

\$SPAD/src/lib sockio-c.c

The Axiom Team

July 29, 2014

**Abstract**

# Contents

1	License	3
---	---------	---

# 1 License

```
/*
Copyright (c) 1991-2002, The Numerical Algorithms Group Ltd.
All rights reserved.
```

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of The Numerical Algorithms Group Ltd. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

```
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER
OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/
```

— \* —

```
/* socket i/o primitives */
```

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

---

The MACOSX platform is broken because no matter what you do it seems to include files from `[[/usr/include/sys]]` ahead of `[[/usr/include]]`. On linux systems these files include themselves which causes an infinite regression of includes that fails. GCC gracefully steps over that problem but the build fails anyway. On

MACOSX the `[/usr/include/sys]` versions of files are badly broken with respect to the `[/usr/include]` versions.

```

— * —

#if defined(MACOSXplatform)
#include "/usr/include/unistd.h"
#else
#include <unistd.h>
#endif
#include <sys/time.h>
#include <sys/stat.h>
#include <errno.h>
#include <string.h>
#if defined(MACOSXplatform)
#include "/usr/include/signal.h"
#else
#include <signal.h>
#endif

#if defined(SGIplatform)
#include <bstring.h>
#endif

#include "com.h"
#include "bsdsignal.h"

#define TotalMaxPurposes 50
#define MaxServerNumbers 100
#define accept_if_needed(purpose) \
    ( purpose_table[purpose] == NULL ? sock_accept_connection(purpose) : 1 )

Sock clients[MaxClients];      /* socket description of spad clients */
Sock server[2];                /* AF_UNIX and AF_INET sockets for server */
Sock *purpose_table[TotalMaxPurposes]; /* table of dedicated socket types */
fd_set socket_mask;            /* bit mask of active sockets */
fd_set server_mask;            /* bit mask of server sockets */
int socket_closed;             /* used to identify closed socket on SIGPIPE */
int spad_server_number = -1;   /* spad server number used in sman */
int str_len = 0;
int still_reading = 0;

#include "bsdsignal.h1"
#include "sockio-c.h1"

void
sigpipe_handler(int sig)

```

```

{
    socket_closed = 1;
}

int
wait_for_client_read(Sock *sock,char *buf,int buf_size,char *msg)
{
    int ret_val;
    switch(sock->purpose) {
    case SessionManager:
    case ViewportServer:
        sock_accept_connection(sock->purpose);
        ret_val = sread(purpose_table[sock->purpose], buf, buf_size, msg);
        sock->socket = 0;
        return ret_val;
    default:
        sock->socket = 0;
        return -1;
    }
}

int
wait_for_client_write(Sock *sock,char *buf,int buf_size,char *msg)
{
    int ret_val;
    switch(sock->purpose) {
    case SessionManager:
    case ViewportServer:
        sock_accept_connection(sock->purpose);
        ret_val = swrite(purpose_table[sock->purpose], buf, buf_size, msg);
        sock->socket = 0;
        return ret_val;
    default:
        sock->socket = 0;
        return -1;
    }
}

int
sread(Sock *sock,char *buf,int buf_size,char *msg)
{
    int ret_val;
    char err_msg[256];
    errno = 0;
    do {
        ret_val = read(sock->socket, buf, buf_size);
    } while (ret_val == -1 && errno == EINTR);
    if (ret_val == 0) {
        FD_CLR(sock->socket, &socket_mask);
        purpose_table[sock->purpose] = NULL;
    }
}

