is set to -1, all packets received by this computer and destined for *lport* will return data with the various read statmemts. Write statements to this socket will cause broadcasts. This mechanism is suitable for broadcast information such as RIP packets.

If the remote host is set to 0, the next packet received by Waterloo TCP destined for this machine's udp port number *lport* will complete the socket and

If the remote host is set to a particular address, either host may initiate traffic.

Multiple udp_opens with ina set to zero is a useful way of accepting multiple incomming sessions.

udp_open will return 0 if the socket cannot be openned. A typical reason would be that the host's physical address cannot be resolved using ARP or normal routing mechanisms. When an error occurs, you might try the Waterloo TCP PING.EXE application to test the accesibility of the other computer.

NOTES: At the date of publication, all details concerning ina = -1 have not been fully implemented and tested. This information is provided for completeness when such provisions have been made.

Although multiple udp_opens may normally be made with the same lport number, only one udp_open should be made on a particular lport if the ina is set to -1. Esentially, the broadcast and non-broadcast protocols can not co-exist.

sock_connected and **sock_wait_connected** is useful only for connection-oriented protocols such as tcp. For compatibility, both functions treat udp sockets which have been openned as connected.

```
/* Sample using tcp_open */
#include <stdio.h>
#include <tcp.h>


main(int argc, char **argv)
{
    word status;
    word port;
    longword host;
    udp_Socket usock;

    sock_init();

    if (argc != 3 ) {
        printf("%s host port\n\r", argv[0] );
        exit( 3 );
    }

    if (!(host = resolve( argv[1]))) {
        puts("Could not resolve host");
        exit( 3 );
    }

    port = atoi( argv[2] );

    printf("Attempting to open '%s' on port %u\n\r", argv[1], port );

    if ( !udp_open( &usock, 0, host, port , NULL )) {
        puts("Unable to open UDP session");
        exit( 3 );
```

```
    }

    /* udp, no need to wait for a connection unless we expect an incomming
     * session.   calling wait_sock_established would return immediately
     */

    puts("Socket is established");

    /* note, no data has been sent, not connection established, the other
     * guy doesn't even know we are interested
     */

    sock_close( &usock );

sock_err:
    switch ( status ) {
        case 1 :    puts("Connection closed normally");
                    break;
        case 2 :    puts("Problem occurred...");
                    sockerr( &usock );
                    break;
    }
    exit( (status == 1 ) ? 0 : 1 );
}


/* the results of running 'TEST sunee 25' are :

    Attempting to open 'sunee' on port 25
    Socket is established
    Connection closed normally
*/
```

usr_init *pointer to user-defined configuration function*

void (*usr_init)(char *name, char *value);

As described under sock_init, a configuration file is parsed line by line. As the file is read, each line is converted into directives, configuration data and trailing comments. The comments are ignored, as are lines not containing both directives and data. The kernal capitalizes the directives for easy comparison while the data are left case sensitive. Both directives and data are passed to an internal function which handles all standard directives and passes all unknown directives and data to an optional user function, allowing application programmers to extend the configuration file. Users are asked to chain their configuration patches as shown in the example. If you intend to distribute your program, you are encouraged to register your directives with me. The user application may use strupr or truncate or otherwise mutilate the string, but it must not extend the string using concatenation. For more information, see the appendix on configuration.

The following example show how you can read parameters from the config file.

```
#include <stdio.h>
#include <tcp.h>

static void (*other_init)();
static void my_init(char *name, char *value)
{
    printf("Was passed '%s' = '%s' \n\r", name, value );

    if ( !strcmp( name , "GOOFBALL" ))
          puts("Found the goofball parameter" );
    else if (other_init)
          (*other_init)( name, value );
}

main(int argc, char **argv)
{
    other_init = usr_init;          /* save old init function for chaining */
    usr_init = my_init;             /* declare mine to be default */
    sock_init();

    exit(0);
}
```

Here is a sample WATTCP.CFG file

```
my_ip=129.97.176.99
netmask=255.255.254.0
nameserver=129.97.128.24

#note ordering is not important
```

```
goofballer=steve
goOFball=joe
goofballow=tom

domainslist="waterloo.edu"
gateway=129.97.176.139
```

This is the output from the program, note that the 'name' parameters are converted to capitals to simplify comparisons but the parameters are left in lower case. You may perform strlwr or strupr on either parameter if you so choose.

```
Was passed 'GOOFBALLER' = 'steve'
Was passed 'GOOFBALL' = 'joe'
Found the goofball parameter
Was passed 'GOOFBALLOW' = 'tom'
```

Index