The Simple Sockets Library

Version 7

Charles E. Campbell, Jr., Ph.D. Terry McRoberts

September 29, 2009

Abstract

The *Simple Sockets Library* (SSL) allows C programmers to use interprocess communications via Berkeley sockets simply and quickly. The programmer is able to move information easily between processes on the same or different machines connected via Ethernet (tm) and using the TCP/IP protocol. Most of the SSL's functions resemble C's file i/o functions, so C programmers will find the SSL easy to learn.

The SSL currently runs under various UNIX (tm) machines: IRIX (tm) (Silicon Graphics), SunOS (tm) (Sun), Domain O/S (tm) (Apollo), Ultrix (tm) (Dec), AIX (tm) (IBM), SCO (tm) , OSF (tm) , and VMS 5.x (tm) , and MS-DOS (tm) (using Borland C++ 5.0).

Keywords

Socket Interprocess Communications
Berkeley Sockets
TCP/IP
Machine Control
Intermachine Communications
Ethernet
Robotics

Contents

1 Copyright

2 Introduction

The *Simple Sockets Library* (henceforth to be abbreviated as the SSL) allows C programmers to develop systems of cooperating programs using Berkeley streaming Sockets running under the TCP/IP protocol over Ethernet. The SSL provides simple ways to move information between programs running on the same or different machines and does so with little overhead. During experiments with the *Sreadbytes* and *Swrite* functions, for example, between two machines, a $44,000$ bytes/second transfer rate was achieved with 4 bytes per packet and over $1,000,000$ bytes/second was achieved with 128 bytes per packet. Of course, heavily loaded networks will affect the rate individual users achieve. The SSL provides *two-way* communications.

The SSL was designed to resemble the FILE i/o system provided by the standard C libraries (ie. fopen, fclose, fputs, etc. map to SSL analogs Sopen, Sclose, Sputs, etc.). Thus, C programmers typically find the SSL easy to learn.

A good analogy for the SSL refers to the phone system. There are three types of Sockets supported by the SSL that concern the user: a *server*, a *client*, and an *accept* socket. A server is analogous to someone waiting by several phones for a call, a client is analogous to someone making a call, and an accept is analogous to a (server) person accepting the call by picking up one of the phones.

Thus, one must have a server for a client to make a successful connection. The server must in turn accept the connection. Unlike our overworked server person, however, the computer using the SSL is perfectly happy handling multiple "accept" Sockets concurrently.

The SSL itself is provided in three main parts (see Table 1). The *Library* is composed of C functions and the *PortMaster* and the *Utilities* are self-contained programs.

The SSL was used at the Intelligent Robotics Laboratory (IRL) since May, 1991. The software appears to be stable at the current writing and reasonably robust. For example, the PortMaster, a background process provided in the SSL distribution, is resistant to hanging (and thereby being unresponsive). It uses *Stimeoutwait*s (see the Reference Manual) to return to its normal quiescent mode when communications with one of its clients ceases

Table 1: **The Three Parts of the SSL**

Library The functions that the users will link to are in the library.

PortMaster The PortMaster is a däemon program which runs in the background. It allows *clients* to connect to *servers* using any available ports.

Utilities There are several utility programs provided with the SSL: `sktdbg`, `srmsrvr`, and `spmtable`.

unexpectedly.

The IRL uses the SSL to control its three robots; its use of the Ethernet is consequently somewhat heavy. Those of you who wish to control machinery or otherwise present a heavy load may wish to purchase a "bridge" to isolate your machines from your local network. This isolation will benefit both the you and the network, as the heavy user will not be afflicted with slowdowns due to the network and the network will not be bothered with lots of packets which slows all its users down.

3 The Library

The user of the SSL accesses it by linking his or her program to a *library*. A software library is typically a single file which consists of some linkage information along with pre-compiled *object code*; unfortunately, methods for producing a library and linking to it vary from system to system and compiler to compiler. Directions for several systems appear in the Section on Implementation.

Practically speaking, a library is composed of a number of pre-compiled C functions to be utilized by the programmer. The calls made available in the SSL are reminiscent of those used by the various file functions in the C language (see Table 2).

Table 2: Simple Sockets and Related File I/O Functions

File Function	*Socket Function*
fopen	Sopen
fclose	Sclose
fread	Sread
fwrite	Swrite
fgets	Sgets
fputs	Sputs
fprintf	Sprintf
fscanf	Sscanf

The Sockets library depends upon the *PortMaster* running on your host machine. Only one copy of the PortMaster need be running at any time on a given machine, and may be started up by anyone by running the *Spm* program in the background. On Unix, the *Spm* program will put itself into the background. One may determine if your system has a PortMaster running by typing *spmtable* first – it will inform you of any existing servers if the PortMaster is up or tell you that the PortMaster is not up otherwise.

A new feature of the SSL allows servers to share another machine's PortMaster. Thus, the SSL supports both distributed and centralized socket name to port mapping. If the machine on which a server is to run doesn't have a PortMaster, such as older MSDOS machines, then it can "borrow" another machine's PortMaster.

A program may open a server Socket, a client Socket, or an accept Socket. Following the phone system analogy, assume that a server Socket is opened first. Then, a different program may open a client Socket which attempts to connect to the server – it succeeds when the server program "notices" that there is a connection waiting on the server Socket and then generates an accept Socket. The client Socket can open, however, before the server accepts the client; there just won't be any two-way communication until the server does accept the client.

3.1 User Guide

Software using C must `#include "sockets.h"` and must link to the SSL. Directions for doing so will vary from system to system and also depend upon where the installer placed the libraries. Please refer to your compiler documentation and the SSL installer to know how to set up library linkage paths and for the include path to the *sockets.h* file.

The software below illustrates some typical Socket code fragments to get a connected socket, assuming that the `sockets.h` file is on your compiler's include search path. If the `sockets.h` file has been placed in a standard system directory (not wise - updates to the compiler may wipe it off!) then you may have to substitute `#include <sockets.h>` for `#include` "sockets.h"' in the following examples.

Example 3.1 *A server with one accept*

```
#include "sockets.h"
Socket *server;
Socket *skt;
server= Sopen("servername","s");
skt   = Saccept(server);
...
Sclose(skt);
Sclose(server);
```

Note in Example 3.1 that the servername is completely up to the programmer to select. Please note that all servers share the same namespace on any given machine, however.

Example 3.2 *A server with multiple accepts*

```
#include "sockets.h"
Socket *server=NULL;
Socket *skt=NULL;
server= Sopen("servername","s");
do {
  skt= Saccept(server);
  ...
  Sclose(skt);
  } while(whatever);
Sclose(server);
```

Again, the programmer may select any servername. Note that the *accept* Socket ("skt") is re-used in Example 3.2. Actually, coding of multiple accept sockets is usually a bit more involved; see `multiskt.c` in the EXAMPLES directory. I usually set up a doubly-linked list containing a Socket pointer for each such accepted client and put them on the mask (using `Smaskset()`). I can then use `Smaskwait()` or `Smasktimeoutwait()` to wait for any Socket, including the server itself, that needs attention. Upon closing such a Socket, I also use `Smaskunset()` to remove it from the mask and do a double-link deletion.

Example 3.3 *A client*

```
#include "sockets.h"
Socket *client;
while(1) {
  client= Sopen("serverName","c");
  if(client) break;
  sleep(1);
  }
...
Sclose(client);
```

In Example 3.3, the `serverName` may be in the optional form `serverName@machineName`. Without the machine name specification, the PortMaster on the machine that the client is running on will be searched for the server. The `smsrvr.c` and `smclient.c` example programs provided in the SSL distribution give complete illustrations of Examples 3.1 and 3.3.

