```c
/*
 * This is a free program sample that may be reproduced in any form.
 * The author's information should be retained to preserve its identity.
 *
 * Date written: January 1, 2002
 * Written by: Peraphon Sophatsathit
 * Department of Mathematics, Faculty of Science, Chulalongkorn University.
 * email: Peraphon.S@chula.ac.th, http://pioneer.netserv.chula.ac.th/~sperapho
 *
 * Distributed Operating Systems (2301462) classnote.
 * Description: This sample server module illustrates remote host
 *           communication over standard TCP/IP connection.
 */

#include     <stdio.h>
#include     <stdlib.h>
#include     <string.h>
#include     <errno.h>

/*
 * porting from BSD to SVR4
 */
#ifdef __USE_BSD
#  include   <machine/param.h>
#endif

#include     <sys/types.h>
#include     <sys/socket.h>
#include     <netinet/in.h>
#include     <signal.h>
#include     <unistd.h>
#include     <time.h>

/*
 * prototypes
 */
unsigned int alarm(unsigned int);
void         (*signal(int, void (*disp)(int))) (int);
void         handler(int);
void         t_out(int);
void         bzero(void *, size_t);
int          socket(int, int, int);
int          bind(int, const struct sockaddr *, socklen_t);
int          listen(int, int);
int          accept(int, struct sockaddr *, socklen_t *);
int          proc_loop(int);
int          driver(int, int, int, char *);
int          str_echo(int);
void         clear_buff(char *, int);
```

```
/*
 * globals
 */
#define      Small        20
#define      TRUE         1
#define      FALSE        0
#define      LISTENQ      1024
#define      SERV_PORT    9877
#define      Null_char    '\0'

/*
 * error return code
 */
#define      Normal       0
#define      Err_socket   1
#define      Err_bind     2
#define      Err_listen   3
#define      Err_accept   4
#define      Err_connect  5
#define      Err_write    6
#define      Err_read     7
#define      Err_IP       10
#define      Err_fork     88
#define      Err_usage    99

int          flag = FALSE;

/*
 * The purpose of signal calls employed in this program is to prevent
 * runaway processing.  The user may terminate (kill) the process any time
 * via user command (ctrl C) or timer.  The latter can be set to any
 * positive integer ranging from 0 to N (N is recommneded to be small to
 * have any effect).  All signals may appear to have no effect if control
 * is suspended by '(blocking) read'.  In which case, one must send a
 * message by typing from keyboard to get out of 'read' wait.
 * Note that in order for the signals to have an immediate effect,
 * non-blocking read must be set along with extra precaution to handle
 * any 'non-blocking' timing and synchronization idiosyncracies.
 */

int
main(int ac, char **av)
{
      int          rt_code;
      unsigned int sec;
	signal(SIGINT, handler);
	signal(SIGQUIT, handler);
	signal(SIGALRM, t_out);

	switch (ac)
	{
		case 2:
			sec = atoi(av[1]);
			if (sec > 0)
				alarm(sec);
			break;
		default:
			printf("\nUsage: %s   wait_sec\n\n", av[0]);
			printf("Example: %s   0  (no timer is set)\n", av[0]);
			printf("Example: %s   3  (3 seconds timeout)\n\n", av[0]);
			printf("Wait time for 0 second is recommended.  Any other values\n");
			printf("can be used as a precaution to prevent the process from\n");
			printf("running away, but will cause an abnormal termination.\n");
			printf("However, too long a wait will have no effect if the process\n");
			printf("has already terminated.\n");
			return Err_usage;
	}
	rt_code = proc_loop((int)sec);
	if (rt_code > Normal || flag == TRUE)
		printf("Abnornal termination of RPC loop\n");
	fflush(stdout);
	fflush(stderr);
	return Normal;
}

/*
 * set up standard TCP/IP connection
 */
int
proc_loop(int num)
{
	int			counter = 0;
	int			listenfd, connfd;
	char			buf[BUFSIZ];
	time_t			ticks;
	socklen_t		clilen;
	struct sockaddr_in	servaddr, cliaddr;
```

```
        /*
         * open a socket to accept incoming request from client(s)
         */
        if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        {
                return Err_socket;
        }
        bzero(&servaddr, sizeof(servaddr));
        servaddr.sin_family  = AF_INET;
        servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
        servaddr.sin_port    = htons(SERV_PORT);

        if (bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0)
        {
                return Err_bind;
        }
        if (listen(listenfd, LISTENQ) < 0)
        {
                return Err_listen;
        }
        clilen = sizeof(cliaddr);
        if ((connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &clilen)) < 0)
        {
                return Err_accept;
        }
        ticks  = time(NULL);
        snprintf(buf, sizeof(buf), "%.24s\n", ctime(&ticks));
        if (driver(counter, listenfd, connfd, buf) > 0)
        {
                perror("fork and exec failed");
        }
        close(connfd);
        return Normal;
}

/*
 * The driver function spawns a child process to start a TCP socket to
 * communicate with the client process.