```

```

        close(sock->socket);
        return wait_for_client_read(sock, buf, buf_size, msg);
    }
    if (ret_val == -1) {
        if (msg) {
            sprintf(err_msg, "reading: %s", msg);
            perror(err_msg);
        }
        return -1;
    }
    return ret_val;
}

int
swrite(Sock *sock, char *buf, int buf_size, char *msg)
{
    int ret_val;
    char err_msg[256];
    errno = 0;
    socket_closed = 0;
    ret_val = write(sock->socket, buf, buf_size);
    if (ret_val == -1) {
        if (socket_closed) {
            FD_CLR(sock->socket, &socket_mask);
            purpose_table[sock->purpose] = NULL;
            /*      printf("    closing socket %d\n", sock->socket); */
            close(sock->socket);
            return wait_for_client_write(sock, buf, buf_size, msg);
        } else {
            if (msg) {
                sprintf(err_msg, "writing: %s", msg);
                perror(err_msg);
            }
            return -1;
        }
    }
    return ret_val;
}

int
sselect(int n, fd_set *rd, fd_set *wr, fd_set *ex, void *timeout)
{
    int ret_val;
    do {
        ret_val = select(n, (void *)rd, (void *)wr, (void *)ex, (struct timeval *) timeout);
    } while (ret_val == -1 && errno == EINTR);
    return ret_val;
}

int

```

```

fill_buf(Socket *sock, char *buf, int len, char *msg)
{
    int bytes = 0, ret_val;
    while(bytes < len) {
        ret_val = sread(sock, buf + bytes, len - bytes, msg);
        if (ret_val == -1) return -1;
        bytes += ret_val;
    }
    return bytes;
}

int
get_int(Socket *sock)
{
    int val = -1, len;
    len = fill_buf(sock, (char *)&val, sizeof(int), "integer");
    if (len != sizeof(int)) {
#ifdef DEBUG
        fprintf(stderr, "get_int: caught error\n", val);
#endif
        return -1;
    }
#ifdef DEBUG
    fprintf(stderr, "get_int: received %d\n", val);
#endif
    return val;
}

int
sock_get_int(int purpose)
{
    if (accept_if_needed(purpose) != -1)
        return get_int(purpose_table[purpose]);
    else return -1;
}

int
get_ints(Socket *sock, int *vals, int num)
{
    int i;
    for(i=0; i<num; i++)
        *vals++ = get_int(sock);
    return 0;
}

int
sock_get_ints(int purpose, int *vals, int num)
{
    if (accept_if_needed(purpose) != -1)
        return get_ints(purpose_table[purpose], vals, num);
}

```

```

    return -1;
}

int
send_int(Sock *sock,int val)
{
    int ret_val;
    ret_val = swrite(sock, (char *)&val, sizeof(int), NULL);
    if (ret_val == -1) {
        return -1;
    }
    return 0;
}

int
sock_send_int(int purpose,int val)
{
    if (accept_if_needed(purpose) != -1)
        return send_int(purpose_table[purpose], val);
    return -1;
}

int
send_ints(Sock *sock, int *vals, int num)
{
    int i;
    for(i=0; i<num; i++)
        if (send_int(sock, *vals++) == -1)
            return -1;
    return 0;
}

int
sock_send_ints(int purpose, int *vals, int num)
{
    if (accept_if_needed(purpose) != -1)
        return send_ints(purpose_table[purpose], vals, num);
    return -1;
}

int
send_string_len(Sock *sock,char *str,int len)
{
    int val;
    if (len > 1023) {
        char *buf;
        buf = malloc(len+1);
        strncpy(buf,str,len);
        buf[len]='\0';
        send_int(sock,len+1);
    }
}

```



```

        val = swrite(sock, buf, len+1, NULL);
        free(buf);
    } else {
        static char buf[1024];
        strncpy(buf, str, len);
        buf[len] = '\0';
        send_int(sock, len+1);
        val = swrite(sock, buf, len+1, NULL);
    }
    if (val == -1) {
        return -1;
    }
    return 0;
}

int
send_string(Sock *sock, char *str)
{
    int val, len = strlen(str);
    send_int(sock, len+1);
    val = swrite(sock, str, len+1, NULL);
    if (val == -1) {
        return -1;
    }
    return 0;
}

int
sock_send_string(int purpose, char *str)
{
    if (accept_if_needed(purpose) != -1)
        return send_string(purpose_table[purpose], str);
    return -1;
}

int
sock_send_string_len(int purpose, char * str, int len)
{
    if (accept_if_needed(purpose) != -1)
        return send_string_len(purpose_table[purpose], str, len);
    return -1;
}

int
send_strings(Sock *sock, char ** vals, int num)
{
    int i;
    for(i=0; i<num; i++)
        if (send_string(sock, *vals++) == -1)