Note that the Sopen() of a client will fail if the server is not already up; in which case, a null pointer is returned. Example 3.3 illustrates a probing method (in this case, once a second). Example 3.4 shows an option to have Sopen() block until the requested server is available.

Example 3.4 *A blocking until open client*

```
#include "sockets.h"
Socket *client;
client= Sopen("serverName","b");
...
Sclose(client);
```

In Example 3.4, the requested server may or may not be up already. If the server is available, a normal client connection is opened. If the server is *not* available, then the client will block (wait) until the server is brought up. Internally, the PortMaster places any clients that open using this option on a waiting list when the server is not available. When the server connects to the PortMaster, the PortMaster then checks its waiting list for clients who wish to connect.

Example 3.5 *Using Sopenv*

```
#include "sockets.h"
Socket *client;
while(1) {
  client= Sopenv("serverName","c","SKTPATH");
  if(client) break;
  sleep(1);
  }
...
Sclose(client);
```

The SSL allows the programmer to set up a group of machines for making client connections. In Example 3.5, the SSL tries to open a client to a server named `serverName` on any machine on the SKTPATH. The "SKTPATH" is an environment variable typically of the form `machine1:machine2:machine3`, ie. a list of machine names separated by colons. This example also shows a typical client connection attempt and sleep polling loop.

A somewhat larger example is included in the distribution: `multiskt.c`. This example program concurrently accepts multiple clients on a single server, reads any messages sent its way from any of its clients, and sends a modified version back to the sending client. It uses the *Smaskwait* functions to block the multiskt process for friendly multi-tasking. It is a useful program to study in conjunction with the Manual below.

Connected sockets are a pre-requisite for any of the I/O Socket functions described in the next section. Note that the server should have a unique name on any given machine. Servers may have the same name on different machines, although this may be confusing. It is suggested that one use the name of the process (at least as a prefix) for that purpose. Note that the only things that one can do with a server Socket are: open one, close one, generate an accept Socket with one, and test/block on one (for clients requesting connection). The SSL supports up to 10 clients concurrently awaiting acceptance on a server Socket, although many machines will limit that to 5 (silently).

The SSL supports two way communication of data between a client and an accept Socket. Since the processes are not likely to be synchronized, often one process must somehow wait until the other process has sent it something. There are two methods to support this under the SSL: polling and interrupt-driven. The polling method uses a lot of CPU time, but lets the program do other tasks in the meantime. The interrupt-driven approach hangs the process (aka blocks and suspends) until something shows up on the Socket. The SmaskXXX group of functions allows one to hang the process until something shows up on any number of Sockets – sort of a big OR gate!

The SmaskXXX alternative to polling also allows the programmer to insert non-socket file descriptors into the mask via Smaskfdset (and to remove them via Smaskunfdset) such as serial ports (ex. open with DEVICE), graphics queues (ex. SGIs with qgetfd), etc.

3.2 Reference Manual

This section contains a description of the functions available in the Socket Library. The SSL returns three types of Sockets: a server, client, and an accept Socket. A Socket itself is a data structure *type* set up by the `sockets.h` header file. The Socket is deliberately used in a fashion reminiscent of the use of FILE pointers so that the C user who knows how to read and write a file will immediately feel comfortable with reading and writing Sockets.

Return Type	*Function Name*	*Argument List*
Socket *	Saccept	(Socket *skt)
void	Sclose	(Socket *skt)
char *	Sgets	(char *buf, int maxbuf, Socket *skt)
int	Smaskfdisset	(int fd)
void	Smaskfdset	(int fd)
fd_set	Smaskget	()
int	Smaskisset	(Socket *skt)
void	Smaskpop	()
void	Smaskpush	()
void	Smaskset	(Socket *skt)
int	Smasktest	()
void	Smasktime	(long seconds,long useconds)
void	Smaskunset	(Socket *skt)
void	Smaskunfdset	(int fd)
void	Smaskuse	(fd_set usermask)
int	Smaskwait	()
Socket *	Sopen	(char *skthost, char *mode)
Socket *	Sopenv	(char *srvrname,char *ctrl,char *envvar)
int	Speek	(Socket *skt, char *buf, int buflen)
unsigned long	Speeraddr	(Socket *skt)
char *	Speername	(Socket *skt)
void	Sprintf	(Socket *skt, char *fmt,...)
char *	Sprtskt	(Socket *skt)
void	Sputs	(char *buf, Socket *skt)
int	Sread	(Socket *skt, char *buf, int buflen)
int	Sreadbytes	(Socket *skt, char *buf, int buflen)
int	Srmsrvr	(char *skthost)
int	Sscanf	(Socket *skt, char *fmt, ...)
int	Stest	(Socket *skt)
int	Stestfd	(int fd)
int	Stimeoutwait	(Socket *skt,long seconds,long useconds)
int	Svprintf	(Socket *skt, char *fmt, void *args)
int	Swait	(Socket *skt)
int	Swrite	(Socket *skt, char *buf, int buflen)

Socket *Saccept(Socket *skt) This function takes a server Socket and produces a Socket which has accepted a connection. This function may be used as often as wanted on the same server Socket! Note that one must have successfully opened a server prior to using Saccept to accept connections (see Examples 3.1 and 3.2).

If this operation is unsuccessful, then Saccept will return a null Socket pointer.

void Sclose(Socket *skt) This function closes a Socket of any type (server, client, accept). The Sclose function communicates with the local PortMaster whenever a server is closed.

char *Sgets(char *buf, int maxbuf, Socket *skt) This is an I/O function which assumes that one has already opened a connected Socket (*skt*). It will attempt to get a null-terminated string from the Socket (up to maxbuf characters). This call will block (aka hang, sleep) until a string shows up, but otherwise acts much as a fgets() function does.

If Sgets has a socket error, then a null pointer is returned. Otherwise, it returns the "*buf*" pointer.

int Smaskfdisset(int fd) The Smaskfdisset() function works in conjunction with the Smaskwait() function. After the Smaskwait() function returns, one can use Smaskfdisset(skt) to test if a particular file descriptor is read-ready.

void Smaskfdset(int fd) The SmaskXXX functions use a *mask* of type *fd_set* internally. Other operations, such as use of a serial port under Unix or the *GL* queue for SGI's Irises (see their qgetfd() function) use file descriptors. This function allows one to set up the SmaskXXX mask with one or more of those file descriptors.