 */
int
driver(int count, int listenfd, int connfd, char *sm)
{
        int     pid = 0;
        int     rt  = 0;
        pid = fork();
        if (pid == 0)
        {
                printf("begin child process\n");
                close(listenfd);
                rt = str_echo(connfd);
        }
        else if (pid > 0)
        {
                printf("parent: spawn succeeded!\n");
        }
        else
        {
                printf("fork failed: parent exiting...\n");
                rt = Err_fork;
        }
        return rt;
}

/*
 * read loop: first send prompt string to client and enter read/receive
 * message loop.  The process terminates when ctrl-D is received or
 * interrupts from pending signals.
 */
int
str_echo(int sockfd)
{
        int     n;
        char    line[BUFSIZ];

        strcpy(line, "begin typing message, ctrl-D to quit\n");
        n = strlen(line);
        write(sockfd, line, n);
        clear_buff(line, n);

        for (; flag == FALSE; )
        {
                if ((n = read(sockfd, line, BUFSIZ)) == 0)
                {
                        printf("connection closed by other end\n");
                        break;
                }
                write(sockfd, line, n);
                fprintf(stdout, "echo> %s", line);
                clear_buff(line, n);
        }
        return Normal;
}
```

```
/*
 * clear R/W buffer to null
 */
void
clear_buff(char *line, int n)
{
	register int  i;

	for (i = 0; i < n; i++)
		line[i] = Null_char;
	return;
}

/*
 * time out by alarm clock
 */
void
t_out(int sig)
{
	signal(SIGALRM, t_out);
	flag = TRUE;
	printf("timeout by ALARM signal\n");
	return;
}

/*
 * interrupt and kill signals
 */
void
handler(int sig)
{
	signal(SIGINT, handler);
	signal(SIGQUIT, handler);
	flag = TRUE;
	printf("receiving INT/QUIT signal\n");
	return;
}
```

```
/*
 * This is a free program sample that may be reproduced in any form.
 * The author's information should be retained to preserve its identity.
 *
 * Date written: January 2, 2002
 * Written by: Peraphon Sophatsathit
 * Department of Mathematics, Faculty of Science, Chulalongkorn University.
 * email: Peraphon.S@chula.ac.th, http://pioneer.netserv.chula.ac.th/~sperapho
 *
 * Distributed Operating Systems (2301462) classnote.
 * Description: This sample client module illustrates remote host
 *              communication over standard TCP/IP connection.
 */

#include        <stdio.h>
#include        <stdlib.h>
#include        <string.h>
#include        <errno.h>

/*
 * porting from BSD to SVR4
 */
#ifdef __USE_BSD
#  include      <machine/param.h>
#endif

#include        <sys/types.h>
#include        <sys/socket.h>
#include        <netinet/in.h>
#include        <arpa/inet.h>
#include        <signal.h>
#include        <unistd.h>
#include        <time.h>

/*
 * prototypes
 */
unsigned int    alarm(unsigned int);
void            (*signal(int, void (*disp)(int))) (int);
void            handler(int);
void            t_out(int);
void            bzero(void *, size_t);
void            *memcpy(void *dest, const void *str, size_t nbytes);
int             socket(int, int, int);
int             connect(int, const struct sockaddr *, socklen_t);
int             inet_pton(int, const char *, void *);
int             inet_aton(const char *, struct in_addr *);
int             proc_loop(char *);
int             str_cli(FILE *, int);
void            clear_buff(char *, int);

/*
 * error return code
 */
#define         Normal          0
#define         Err_socket      1
#define         Err_bind        2
#define         Err_listen      3
#define         Err_accept      4
#define         Err_connect     5
#define         Err_write       6
#define         Err_read        7
#define         Err_IP          10
#define         Err_fork        88
#define         Err_usage       99

/*
 * globals
 */
#define         Small           20
#define         TRUE            1
#define         FALSE           0
#define         SERV_PORT       9877
#define         Null_char       '\0'

int             flag = FALSE;

/*
 * The purpose of signal calls employed in this program is to prevent
 * runaway processing.  The user may terminate (kill) the process any time
 * via user command (ctrl C) or timer.  The latter can be set to any
 * positive integer ranging from 0 to N (N is recommneded to be small to
 * have any effect).  All signals may appear to have no effect if control
 * is suspended by '(blocking) read'.  In which case, one must send a
 * message by typing from keyboard to get out of 'read' wait.