```

```

        return -1;
    return 0;
}

int
sock_send_strings(int purpose, char **vals, int num)
{
    if (accept_if_needed(purpose) != -1)
        return send_strings(purpose_table[purpose], vals, num);
    return -1;
}

char *
get_string(Sock *sock)
{
    int val, len;
    char *buf;
    len = get_int(sock);
    if (len < 0) return NULL;
    buf = malloc(len*sizeof(char));
    val = fill_buf(sock, buf, len, "string");
    if (val == -1){
free(buf);
return NULL;
}
#ifdef DEBUG
    fprintf(stderr, "get_string: received \"%s\" \n", buf);
#endif
    return buf;
}

char *
sock_get_string(int purpose)
{
    if (accept_if_needed(purpose) != -1)
        return get_string(purpose_table[purpose]);
    else return NULL;
}

char *
get_string_buf(Sock *sock, char *buf, int buf_len)
{
    int val;
    if(!str_len) str_len = get_int(sock);
    if (str_len > buf_len) {
        val = fill_buf(sock, buf, buf_len, "buffered string");
        str_len = str_len - buf_len;
        if (val == -1)
            return NULL;
    }

```

```

        return buf;
    }
    else {
        val = fill_buf(sock, buf, str_len, "buffered string");
        str_len = 0;
        if (val == -1)
            return NULL;
        return NULL;
    }
}

char *
sock_get_string_buf(int purpose, char * buf, int buf_len)
{
    if (accept_if_needed(purpose) != -1)
        return get_string_buf(purpose_table[purpose], buf, buf_len);
    return NULL;
}

int
get_strings(Sock *sock, char **vals, int num)
{
    int i;
    for(i=0; i<num; i++)
        *vals++ = get_string(sock);
    return 0;
}

int
sock_get_strings(int purpose, char ** vals, int num)
{
    if (accept_if_needed(purpose) != -1)
        return get_strings(purpose_table[purpose], vals, num);
    return -1;
}

int
send_float(Sock *sock, double num)
{
    int val;
    val = swrite(sock, (char *)&num, sizeof(double), NULL);
    if (val == -1) {
        return -1;
    }
    return 0;
}

int
sock_send_float(int purpose, double num)
{

```

```

    if (accept_if_needed(purpose) != -1)
        return send_float(purpose_table[purpose], num);
    return -1;
}

int
send_sfloats(Sock *sock, float *vals, int num)
{
    int i;
    for(i=0; i<num; i++)
        if (send_float(sock, (double) *vals++) == -1)
            return -1;
    return 0;
}

int
sock_send_sfloats(int purpose, float * vals, int num)
{
    if (accept_if_needed(purpose) != -1)
        return send_sfloats(purpose_table[purpose], vals, num);
    return -1;
}

int
send_floats(Sock *sock, double *vals, int num)
{
    int i;
    for(i=0; i<num; i++)
        if (send_float(sock, *vals++) == -1)
            return -1;
    return 0;
}

int
sock_send_floats(int purpose, double *vals, int num)
{
    if (accept_if_needed(purpose) != -1)
        return send_floats(purpose_table[purpose], vals, num);
    return -1;
}

double
get_float(Sock *sock)
{
    int val;
    double num = -1.0;
    val = fill_buf(sock, (char *)&num, sizeof(double), "double");
#ifdef DEBUG
    fprintf(stderr, "get_float: received %f\n", num);
#endif
}