Smask Smaskget() The SmaskXXX functions use a *mask* of type *fd_set* internally. Other operations, such as use of a serial port under Unix, can also be used to set masks. This function provides access to the mask set up by Smaskset. The mate to this function is Smaskuse().

int Smaskisset(Socket *skt) The Smaskisset() function works in conjunction with the Smaskwait() function. After the Smaskwait() function returns, one can use Smaskisset(skt) to test if a particular Socket is read-ready. It is simpler then and thereby faster than Smasktest().

void Smaskpop() The Smaskpop() function, along with the Smaskpush() function, allows the programmer to manipulate an internal stack of masks. The "top" mask is the current mask.

void Smaskpush() The Smaskpush() function, along with the Smaskpop() function, allows the programmer to manipulate an internal stack of masks. The "top" mask is the current mask.

void Smaskset(Socket *skt) This function is part of the *SmaskXXX* group. Sockets may be set up for Smaskwait'ing one at a time. Note that all Sockets are effectively OR'ed together for blocking; to clear the mask, pass a NULL Socket pointer to Smaskset: Smaskset((Socket *) NULL);.

int Smasktest() This function is part of the *SmaskXXX* group. The Smasktest function supports polling – one may determine if anything is on any of the Smaskset Sockets and immediately receive a positive integer if there is, a 0 if there isn't, and a negative number if there's an error.

void Smasktime(long seconds,long useconds) This function is part of the *SmaskXXX* group. Normally Smaskwait will block "forever" (well, at least as long as the machine stays up!). However, one can specify a time limit – then Smaskwait will return with a value of 0 if the time limit is exceeded. One may specify the time limit in seconds and micro-seconds. To clear the time limit (ie. restore Smaskwait to waiting forever), pass a negative time in either or both arguments to Smasktime. Waiting forever is also set up when both arguments to Smasktime are zero.

void Smaskunfdset(int fd) This function is part of the *SmaskXXX* group. File descriptors may be entered into the mask via the Smaskfdset() function, and this function can remove it from the mask. This function is similar to the Smaskunset() function.

void Smaskunset(Socket *skt) This function is part of the *SmaskXXX* group. Sockets may be removed from the Smaskwait's mask one at a time via this function. Like *Smaskset*, a *Smaskunset((Socket *) NULL);* will clear the entire mask.

void Smaskuse(Smask usermask) This function is part of the *SmaskXXX* group. The internal *mask* in SmaskXXX can be set up by the user via this function. Often, Smaskset will be used to set up an internal mask, Smaskget will be used to obtain the resulting mask, and Smaskuse will be used to apply that mask.

int Smaskwait() This function is part of the *SmaskXXX* group. The *Smaskwait* function will block (aka hang) the process until any Socket set via Smaskset has something waiting on it to be read. Note that Sockets set by Smaskset are retained; ie. they must be explicitly cleared by passing a NULL Socket pointer to Smaskset.

Smaskwait returns a positive number if something is waiting on a Socket, zero if a time-out occurs, and a negative number on an error.

Socket *Sopen(char *skthost, char *mode) This function is the workhorse for opening Sockets. Basically, it can make either a server or a client depending on the mode.

Mode	*Effect*
s	open a server
S	open a server but, on failure, retry after using Srmsrvr
c	open a client
b	open a blocking client
s`###`	open a server with the specified port number
S`###`	like *S* above, but with the specified port
c`###`	open a client to a server with the specified port

The *skthost* is the name of the server. If one is making a *client* Socket, then one may optionally use the form "servername@hostname" for the skthost. The "@hostname" form is not necessary for clients to servers that are on the same host as the client, as the default host is the local host. The "servername@hostname" form makes no sense for servers, and so, for opening servers, Sopen will ignore any "@hostname" portion of skthost.

The *"S"* mode for opening servers, like the *"s"* mode, will cause Sopen to attempt to open a server, but if unsuccessful, Sopen will then use Srmsrvr using the provided skthost and then retry opening the server. In the case of *s*`###` or *S*`###`, the servers are set up with the TCP/IP option SO_REUSEADDR. This option alleviates some of the difficulties associated with re-starting servers with fixed ports.

If the skthost is null or an empty string (""), the PortMaster is bypassed. One must specify a port number to use in that case. A warning is issued (as well as a null `Socket*`) because that is typically a programming error, not a user error.

The PortMaster is also always bypassed for clients which specify port number (*c*`###`). In the latter case, a client connection is immediately attempted to the user-specified port number. If the function is not successful, a null Socket pointer is returned.

If the user-specified server port mode is used with a skthost, the server will be registered with its PortMaster; clients need *not* use specified-port mode to connect (ie. they may set `mode` to `"c"`) so long as they specify the correct name of the server.

PortMaster sharing (by setting the environment variable `PMSHARE`) causes servers to work with the *PMSHARE*-specified machine's PortMaster. Clients should use the name of the machine that the *PortMaster* is on; the eventual connection will be made with the appropriate server program, even though it is running on a different machine than the PortMaster it uses. See Example 4.1 for an illustration of how PortMaster sharing works.

Socket *Sopenv(char *srvrname,char *ctrl,char *envvar) The *Sopenv* function is used to open client Sockets, but is formulated to resemble the *Sopen* function with one additional argument. Assuming your computer system supports the concept of environment variables, the Sopenv function can use an environment variable of your choice to help it open a client. Typically, a SKTPATH environment variable gets set up by the user with a colon-separated list of machine names (ex. setenv SKTPATH "gryphon:dragon:xorn"). The Sopenv function attempts to open a client to a server of the given name on the current machine first, and then attempts to do so on each machine in the given sequence. If `env_var` is NULL or a null string, then `SKTPATH` will be used instead.

If successful, a Socket pointer is returned, otherwise a NULL Socket pointer is returned. Please see Saccept on how to use servers.

int Speek(Socket *skt, char *buf, int buflen) The *Speek* function behaves like Sread — up to *buflen* bytes, the buffer will be filled by whatever is currently on the Socket. This function does not remove those bytes from the Socket, however, and those bytes will be returned again on subsequent Speek, Sread, etc. calls. This function does not block (aka hang, sleep).

It will return EOF on select error, some negative number on recv error, zero if no data is present on the Socket, and a positive count of the available bytes otherwise. In the latter case, *buf* will have a copy of the bytes available.

MS-DOS: Speek will only return a 1 when data is available on the Socket, not the number of bytes available. Furthermore, the buffer *buf* will be a zero-length string with just a null byte.

unsigned long Speeraddr(Socket *skt) The *Speeraddr* function returns the internet address of the peer Socket.

char *Speername(Socket *skt) The *Speername* function returns the name of the peer Socket. Internally, it has three buffers; thus it can be used up to three times in the same printf statement.

void Sprintf(Socket *skt, char *fmt,...) The *Sprintf* function acts in an analogous fashion to fprintf, sprintf, etc, by putting formatted strings, appropriately null byte terminated, through the Socket. Thus, one may use either *Sgets* or an appropriate *Sscanf* to receive the information.

char *Sprtskt(Socket *skt) The *Sprtskt* function returns a string describing the Socket.

void Sputs(char *buf, Socket *skt) The *Sputs* function puts a null byte terminated string on the Socket in a fashion analogous to fputs.

int Sread(Socket *skt, char *buf, int buflen) The *Sread* function is similar to the *read* function in Unix. This function can block (aka hang, sleep) if nothing is on the Socket. Otherwise, it will return up to *buflen* bytes in the buffer "*buf*". This function will return whatever is on the Socket, and doesn't try to insure that buflen bytes are read. It returns the number of bytes read from the Socket.

On error, an EOF is returned.

int Sreadbytes(Socket *skt, char *buf, int buflen) The *Sreadbytes* function behaves much like Sread, and it too can block (aka hang, sleep) if nothing is on the Socket.

However, it will not return until buflen bytes are read from the Socket and placed in "buf".

On error, an EOF is returned.

int Srmsrvr(char *skthost) The *Srmsrvr* function is made available to take care of those situations when skt = Sopen("servername","s") fails because the servername has been inadvertently left in the PortMaster's PortTable. This untoward event can happen, for example, when a process is aborted/exited without calling Sclose on its server(s). The Srmsrvr function will remove a server (or server@host) from the PortMaster's PortTable. Note: it will <u>not</u> close the associated Socket, nor free up any Socket memory, and is <u>not</u> a substitute for Sclose.

The *Srmsrvr* function will return either PM_OK (if successful) or PM_SORRY (otherwise).

int Sscanf(Socket *skt, char *fmt, ...) The *Sscanf* function acts much the same as sscanf and fscanf, taking format strings and additional arguments in a like manner. The arguments Sscanf takes, of course, must all be appropriate pointers. This function will block if insufficient data is available on the socket according to the *fmt*.