 * Note that in order for the signals to have an immediate effect,
 * non-blocking read must be set along with extra precaution to handle
 * any 'non-blocking' timing and synchronization idiosyncracies.
 */
int
main(int ac, char **av)
{
        char            buf[Small];
        int             rt_code;
        unsigned int    sec = 0;
        signal(SIGINT, handler);
        signal(SIGQUIT, handler);
        signal(SIGALRM, t_out);

        switch (ac)
        {
                case 3:
                        sec = atoi(av[2]);
                        if (sec > 0)
                                alarm(sec);
                case 2:
                        strcpy(buf, av[1]);
                        break;
                default:
                        printf("\nUsage: %s  IPaddress          [ wait_sec ]\n\n", av[0]);
                        printf("Example: %s  161.200.192.17     (default to no timeout)\n", av[0]);
                        printf("Example: %s  161.200.192.17  0  (same as default)\n", av[0]);
                        printf("Example: %s  161.200.192.17  5  (timeout in 5 seconds)\n\n", av[0]);
                        printf("Description: Wait 0 second for timeout which is recommended.\n");
                        printf("Any other values can be used as a precaution to prevent the\n");
                        printf("process from running away, but will cause an abnormal\n");
                        printf("termination.  However, too long a wait will have no effect\n");
                        printf("if the process has already terminated.\n\n");
                        return Err_usage;
        }
        rt_code = proc_loop(buf);
        if (rt_code > Normal || flag == TRUE)
                printf("Abnornal termination of RPC loop\n");
        fflush(stdout);
        fflush(stderr);
        return Normal;
}

/*
 * setup C/S connection
 */
int
proc_loop(char *adr)
{
        int                     sockfd;
        int                     rt;
        struct sockaddr_in      servaddr;

        /*
         * open client socket to communicate with the server via
         * standard TCP connection.
         */
	if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
	{
		return Err_socket;
	}
	bzero(&servaddr, sizeof(servaddr));
	servaddr.sin_family  = AF_INET;
	servaddr.sin_port    = htons(SERV_PORT);
	if (inet_pton(AF_INET, adr, &servaddr.sin_addr) <= 0)
	{
		printf("invalid IP address: <%s>\n", adr);
		return Err_IP;
	}

	if (connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0)
	{
		return Err_connect;
	}
	rt = str_cli(stdin, sockfd);
	return rt;
}

/*
 * returned value:
 *    -1    af does not contain a valid address family
 *     0    src does not contain a character string representing a valid network address
 *     1    network address is successfully converted
 */
int
inet_pton(int family, const char *strptr, void *addrptr)
{
	struct in_addr		in_val;
	int			rt;

	if (family == AF_INET)
	{
		if (inet_aton(strptr, &in_val) == 1)
		{
			memcpy(addrptr, &in_val, sizeof(struct in_addr));
			rt = 1;
		}
		else
		{
			rt = 0;
		}
	}
	else
	{
		errno = EAFNOSUPPORT;
		rt = -1;
	}
	return rt;
}

/*
 * send and receive messages
 */
int
str_cli(FILE *fp, int sockfd)
{
	int	n, m;
	char	sline[BUFSIZ], rline[BUFSIZ];

	/*
	 * read prompt string from host (this must be changed if
	 * different handshake protocol is used
	 */
	n = read(sockfd, rline, BUFSIZ);
	rline[n] = Null_char;
	fputs(rline, stdout);
	clear_buff(rline, BUFSIZ);
	clear_buff(sline, BUFSIZ);
	/*
	 * read from stdin and send it over to server.  Echo the info
	 * getting back from server.
	 */
	while (flag == FALSE && fgets(sline, BUFSIZ, fp) != NULL)
	{
		m = strlen(sline);
		write(sockfd, sline, m);
		if ((n = read(sockfd, rline, BUFSIZ)) == 0)
		{
			printf("connection closed by server\n");
			break;
		}
		fputs(rline, stdout);
		clear_buff(rline, n);
		clear_buff(sline, m);
	}
	return Normal;
}
```

```
/*
 * clear R/W buffer to null
 */
void
clear_buff(char *line, int n)
{
	register int  i;

	for (i = 0; i < n; i++)
		line[i] = Null_char;
	return;
}

/*
 * time out by alarm clock
 */
void
t_out(int sig)
{
	signal(SIGALRM, t_out);
	flag = TRUE;
	printf("timeout by ALARM signal\n");
	return;
}

/*
 * interrupt and kill signals
 */
void
handler(int sig)
{
	signal(SIGINT, handler);
	signal(SIGQUIT, handler);
	flag = TRUE;
	printf("receiving INT/QUIT signal\n");
	return;
}
```