```

```

    return num;
}

double
sock_get_float(int purpose)
{
    if (accept_if_needed(purpose) != -1)
        return get_float(purpose_table[purpose]);
    else return 0.0;
}

int
get_sfloats(Sock *sock, float *vals, int num)
{
    int i;
    for(i=0; i<num; i++)
        *vals++ = (float) get_float(sock);
    return 0;
}

int
sock_get_sfloats(int purpose, float * vals, int num)
{
    if (accept_if_needed(purpose) != -1)
        return get_sfloats(purpose_table[purpose], vals, num);
    return -1;
}

int
get_floats(Sock *sock, double *vals, int num)
{
    int i;
    for(i=0; i<num; i++)
        *vals++ = get_float(sock);
    return 0;
}

int
sock_get_floats(int purpose, double *vals, int num)
{
    if (accept_if_needed(purpose) != -1)
        return get_floats(purpose_table[purpose], vals, num);
    return -1;
}

int
wait_for_client_kill(Sock *sock, int sig)
{

```

```

int ret_val;
switch(sock->purpose) {
case SessionManager:
case ViewportServer:
    sock_accept_connection(sock->purpose);
    ret_val = send_signal(purpose_table[sock->purpose], sig);
    sock->socket = 0;
    return ret_val;
default:
    sock->socket = 0;
    return -1;
}
}

```

```

int
sock_get_remote_fd(int purpose)
{
    if (accept_if_needed(purpose) != -1)
        return purpose_table[purpose]->remote_fd;
    return -1;
}

```

```

int
send_signal(Sock *sock, int sig)
{
    int ret_val;
    ret_val = kill(sock->pid, sig);
    if (ret_val == -1 && errno == ESRCH) {
        FD_CLR(sock->socket, &socket_mask);
        purpose_table[sock->purpose] = NULL;
        /* printf("    closing socket %d\n", sock->socket); */
        close(sock->socket);
        return wait_for_client_kill(sock, sig);
    }
    return ret_val;
}

```

```

int
sock_send_signal(int purpose, int sig)
{
    if (accept_if_needed(purpose) != -1)
        return send_signal(purpose_table[purpose], sig);
    return -1;
}

```

```

int
send_wakeup(Sock *sock)
{
    return send_signal(sock, SIGUSR1);
}

```

```

}

int
sock_send_wakeup(int purpose)
{
    if (accept_if_needed(purpose) != -1)
        return send_wakeup(purpose_table[purpose]);
    return -1;
}

Sock *
connect_to_local_server_new(char *server_name, int purpose, int time_out)
{
    int max_con=(time_out == 0 ? 1000000 : time_out), i, code=-1;
    Sock *sock;
    char name[256];

    make_server_name(name, server_name);
    sock = (Sock *) calloc(sizeof(Sock), 1);
    if (sock == NULL) {
        perror("allocating socket space");
        return NULL;
    }
    sock->socket = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sock->socket < 0) {
        perror("opening client socket");
        return NULL;
    }
    memset(server[1].addr.u_addr.sa_data, 0,
            sizeof(server[1].addr.u_addr.sa_data));
    sock->addr.u_addr.sa_family = AF_UNIX;
    strcpy(sock->addr.u_addr.sa_data, name);
    for(i=0; i<max_con; i++) {
        code = connect(sock->socket, &sock->addr.u_addr,
                       sizeof(sock->addr.u_addr));
        if (code == -1) {
            if (errno != ENOENT && errno != ECONNREFUSED) {
                perror("connecting server stream socket");
                return NULL;
            } else {
                if (i != max_con - 1) sleep(1);
                continue;
            }
        } else break;
    }
    if (code == -1) {
        return NULL;
    }
    send_int(sock, getpid());
    send_int(sock, purpose);
}