The *Sscanf* function returns the number of arguments for which it read data, which may be zero or incomplete if a socket error occurred. Normally, the returned count should equal the number of arguments with which Sscanf was provided.

int Stest(Socket *skt) The *Stest* function allows one to determine if anything is available on the specified Socket, *without blocking*.

It will return EOF on select error, some negative number on recv error, zero if no data is present on the Socket, and a positive count of the available bytes otherwise.

MS-DOS: Stest will only return a 1 when data is available on the Socket, not the number of bytes available.

int Stestfd(int fd) The *Stestfd* function allows one to determine if anything is available on the specified file descriptor, *without blocking*.

It will return EOF on select error, some negative number on recv error, zero if no data is present on the file descriptor, and a positive count of the available bytes otherwise.

MS-DOS: Stestfd will only return a 1 when data is available on the file descriptor, not the number of bytes available.

int Stimeoutwait(Socket *skt,long seconds,long useconds) The *Stimeoutwait* function blocks on the given Socket, but for no longer than the number of seconds plus the number of microseconds specified. The function returns the number of bytes available on the Socket (which may be zero), -1 if there was an error, and -2 on timeout.

int Svprintf(Socket *skt, char *fmt, void *args) The *Svprintf* function reads bytes using a printf family format string from the Socket. As each format code is processed, the associated argument is changed in the *args* vector. It returns the number of format items processed.

int Swait(Socket *skt) The *Swait* function will block (aka hang, sleep) until the specified Socket has data available.

It will return EOF on select error, some negative number on recv error, zero if no data is present on the Socket, and a positive count of the available bytes otherwise.

int Swrite(Socket *skt, char *buf, int buflen) The *Swrite* function will write *buflen* bytes from the *buf* buffer onto the specified Socket.

The function will return a count of the number of bytes transmitted through the Socket, which should be equal to *buflen*. If the output is less than *buflen*, then an error occurred while writing to the Socket.

3.3 Hints

The *Hints* below have been found useful at the Intelligent Robotics Laboratory. They are based on almost a year's experience with the SSL.

1. The "Sscanf" function is dangerous. It will block until all of the format codes in its *fmt* string have been used. If somehow a programmer sends an improper string and Sscanf is used to receive it, then the receiving program will probably hang for quite a while. Dr. Campbell has found that it is somewhat safer to use *Sgets* to get an entire string from the Socket and then use *sscanf* and *stpnxt* to parse it.

2. Polling is unfriendly to other processes – it wastes a tremendous amount of CPU time merely querying the Socket(s) to see if anything is awaiting action. The various blocking functions are much better: Swait, Stimeoutwait, SmaskXXX, etc.

3. Unless your processes will always run on the same kind of machine, portability considerations argue against using Sread and Swrite to move non-character string data around – ie. floats, doubles, and even ints. Even if the programmer tests out the machines and verifies that (s)he can safely move, say, a double around, or even a vector of doubles, data structures (`struct XXX { ... }`) may have "holes" in them placed there for word or byte alignment reasons. These "holes" may not be there (or may be placed differently) by other machines and compilers.

4. The Srmsrvr function is a rather rude function – there is no ownership check of a server. Please use your process name (or your id) as part of your server name so that servers do not clash. Remember: your server's name is in only one "name space" shared by all users on your machine.

5. For those of you who wish to use other services (ex. serial ports) which can use the *select* function provided by TCP/IP , check out Smaskfdset and its mate, Smaskunfdset.

6. The IRL has been using a leading two character command convention: `tp~L~x~y~z rx ry rz` will tell the "left" T3 robot to move to "x y z" (inches) in space with an orientation of "rx ry rz" degrees (roll, pitch, yaw). Longer words provide more readability but take up more bandwidth. Experimentally, Dr. Campbell has found that TCP/IP can transfer about 10,000 packets per second with up to 128 bytes in a packet. Loaded systems will, of course, achieve less throughput.

7. It is a good idea for clients to inform servers that they are quitting. The IRL has used the following convention: `tq` *t*ells the server that the messaging client is *q*uitting. Also, a `tQ` tells the server to shut itself down. The server then emits a `tq` to all of its clients to inform them that the server is shutting down (except for the one client who issued the shutdown command), and then closes itself down with Sclose.

8. The server-client relationship is intimately involved with object-oriented messaging – refer to the literature on object-oriented concepts.

9. The SSL can be used to support a *no startup sequencing* paradigm. Typically, clients who attempt to Sopen and fail should go into a poll - sleep loop:

   Example 3.6 *Client Socket Polling*

   ```
   skt= Sopen("servername","c");
   while(!skt) {
     sleep(1);
     skt= Sopen("servername","c");
     }
   ```

   The *multiskt.c* program illustrates how servers should be able to accept multiple clients using blocking judiciously.

   This non-sequencing approach allows one to bring up programs which have servers and/or clients without worrying about what comes first. For two programs, this isn't terribly serious, but at the IRL numerous programs providing numerous services are available and sequencing would be a nightmare.

10. As alluded to earlier, clients should tell servers and servers should tell clients when they are going down. The program that is *not* quitting should return to an "awaiting another client (or server)" mode (ie. see the preceding item). Thus, a troublesome program can be brought down, the programs to which it is communicating will gracefully terminate their connections, and the programmer can then (hopefully) fix the problem. When the program is restarted, the non-sequencing approach will allow the program to seamlessly re-enter into communications with the other software.

11. When a machine is down, and an attempt to connect a client Socket using either *Sopen* directly or *Sopenv* to a server either normally or possibly on that machine, the Sopen will fail but will take an inordinately long time doing so (connection timeout).

12. When a program dies abruptly, connected Sockets appear to *select*() to have something on them (select is used by Stest, Smasktest, Swait, Smaskwait, Speek, Stimeoutwait). Hence, functions which normally block (Smaskwait, Swait) no longer block. On computers which can MSG_PEEK (Unix, Vms), the Stest, Speek, Sgets, and Sscanf functions will detect this situation and will return an error indication (EOF, NULL pointer, incomplete argument processed count) to indicate that the Socket has a problem (one should then Sclose the affected Socket).

    Furthermore, attempts to write to such dead Sockets may generate SIGPIPEs on Unix boxes. Unix programmers should write and install signal handlers for SIGPIPEs.

13. For multiple concurrent Sockets, one should generate routines for both closing and opening them.

Example 3.7 *Client Opening Function*

```
#include "sockets.h"
typedef struct clientlist_str ClientList;
struct clientlist_str {
    Socket *skt;
    ClientList *nxt,*prv;
    }
ClientList *clhd=NULL;
ClientList *cltl=NULL;
 ...
ClientList *openClient(char *srvrname)
{
Socket *skt;
ClientList *clist;
while(1) {
    skt= Sopen(srvrname,"c");
    if(!skt) sleep(1);
    }
clist= (ClientList *) malloc(sizeof(ClientList));
if(cltl) cltl->nxt= clist;
else     clhd     = clist;
clist->prv= cltl;
clist->nxt= NULL;
cltl      = clist;
Smaskset(skt);
```

```
clist->skt= skt;
return clist;
}
```

The `openClient`() function polls once a second in attempting to open a client Socket. Once it succeeds, a ClientList data structure is allocated, double-linked, and a pointer to it returned. In addition, the new client Socket is added to the Smask.

Example 3.8 *Client Closing Function*

```
void closeClient(ClientList *clist)
{
if(clist) {
    Smaskunset(clist->skt);
    Sclose(clist->skt);
    clist->skt= NULL;
    }
}
```

The `closeClient`() function removes the client Socket from the Smask system, closes the Socket, and then sets the skt pointer to NULL to guarantee that the now dead socket won't be inadvertently re-used somehow.

14. It is usually advisable to use some sort of handshaking protocol between server and client, especially when large data blocks are being moved. The TCP/IP buffers can easily get filled and data get lost otherwise. To facilitate this, the SSL sets sockets up with the TCP_NODELAY option so that small packets are moved out across and not collected (collection of small packets normally improves network efficiency since there's less overhead per byte of data).

4 The PortMaster

Servers have names provided by the program which opens them. When a server is to be opened, the Sopen function temporarily opens a Socket to the PortMaster running on the same machine. The Sopen function then tells the PortMaster the server's *name* and the (random) *port* assigned to the new server, and then closes down the connection. The PortMaster retains a list of all active servers and ports running on its machine.

A client is opened using the Sopen function, too. In that case, a connection is made to the PortMaster on the machine where the requested server is running. The PortMaster then tells the Sopen function the port number associated with the requested server, and then closes down the temporary connection. The Sopen function then attempts to connect to the server using the given port.

The server program can use its server Socket to test if any clients are waiting to be connected to it (via the Stest function). If a client is waiting, then the server's program can *accept* the connection, generating an accept Socket (via Saccept). Once the connection is accepted, the program attempting to open the client Socket will finally receive a Socket pointer.

4.1 Users Guide

The PortMaster is probably the simplest program to run: under UNIX, type `Spm &` and under VMS type `run/detach Spm`. The PortMaster will only allow one copy of itself to run on any given computer. It uses a fixed port address (1750, now registered with IANA) and implements a table of server names mapping to random ports. Although knowledgeable users could change the PortMaster's fixed port, this is discouraged: hopefully, PortMasters will proliferate across the world and, if they all use the same port on their machines, will be able to communicate with one another. In other words, the SSL will be unable to communicate with other SSL systems which use a different port for its PortMaster.

Since MS-DOS is not a multi-tasking operating system, unlike UNIX, VMS, or Amiga-DOS, one cannot have a PortMaster running in the background — there is no background. Hence, MS-DOS programs are currently restricted to using client Sockets only, *unless* they can share another machine's PortMaster (see below).

The PortMaster now supports a firewall: one can instruct the PortMaster to restrict access to a group of machines by their host addresses. See the section on *The PortMaster Firewall* below.

4.2 Sharing PortMasters

PortMasters can now be *shared*. This feature was installed mainly to support MSDOS – since it doesn't have a background process capability, it cannot run PortMasters. Hence, it is normally restricted to running clients only, because servers normally announce their presence to their host's PortMaster. If the environment variable `PMSHARE` is set to some other machine, that other machine's PortMaster will be used by the host's processes attempting to open servers. From the client's viewpoint, the server appears to be on the machine with the PortMaster – the client doesn't need to know about where the server actually is.

Example 4.1 *PortMaster Sharing*

Machine A	**Machine B**	**Machine C**
PMSHARE=machineb		
opens server *ASrvr*		
	B's PortMaster now has *ASrvr* on its list	
		pgm opens client to ASrvr@machineb
	B's PortMaster satisfies client's request	
		pgm has client to *ASrvr* on A

In Example 4.1, three machines are in use. However, machines B and C could have been the same machine. In essence, when A shares B's PortMaster, the servers on A appear to be on B insofar as Sopen'ing a client is concerned. Internally, of course, B's PortMaster knows where the *ASrvr* is, and clients end up being connected to the server on machine A.

4.3 Theory of Operation

Normally, users of the SSL will not need to read this section. However, for those who are curious...

There are two main benefits to using the SSL: the functions are similar to those that the C programmer already knows how to use and hence the SSL has a rapid learning curve, and second, servers are assigned to currently available ports. The PortMaster is integral to providing the second benefit.

Berkeley sockets are assigned ports, which are basically just integers. The port is used internally to assign the flow of data to the correct places. One must assign a specific port to a server or, alternatively, allow the system to assign any available port to it. Clients must use that same port number to connect to the desired server. Hence, the problem with using the "any available port" is how to get the client process to know what the currently assigned port is! Programmers using Berkeley sockets have typically just assigned a fixed port to their server and hard-coded the clients with that port. If some other process just happens to use that same port, then things get messy – re-compile, wait until the other process goes away, etc. On a machine where there are many users, co-operation between the users may be impractical.

The PortMaster solves the problem of associating a server name with a randomly assigned (and available) port, and making that association available to clients. The PortMaster itself uses a fixed port, and so servers and clients always know "where" it is. Whenever a server opens (using the Sopen("XXX","s") call), the Sopen function makes a Socket data structure, creates a Berkeley socket, initializes and binds the socket with any available port, and then connects to the PortMaster, *using its fixed port*. It then sends a "message" (PM_SERVER, which is simply an integer) to the PortMaster, telling it the type of Socket it is. Subsequently, the PortMaster gets the new server's name and the port it is assigned to.

When a client opens, it makes a Socket, initializes a Berkeley socket as a client (using the AF_INET format), gets a "host" entity pointer (the server's machine is known as a host), and then connects to the PortMaster on the host machine. It then sends a "PM_CLIENT" to that PortMaster and the desired server's name; that PortMaster will respond with the associated port number (if there is one). The client then closes down the connection to the target host's PortMaster, and initializes a Berkeley socket with the port number it just received and returns a client Socket pointer to the user.

Thus, the SSL's PortMaster system is a distributed database. There are advantages and disadvantages to this scheme. No PortMaster knows anything about servers residing on other machines; if a machine goes down, other machines which have no need to communicate with the downed machine are not affected.

On the other hand, a single machine with a single PortMaster would obviate the need for clients to know what machines they wish to connect to, and could force servers on all machines to have unique names (as it is, servers can have the same name so long as they run on different machines). As a palliative, the *Sopenv* function supports the concept of a *machine path* via the use of environment variables. Using the Sopenv function, clients can "hunt" down a given server by attempting to open a client one at a time on each machine until a server of that name is found. Thus, the user can designate a local group of machines via an environment variable (typically, the "SKTPATH" environment variable is used for this purpose).

The PortMaster talks to its temporary clients using a relatively simple protocol based on "messages" (integers) defined in the `sockets.h` header file. Table 3 illustrates the protocol. If the PortMaster is using its firewall capability, it immediately checks all of its temporary clients for approved machine status; a PM_SORRY is issued immediately prior to the protocol in Table 3 upon failure to be approved. The connection is then summarily severed.

As indicated in Table 3, the PortMaster handles more events than just the PM_SERVER and PM_CLIENT events mentioned earlier. When a client opens to the PortMaster, it identifies the type of service it wishes by sending that PortMaster a "message": PM_SERVER, PM_CLIENT, PM_CLOSE, PM_TABLE, or PM_QUIT. These messages are discussed below. Whenever the PortMaster does not understand the initial message (due to garbling or whatever), the PortMaster will immediately issue a PM_RESEND message and then wait for data. It will continue doing so for up to PM_MAXTRY times (in the `sockets.h` file as sent, PM_MAXTRY is defined as 20). If the limit of re-tries is reached, the PortMaster will peremptorily close down the connection and go back to its usual quiescent state waiting for a new connection.

The PortMaster will wait no more than TIMEOUT seconds for data, which is set to 20 seconds in `Spm.c` as delivered. If that amount of time elapses, the PortMaster will summarily close down the temporary client and continue for more business.