```

```

    send_int(sock, sock->socket);
    sock->pid = get_int(sock);
    sock->remote_fd = get_int(sock);
    return sock;
}

Sock *
connect_to_local_server(char *server_name, int purpose, int time_out)
{
    int max_con=(time_out == 0 ? 1000000 : time_out), i, code=-1;
    Sock *sock;
    char name[256];

    make_server_name(name, server_name);
    sock = (Sock *) calloc(sizeof(Sock), 1);
    if (sock == NULL) {
        perror("allocating socket space");
        return NULL;
    }
    sock->purpose = purpose;
    /* create the socket */
    sock->socket = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sock->socket < 0) {
        perror("opening client socket");
        return NULL;
    }
    /* connect socket using name specified in command line */
    memset(server[1].addr.u_addr.sa_data, 0,
           sizeof(server[1].addr.u_addr.sa_data));
    sock->addr.u_addr.sa_family = AF_UNIX;
    strcpy(sock->addr.u_addr.sa_data, name);
    for(i=0; i<max_con; i++) {
        code = connect(sock->socket, &sock->addr.u_addr,
                      sizeof(sock->addr.u_addr));
        if (code == -1) {
            if (errno != ENOENT && errno != ECONNREFUSED) {
                perror("connecting server stream socket");
                return NULL;
            } else {
                if (i != max_con - 1) sleep(1);
                continue;
            }
        } else break;
    }
    if (code == -1) {
        return NULL;
    }
    send_int(sock, getpid());
    send_int(sock, sock->purpose);
    send_int(sock, sock->socket);
}

```



```

    sock->pid = get_int(sock);
    /* fprintf(stderr, "Got int form socket\n"); */
    sock->remote_fd = get_int(sock);
    return sock;
}

/* act as terminal session for sock connected to stdin and stdout of another
   process */
void
remote_stdio(Socket *sock)
{
    char buf[1024];
    fd_set rd;
    int len;
    while (1) {
        FD_ZERO(&rd);
        FD_SET(sock->socket, &rd);
        FD_SET(0, &rd);
        len = sselect(FD_SETSIZE, (fd_set *)&rd, (fd_set *)0, (fd_set *)0, NULL);
        if (len == -1) {
            perror("stdio select");
            return;
        }
        if (FD_ISSET(0, &rd)) {
            fgets(buf, 1024, stdin);
            len = strlen(buf);
            /*
             * gets(buf);
             * len = strlen(buf);
             * (buf+len) = '\n';
             * (buf+len+1) = '\0';
             */
            swrite(sock, buf, len, "writing to remote stdin");
        }
        if (FD_ISSET(sock->socket, &rd)) {
            len = sread(sock, buf, 1024, "stdio");
            if (len == -1)
                return;
            else {
                *(buf + len) = '\0';
                fputs(buf, stdout);
                fflush(stdout);
            }
        }
    }
}

/* initialize the table of dedicated sockets */
void
init_purpose_table(void)

```

```

{
    int i;
    for(i=0; i<TotalMaxPurposes; i++) {
        purpose_table[i] = NULL;
    }
}

int
make_server_number(void )
{
    spad_server_number = getpid();
    return spad_server_number;
}

void
close_socket(int socket_num, char *name)
{
    close(socket_num);
#ifdef RTplatform
    unlink(name);
#endif
}

int
make_server_name(char *name, char * base)
{
    char *num;
    if (spad_server_number != -1) {
        sprintf(name, "%s%d", base, spad_server_number);
        return 0;
    }
    num = getenv("SPADNUM");
    if (num == NULL) {
        /*      fprintf(stderr,
            "\n(AXIOM Sockets) The AXIOM server number is undefined.\n");
        */
        return -1;
    }
    sprintf(name, "%s%s", base, num);
    return 0;
}

/* client Spad server sockets.  Two sockets are created: server[0]
   is the internet server socket, and server[1] is a UNIX domain socket. */
int
open_server(char *server_name)
{
    char *s, name[256];