PM_CLIENT tells the PortMaster that a client is attempting to open. The PortMaster normally responds with a PM_OK and the program attempting to open a client responds with a null-byte terminated string (the requested server name). If the PortMaster finds the requested server in its list, then it responds with a PM_OK and then the port number, otherwise it sends a PM_SORRY.

If the PortMaster is using a firewall, it may respond immediately with a PM_SORRY if the client is originating from an unapproved machine.

PM_CLIENTWAIT acts much like a PM_CLIENT, except that the PortMaster will wait (block) before sending the final PM_OK and port number. Thus clients can request a block until the desired server is up.

PM_CLOSE tells the PortMaster that a server is closing down. The PortMaster will normally respond with a PM_OK; the program closing down the server (via *Sclose*) will then send the port number. The PortMaster will then remove the associated server from its internal list of servers and respond with a PM_OK if the designated server was in its list and a PM_SORRY otherwise.

PM_FWINIT tells the PortMaster to re-read its firewall datafile. The PortMaster will then respond with a PM_OK if it worked or PM_RESEND if something got garbled.

PM_QUIT tells the PortMaster that a shutdown command is to be sent. The PortMaster responds with a PM_OK, and the program shutting down the PortMaster issues a null-byte terminated string, "PortMaster" (just to guarantee that a shutdown is really wanted). The PortMaster will then shutdown. This operation requires some knowledge of the protocol as no SSL function is provided to make shutting down the PortMaster easy. However, the `sktdbg` program does provide an easy way to shut down the PortMaster. Concerned system administrators may wish to remove the "shut" command from `sktdbg`. Shutting down the PortMaster does not affect currently connected Sockets; however, new connections between servers and clients cannot be made. On the other hand, starting up a new PortMaster is easy and anyone may do so – the `Spm` program itself will refuse to allow two PortMasters to run concurrently (and will say so).

PM_SERVER tells the PortMaster that a server has been opened. The PortMaster will normally respond with a PM_OK and then the program opening a server will send a string giving the new server's name and then will send its port.

The PortMaster will then respond with a PM_OK upon success or a PM_SORRY if a server by that name already exists.

If the PortMaster is using a firewall, it may respond immediately with a PM_SORRY if the server is originating from an unapproved machine. When PortMaster sharing is not operating, this event should never occur, as servers can then only be legally generated on the same machine that the PortMaster resides upon. The PortMaster enters its own host onto its internal firewall table whenever use of a firewall is enabled.

PM_TABLE tells the PortMaster to give out a list of the servers it currently knows about. Normally the PortMaster will respond with a PM_OK, a count of servers (in network standard 2-byte format, see the Berkeley "ntohs" function for details), and a sequence of strings giving the server name and associated port number. The `spmtable` program and `sktdbg`'s "table" function use this facility.

4.3.1 The PortMaster Firewall

The PortMaster supports a *firewall* to keep out socket requests from unapproved machines. The firewall consists of a list of machines' internet addresses in a file. One specifies the file by one of two methods:

1. Startup with `Spm -f`*full-path-to-filename*
2. Initialization of an environment variable, SPMFIREWALL, to the *full-path-to-filename*.

The *full-path-to-filename* file contains records of the form:

```
* * number number number number
```

where the four `number`s are the internet addresses of the machines permitted to talk to the PortMaster. The `*` indicates accept any number in that field. The firewall function examines incoming internet addresses by use of the *getpeername* function.

The firewall file, often called *spmfirewall.dat*, may contain blank lines. The `#` is assumed to begin a comment and is stripped off.

5 The Utilities

There are several utilities provided with the SSL, and they are explained below. These are all complete programs in themselves, and can serve as lessons on how to use the SSL.

sktdbg `server-name {s|c}` *sktdbg* is the SSL's testing program. One may open server Sockets, accept Sockets, and client Sockets, test out how other program's Sockets are communicating, etc. See the subsection on *sktdbg* below. The "server-name" may take two forms: `servername` or `servername@machine-name`. The latter form may be used when attempting to open clients to servers residing on machines other than the one the user is currently running `sktdbg` on.

spmchk `[machine]` This program tests the current machine by default, the named machine otherwise, for the presence of a PortMaster. It will return a 0 if the PortMaster is present, or a 1 otherwise, and is useful in scripts:

spmchk || (nohup Spm > /dev/null &)

Translated for non-csh/ksh users: if spmchk finds no PortMaster on the current machine, start one up in the background in "no hangup on user exit" mode, with output headed to the bit bucket. (thanks go to Marty Olevitch for this program).

For VMS users:

```
spmchk:= [fullpath]spmchk.exe
set noon
spmchk
if '$SEVERITY' .ne. 1 then run/detach [fullpath]Spm.exe
set on
```

spmtable `[machine [machine [machine ...]]]` This program lists the servers and port numbers on the requested machines. If no machine is listed, then the current machine will be used.

srmsrvr `[server-name [server-name [server-name ...]]]` Sometimes the all-knowledgeable and omnipotent programmer finds a need to hit the control-c key or otherwise kill a running program with a server Socket. In such cases, the PortMaster does not get informed that the server has gone down. This utility will tell the PortMaster to forcibly remove the named server(s) from its list.

Except when using `sktdbg` to open a server with `sktdbg`, the server names above can also have the form `srvr@machine-name`. This name specifies a server on a specific machine. Note that the machine-name is its normal abbreviation (ie. hostname) or its full name (ie. hostname.someplace.com).

5.1 The `sktdbg` Program

The `sktdbg` program was originally a testing facility to debug the SSL itself, but has proven exceptionally useful in debugging programs which use the SSL. Consequently, it is provided as part of the package.

To start up `sktdbg`, one must choose between starting up a server or attempting to open a client Socket.

Example 5.1 *Starting up a sktdbg Server*
sktdbg servername s

Example 5.2 *Starting up a sktdbg Client*
sktdbg servername c

One may also ask sktdbg to explain itself.

Example 5.3 *sktdbg Explanation*
sktdbg "?"

When the `sktdbg` program is started, the first thing it does is attempt to create the requested type of Socket. Typically, attempts to make a server will succeed; if it doesn't, it will report a warning, apply Srmsrvr(), and try a second time to open the server again. If it fails twice, then usually the PortMaster is not running on your machine.

Attempts to open a client to a non-existent server also yields an error message: "unable to Sopen(srvrname,c)". If the server is up and running, then a client Socket will be set up.

The `sktdbg` program then prints out a little menu of actions that the user can take with it.

The "Enter" prompt now also shows the number of bytes waiting on the queue for reading. This number is updated only when the prompt is generated; ie. it is not a dynamic value, but is often useful.

In Table 4, there are 20 commands available, and are described below.

accept Used by a server Socket to accept clients. Note that one may first do *test* to determine if a client is waiting.

close Used by a server Socket to close down an accept Socket.

fput This function takes a filename, *fput filename*, which sktdbg then opens. Every line in it, minus trailing white space, is sent via Sputs across the Socket. Sktdbg then closes the file.

fwinit This call issues a PM_FWINIT to the local machine's PortMaster. The PortMaster will then re-read the firewall data file that it was optionally started up with (if it had none, then this command will have no effect). The fwinit command is useful for the owner of the firewall data file to change it and have the PortMaster update itself without bringing down the PortMaster and restarting it.

get Uses Sgets to get a null-terminated string from the client. This call will block until something arrives for up to 60 seconds.

isset Uses Smaskwait() with negligible timeout (1 microsecond!) and then uses Smaskisset() to determine if the socket is read-ready. This function was included mostly for testing Smaskisset().

menu Repeats the *Socket Test Commands* menu (just like "?" below).

peek Uses Speek to peek at what the client has sent. This call will not block, even if nothing is there yet.

printf printf your-word Uses Sprintf to send "your-word 7 8. itworked!" through the Socket.

put put your various sundry strings across Uses Sputs to send your strings on the same line through the Socket.

q This function "quits" – uses Sclose to close down all Sockets that `sktdbg` is using and exits. Just like "quit".

quit This function "quits" – uses Sclose to close down all Sockets that `sktdbg` is using and exits. Just like "q".

read This function uses Sread to read whatever is on the Socket, and prints it out assuming that it received a string.

rmsrvr rmsrvr server-name This function uses Srmsrvr to remove a server.

scanf This function accepts a subset of format code: %c, %d, %f, and %s. It will call Sscanf once for each format code given to it.

shutdown This function is *dangerous*! It will shut down the PortMaster on your current machine and perform a quit. Use of this function is strongly discouraged unless it is necessary to pull down a PortMaster.

table The *table* function communicates with the PortMaster and prints out a table of all current servers and their ports (rather like spmtable).

test The *test* function prints out the number of bytes awaiting perusal by a get or read operation. It does not block, even if nothing is waiting – it will return 0 in such a case.

wait This function will block the process until something shows up on the accept or client Socket.

write write your sundry thoughts and words This function will *Swrite* your words, appropriately null-byte terminated, through the Socket. It can be read by the *read* function.

? Repeats the *Socket Test Commands* menu (just like "menu" above).

This program is really quite simple to learn and run, and is useful to help the installer verify that the SSL is installed correctly.

6 Installation Instructions

Installation varies, of course, from operating system to operating system.

6.1 Unix

1. Make a `SKTS` subdirectory somewhere, and extract the contents of the tape into that subdirectory (typically, `tar -xvop`, but systems will vary and you may need to specify a tape drive).

2. If your machine is not ANSI C compliant, but supports prototyping anyway, edit the `sockets.h` file to #define __PROTOTYPE__ for your machine.

3. Type `make all`.

These instructions should result in the SSL being compiled, the utilities being compiled, and the PortMaster being compiled. A *simpleskts.a* library results. Although the PortMaster gets started by the make instructions, you may occasionally need to re-start it (for example, after the computer goes down). System administrators should know how to modify their operating system to automatically bring up the PortMaster during a re-boot.

Users may wish to place

setenv SKTPATH machine:machine:machine...‘	csh users
export SKTPATH=machine:machine:machine...‘	ksh users
SKTPATH:==machine:machine:machine...‘	vms users
set SKTPATH=machine:machine:machine...‘	msdos users

in their .login, .profile, login.com, and autoexec.bat respectively. To link to the SSL, one must include “sockets.h”. For example, under UNIX, a user should have a .HDR subdirectory with all his/her favorite header files and use “`cc ... -I~/.HDR ... simpleskts.a`”.

6.2 Vms

1. Make a `SKTS` subdirectory somewhere, and extract the contents of the tape into that subdirectory. The original tape was a quarter-inch cartridge in Unix tar format, but you may well receive a different format.

2. Type `@makeskts`

The “makeskts” command script will create several subdirectories, compile the SSL, and set up a “simpleskts.olb” library. It will **not** attempt to start up the PortMaster. Change to the “EXE” subdirectory under the SKTS subdirectory. The PortMaster should be started via `run/detach Spm`; you may or may not have sufficient privileges to do so. One may spawn the PortMaster, but as soon as you log off the PortMaster will rudely terminate. Due to the potential problem with privileges, the command script will not start up the PortMaster itself.

In order to make use of command-line arguments for sktdbg, etc, VMS requires that you first make “logical symbols.” In your `<login.com>`, place the following lines (with appropriately modified YOURDIR):

$ spmchk	:==	$[YOURDIR.SKTS.EXE]spmchk.exe
$ spmtable	:==	$[YOURDIR.SKTS.EXE]spmtable.exe
$ spm	:==	$[YOURDIR.SKTS.EXE]spm.exe
$ srmsrvr	:==	$[YOURDIR.SKTS.EXE]srmsrvr.exe

6.3 Windows 95 (or more recent)

As of Version 2.08, the SSL runs under Windows 95. One may have a PortMaster running in the background on those machines, too! Since the present author only has Borland C++, only that compiler has been tested. Other compilers may need to modify the <Sinit.c> file to include any TCP/IP initialization they may require.

1. Set up your system with an appropriate C compiler, Ethernet card, and TCP/IP .
2. Copy the contents of ¡ssl.tar.gz¿ onto your computer.
3. Use gunzip and untar to get the uncompressed files
4. Open a MSDOS console, change directory to ...\COSMIC
5. mkwin95

The smplskts.lib and various *.exe files should be generated in the COSMIC subdirectory. You may well want to have the Simple Sockets Library's PortMaster (Spm) always running in background. To accomplish this:

1. On your workbench, click the right mouse button
2. Select New
3. Select Shortcut
4. Put the fully specified path to Spm.exe in the Command Line dialog box.
5. Put Spm as the shortcut's name
6. Right-mouse click on the Spm shortcut
7. Select Properties
8. Select the Shortcut tab
9. In the Run dialog, left mouse click on the down arrow and select minimized.
10. Bring up Explorer

11. Click on the + in the left hand window near Windows
12. Click on the + in the left hand window near Start Menu
13. Click on the + in the left hand window near Programs
14. Double click on StartUp
15. Put the mouse cursor on the Spm shortcut; press the right mouse button and drag it over the right hand Explorer window.
16. Select Copy Here. You will now need to re-boot your machine for this operation to take effect.

6.4 Older Ms-Dos

The SSL was compiled and tested using the Wollongong TCP/IP package with MicroSoft C (v6.0). Other compilers and TCP/IP packages may require adjustment to the SSL code. You **must** have a TCP/IP package for your machine, however – that is what provides Berkeley sockets over which the SSL provides a convenient overlay. Also, you **must** have an Ethernet card for your machine – that provides the hardware over which the information flows. Neither the TCP/IP package, the C compiler, nor the Ethernet hardware come with the SSL.

1. The SSL makes several assumptions, unfortunately, about where include files are (ex. see *Speek.c*). The installer will undoubtedly need to customize these include lines. These areas will always be in `#ifdef MSDOS` ... `#endif` zones.
2. Acquire a TCP/IP package for your computer (ie. Wollongong).
3. Acquire a C compiler for your computer (ie. Microsoft C, v6.0).
4. Acquire an Ethernet card and TCP/IP address for your computer.
5. Copy the contents of the provided diskette into your subdirectory.
6. For those of you with the software mentioned in Table 5:
   (a) Make a directory: `c:\c600\socket`
   (b) Make a directory: `c:\c600\socket\exe`
   (c) Copy *wintcp.lib* to `c:\c600\socket`
   (d) Copy *socket.lib* to `c:\c600\socket`
   (e) Copy *slibce.lib* to `c:\c600\socket`
   (f) Disable the MSDOS linker – possibly by renaming it to *doslink*. Use the MicroSoft C linker instead.
   (g) Return to the directory where you placed the SSL software.

(h) Type *makeskt* (which will use the *Makeskt.bat* command script). This script will use *Makelib.*, compile, create the *smplskts.lib* library, and generate *sktdbg.exe*, *spmtable.exe*, and *srmsrvr.exe*.

(i) You should be done installing the software at this point.

7. If your compiler is not ANSI C compliant, but supports prototyping, edit the `sockets.h` file to define __PROTOTYPE__ for your machine.