```

```

init_socks();
bsdSignal(SIGPIPE, sigpipe_handler, RestartSystemCalls);
if (make_server_name(name, server_name) == -1)
    return -2;
/* create the socket internet socket */
server[0].socket = 0;
/* server[0].socket = socket(AF_INET, SOCK_STREAM, 0);
if (server[0].socket < 0) {
    server[0].socket = 0;
} else {
    server[0].addr.i_addr.sin_family = AF_INET;
    server[0].addr.i_addr.sin_addr.s_addr = INADDR_ANY;
    server[0].addr.i_addr.sin_port = 0;
    if (bind(server[0].socket, &server[0].addr.i_addr,
        sizeof(server[0].addr.i_addr))) {
        perror("binding INET stream socket");
        server[0].socket = 0;
        return -1;
    }
    length = sizeof(server[0].addr.i_addr);
    if (getsockname(server[0].socket, &server[0].addr.i_addr, &length)) {
        perror("getting INET server socket name");
        server[0].socket = 0;
        return -1;
    }
    server_port = ntohs(server[0].addr.i_addr.sin_port);
    FD_SET(server[0].socket, &socket_mask);
    FD_SET(server[0].socket, &server_mask);
    listen(server[0].socket, 5);
} */
/* Next create the UNIX domain socket */
server[1].socket = socket(AF_UNIX, SOCK_STREAM, 0);
if (server[1].socket < 0) {
    perror("opening UNIX server socket");
    server[1].socket = 0;
    return -2;
} else {
    server[1].addr.u_addr.sa_family = AF_UNIX;
    memset(server[1].addr.u_addr.sa_data, 0,
        sizeof(server[1].addr.u_addr.sa_data));
    strcpy(server[1].addr.u_addr.sa_data, name);
    if (bind(server[1].socket, &server[1].addr.u_addr,
        sizeof(server[1].addr.u_addr))) {
        perror("binding UNIX server socket");
        server[1].socket = 0;
        return -2;
    }
    FD_SET(server[1].socket, &socket_mask);
    FD_SET(server[1].socket, &server_mask);
    listen(server[1].socket, 5);
}

```

```

    }
    s = getenv("SPADSERVER");
    if (s == NULL) {
/*      fprintf(stderr, "Not a spad server system\n"); */
        return -1;
    }
    return 0;
}

int
accept_connection(Sock *sock)
{
    int client;
    for(client=0; client<MaxClients && clients[client].socket != 0; client++);
    if (client == MaxClients) {
        printf("Ran out of client Sock structures\n");
        return -1;
    }
    clients[client].socket = accept(sock->socket, 0, 0);
    if (clients[client].socket == -1) {
        perror("accept");
        clients[client].socket = 0;
        return -1;
    }
    FD_SET(clients[client].socket, &socket_mask);
    get_socket_type(clients+client);
    return clients[client].purpose;
}

/* reads a the socket purpose declaration for classification */
void
get_socket_type(Sock *sock)
{
    sock->pid = get_int(sock);
    sock->purpose = get_int(sock);
    sock->remote_fd = get_int(sock);
    send_int(sock, getpid());
    send_int(sock, sock->socket);
    purpose_table[sock->purpose] = sock;
    switch (sock->purpose) {
    case SessionManager:
        break;
    case ViewportServer:
        break;
    case MenuServer:
        break;
    case SessionIO:
/*      redirect_stdio(sock); */
        break;
    }
}