8. Follow your compiler's directions to compile to object file format all the C files in `SKTS` and `SKTS\EXE`.

9. Follow your compiler's directions to set up a "library" (a `.LIB` file) with all the `.OBJ` files in the `SKTS` subdirectory.

10. Change to the `SKTS\EXE` subdirectory and follow your compiler's instructions to link each of the object files, one at a time, to the "smplskts.lib" library, the TCP/IP library, and whatever libraries your compiler normally needs.

The installation should yield a *smplskts.lib* (or something similar) library and three executable files (sktdbg.exe, spmtable.exe, and spmtable.exe).

Note that the "SPM.EXE" file should not be generated; you could generate and execute it, but your clone could not use it as it does not allow the PortMaster to run in the background. No attempt has been made to insure that the PortMaster is actually compilable under MicroSoft C, either. Someday a kludge to allow MS-DOS programs to be their own PortMaster may be developed, but that day is not yet. Using Wollongong TCP/IP , one will be restricted to three (or five if its "kernel" is rebuilt) clients.

The software in Table 5 was used by the IRL and is known to work properly with the SSL. Other software may work but has not been tested.

7 Appendix: Miscellaneous Functions

The SSL uses nine functions from Dr. Campbell's *xtdio* library; these have been gathered together and made part of the *simpleskts.a* library. Several of the functions use a special data file, *rdcolor.dat*.

Note: the *error* and *outofmem* functions claim they will "terminate" under certain conditions. Actually, they will call the function *(*error_exit)(int rtn)*, which by default is the *exit* function. Programmers may easily prevent or otherwise control termination by simply setting *error_exit* to point to some other function.

void error(int severity,char *fmt,...) The *error* function prints out an error, warning, or note type of message to stdout. This function takes four kinds of "severity": SEVERE, ERROR, WARNING, and NOTE. Both the SEVERE and ERROR levels will *terminate* the program. The other two, WARNING and NOTE, will not do so. The termination action tends to enforce a standard for what errors versus warnings mean! If your terminal type is supported by the *rdcolor.dat* file (see rdcolor below), SEVERE and ERROR messages are printed with a red leader, WARNINGs with a yellow leader, and NOTEs with a cyan leader.

FILE *fopenv(char *filename,char *ctrl,char *env_var) The *fopenv* provides a file opening service akin to that provided by the *Sopenv* function for the SSL. The *fopen()* function will be used with the given filename and ctrl strings, and upon failure, the environment variable in *env_var* will be used like a PATH variable in attempts to open the file in one directory after another (until success). If none of the *fopen*s succeed, then the *fopenv* function will return a NULL pointer.

void outofmem(void *ptr,char *fmt,...) The *outofmem* function checks if *ptr* is NULL; if it is not, the function immediately returns.

However, if the *ptr* pointer is in fact NULL, then the *fmt* and any subsequent arguments will be used like a regular printf to give a message to the user. The program will then terminate.

void rdcolor(void) This function reads the *rdcolor.dat* file, which essentially is an extended "termcap" file. It uses the "RDCOLOR" environment variable with *fopenv* to search for the *rdcolor.dat* file. The function uses that file to set up various string variables (RED, CLEAR, – see below).

void rdcputs(char *s,FILE *fp) This function puts a string out to the FILE pointer *fp*, except that those strings (XRED, XUWHITE, etc.) are interpreted using the *rdcolor*-provided string instead.

char *sprt(char *s) This function returns a pointer to a static buffer (there are four of them). The string *s* has all of its characters made into "visible" forms: control characters become Â through Ẑ and characters with ANSI values greater than or equal to 128 are made into `~###` sequences. Characters which are normally visible are left unchanged.

void srmtrblk(char *s) This function removes any trailing "blanks" (white space) from the end of a string.

char *stpblk(char *p) This function returns a pointer to the first non-white space character in the string p (ie. steps past blanks).

char *stpnxt(char *s,char *fmt) This function returns a pointer to the first character in s which would not have been read by a *sscanf(s,fmt,...)* function. Useful for quick and dirty parsing.

char *strnxtfmt(char *fmt) The *strnxtfmt* function returns a pointer to the beginning of the (next) format code substring. If *fmt* is NULL, then the previous format string is used (ie. a pointer to the last *fmt* code is retained and the a pointer to the next format code pointer will be returned). This function bears some similarity to strtok except that it jumps from format code to format code (used by Sscanf).

The *rdcolor.dat* file describes terminal escape sequences to use to control color and a few other actions. Each line which does not begin with a white space character is assumed to be a terminal line: terminal types are separated by vertical bars and the line is terminated by a colon. There are a number of strings defined (see Table 6).

Escape sequences may include the sequence `\e` for the ESCAPE character and `\b` for a blank. Please see the *rdcolor.dat* file for examples.

Table 3: **The PortMaster Protocol**

Event	*Client Sends*	*PortMaster Sends*
PM_CLIENT	”sktname”	PM_OK / PM_RESEND PM_OK / PM_SORRY port
PM_CLIENTWAIT	”sktname”	PM_OK / PM_RESEND PM_OK / PM_SORRY port
PM_CLOSE	port	PM_OK / PM_RESEND PM_OK / PM_SORRY
PM_QUIT	”PortMaster”	PM_OK / PM_RESEND
PM_SERVER	”sktname” port	PM_OK / PM_RESEND PM_OK / PM_SORRY
	PM_OK	
PM_TABLE		PM_OK / PM_RESEND count of servers ”server : port” ...
PM_FWINIT		PM_OK / PM_RESEND

Table 4: sktdbg Help Menu

```
Socket Test Commands
  accept
  close
  fwinit
  fput
  get
  menu
  peek
  printf
  put
  q
  quit
  read
  rmsrvr
  scanf
  shutdown
  table
  test
  wait
  write
  ?
(   0 bytes) Enter:
```

Table 5: **Ms-Dos Software known to work with the SSL**

Software Product	*Company*
MicroSoft C, v6.0	MicroSoft
Win/API for DOS, v4.1	The Wollongong Group, Inc. PO Box 51860 Palo Alto, CA 94303-4374 (415)-962-7100
Wollongong Win/TCP, v4.1.1	The Wollongong Group, Inc.

Table 6: rdcolor.dat strings

String Name	*Function*
BLACK	subsequent chars are black
RED	subsequent chars are red
GREEN	subsequent chars are green
YELLOW	subsequent chars are yellow
BLUE	subsequent chars are blue
MAGENTA	subsequent chars are magenta
CYAN	subsequent chars are cyan
WHITE	subsequent chars are white
UBLACK	subsequent chars are underlined and black
URED	subsequent chars are underlined and red
UGREEN	subsequent chars are underlined and green
UYELLOW	subsequent chars are underlined and yellow
UBLUE	subsequent chars are underlined and blue
UMAGENTA	subsequent chars are underlined and magenta
UCYAN	subsequent chars are underlined and cyan
UWHITE	subsequent chars are underlined and white
RVBLACK	subsequent chars are reverse-video and black
RVRED	subsequent chars are reverse-video and red
RVGREEN	subsequent chars are reverse-video and green
RVYELLOW	subsequent chars are reverse-video and yellow
RVBLUE	subsequent chars are reverse-video and blue
RVMAGENTA	subsequent chars are reverse-video and magenta
RVCYAN	subsequent chars are reverse-video and cyan
RVWHITE	subsequent chars are reverse-video and white
NRML	subsequent chars are “normal”
BOLD	subsequent chars are “bold”
CLEAR	the screen is cleared