```

```

}

int
sock_accept_connection(int purpose)
{
    fd_set rd;
    int ret_val, i, p;
    if (getenv("SPADNUM") == NULL) return -1;
    while (1) {
        rd = server_mask;
        ret_val = sselect(FD_SETSIZE, (fd_set *)&rd, (fd_set *)0, (fd_set *)0, NULL);
        if (ret_val == -1) {
            /* perror ("Select"); */
            return -1;
        }
        for(i=0; i<2; i++) {
            if (server[i].socket > 0 && FD_ISSET(server[i].socket, &rd)) {
                p = accept_connection(server+i);
                if (p == purpose) return 1;
            }
        }
    }
}

/* direct stdin and stdout from the given socket */
void
redirect_stdio(Sock *sock)
{
    int fd;
    /* setbuf(stdout, NULL); */
    fd = dup2(sock->socket, 1);
    if (fd != 1) {
        fprintf(stderr, "Error connecting stdout to socket\n");
        return;
    }
    fd = dup2(sock->socket, 0);
    if (fd != 0) {
        fprintf(stderr, "Error connecting stdin to socket\n");
        return;
    }
    fprintf(stderr, "Redirected standard IO\n");
    FD_CLR(sock->socket, &socket_mask);
}

void
init_socks(void)
{
    int i;
    FD_ZERO(&socket_mask);
    FD_ZERO(&server_mask);

```

```

    init_purpose_table();
    for(i=0; i<2; i++) server[i].socket = 0;
    for(i=0; i<MaxClients; i++) clients[i].socket = 0;
}

/* Socket I/O selection called from the BOOT serverLoop function */

int
server_switch(void)
{
    int ret_val, i, cmd = 0;
    fd_set rd, wr, ex, fds_mask;
    FD_ZERO(&rd);
    FD_ZERO(&wr);
    FD_ZERO(&ex);
    fds_mask = server_mask;
    cmd = 0;
    if (purpose_table[SessionManager] != NULL) {
        FD_SET(0, &fds_mask);
        FD_SET(purpose_table[SessionManager]->socket, &fds_mask);
    }
    while (1) {
        do {
            if (purpose_table[MenuServer] != NULL) {
                FD_SET(purpose_table[MenuServer]->socket, &fds_mask);
            }
            rd = fds_mask;
            ret_val = select(FD_SETSIZE, (void *) &rd, (void *) 0, (void *) 0, (void *) 0);
            if (ret_val == -1) {
                /* perror ("Select in switch"); */
                return -1;
            }
            for(i=0; i<2; i++) {
                if (server[i].socket > 0 && (FD_ISSET(server[i].socket, &rd)))
                    accept_connection(server+i);
            }
        } while (purpose_table[SessionManager] == NULL);
        FD_SET(purpose_table[SessionManager]->socket, &fds_mask);
        if (FD_ISSET(purpose_table[SessionManager]->socket, &rd)) {
            cmd = get_int(purpose_table[SessionManager]);
            return cmd;
        }
        if (FD_ISSET(0, &rd)) {
            return CallInterp;
        }
        if (purpose_table[MenuServer] != NULL &&
            (FD_ISSET(purpose_table[MenuServer]->socket, &rd))) {
            cmd = get_int(purpose_table[MenuServer]);
            return cmd;
        }
    }
}

```

```

    }
}

void
flush_stdout(void)
{
    static FILE *fp = NULL;
    if (fp == NULL) {
        fp = fdopen(purpose_table[SessionIO]->socket, "w");
        if (fp == NULL) {
            perror("fdopen");
            return;
        }
    }
    fflush(fp);
}

void
print_line(char *s)
{
    printf("%s\n", s);
}

typedef union {
    double    f;
    long      l[2];
} DoubleFloat;

double
plus_infinity(void )
{
    static int init = 0;
    static DoubleFloat pinf;
    if (! init) {
        pinf.l[0] = 0x7ff00000;
        pinf.l[1] = 0;
        init = 1;
    }
    return pinf.f;
}

double
minus_infinity(void)
{
    static int init = 0;
    static DoubleFloat minf;
    if (! init) {
        minf.l[0] = 0xfff00000L;
        minf.l[1] = 0;
    }
}

```

```

        init = 1;
    }
    return minf.f;
}

double
NANQ(void)
{
    static int init = 0;
    static DoubleFloat nanq;
    if (! init) {
        nanq.l[0] = 0x7ff80000L;
        nanq.l[1] = 0;
        init = 1;
    }
    return nanq.f;
}

```

---



## References

- [1] nothing