7. Appendix A

7a. defs.h

```
/*
 * Definitions for TFTP client and server.
 */

#include        <stdio.h>
#include        <sys/types.h>
#include        <setjmp.h>

#include        "systype.h"

#defineMAXBUFF              2048   /* transmit and receive buffer length */
#defineMAXDATA               512   /* max size of data per packet to send or rcv */
                                  /* 512 is specified by the RFC */
#defineMAXFILENAME           128   /* max filename length */
#defineMAXHOSTNAME           128   /* max host name length */
#defineMAXLINE               512   /* max command line length */
#defineMAXTOKEN   128    /* max token length */

#defineTFTP_SERVICE        "tftp"   /* name of the service */
#defineDAEMONLOG           "/tmp/tftpd.log"
                           /* log file for daemon tracing */

/*
 * Externals.
 */

extern char     command[ ];   /* the command being processed */
extern int      connected;    /* true if we're connected to host */
extern char     hostname[ ];  /* name of host system */
extern int      inetdflag;    /* true if we were started by a daemon */
extern int      interactive;  /* true if we're running interactive */
extern jmp_buf          jmp_mainloop;          /* to return to main command loop */
extern int      lastsend;     /* #bytes of data in last data packet */
extern FILE     *localfp;     /* fp of local file to read or write */
extern int      modetype;     /* see MODE_xxx values */
extern int      nextblknum;   /* next block# to send/receive */
extern char     *pname;                /* the name by which we are invoked */
extern int      port;         /* port number - host byte order */
                                      /* 0 -> use default */
extern char     *prompt;      /* prompt string, for interactive use */
extern long     totnbytes;    /* for get/put statistics printing */
extern int      traceflag;    /* -t command line option, or "trace" cmd */
```

```
extern int      verboseflag;    /* -v command line option */

#defineMODE_ASCII       0       /* values for modetype */
                                /* ascii == netascii */
#defineMODE_BINARY      1       /* binary == octet */

/*
 * One receive buffer and one transmit buffer.
 */

extern char     recvbuff[ ];
extern char     sendbuff[ ];
extern int      sendlen;        /* #bytes in sendbuff[ ] */

/*
 * Define the tftp opcodes.
 */

#defineOP_RRQ           1       /* Read Request */
#defineOP_WRQ           2       /* Write Request */
#defineOP_DATA          3       /* Data */
#defineOP_ACK           4       /* Acknowledgment */
#defineOP_ERROR 5       /* Error*/

#defineOP_MIN           1       /* minimum opcode value */
#defineOP_MAX           5       /* maximum opcode value */

extern int      op_sent;        /* last opcode sent */
extern int      op_recv;        /* last opcode received */

/*
 * Define the tftp error codes.
 * These are transmitted in an error packet (OP_ERROR) with an
 * optional netascii Error Message describing the error.
 */

#defineERR_UNDEF 0      /* not defined */
#defineERR_NOFILE       1       /* File not found */
#defineERR_ACCESS       2       /* Access violation */
#defineERR_NOSPACE      3       /* Disk full or allocation exceeded */
#defineERR_BADOP 4      /* Illegal tftp operation */
#defineERR_BADID 5      /* Unknown TID (port#) */
#defineERR_FILE  6      /* File already exists */
#defineERR_NOUSER       7       /* No such user */
```

```
/*
 * Debug macros, based on the trace flag (-t command line argument,
 * or "trace" command).
 */

#defineDEBUG1(fmt, arg1)   if (traceflag) { \
                                fprintf(stderr, fmt, arg1); \
                                fputc('\n', stderr); \
                                fflush(stderr); \
                           } else ;

#defineDEBUG2(fmt, arg1, arg2)     if (traceflag) { \
                                fprintf(stderr, fmt, arg1, arg2); \
                                fputc('\n', stderr); \
                                fflush(stderr); \
                           } else ;

/*
 * Define macros to load and store 2-byte integers, since these are
 * used in the TFTP headers for opcodes, block numbers and error
 * numbers.  These macros handle the conversion between host format
 * and network byte ordering.
 */

#define ldshort(addr)           ( ntohs (*( (u_short *)(addr) ) ) )
#definestshort(sval,addr)       ( *( (u_short *)(addr) ) = htons(sval) )

#ifdef  lint                /* hush up lint */
#undef ldshort
#undef stshort
short   ldshort();
#endif  /* lint */

/*
 * Datatypes of functions that don't return an int.
 */

char    *gettoken();
FILE    *file_open();
double t_getrtime();    /* our library routine to return elapsed time */
char    *sys_err_str(); /* our library routine for system error messages */
```

7b. netudp.c

```
/*
 * TFTP network handling for UDP/IP connection.
 */

#include        "netdefs.h"

#include        <netinet/in.h>
#include        <arpa/inet.h>
#include        <errno.h>
extern int      errno;

#ifndef CLIENT
#ifndef SERVER
either CLIENT or SERVER must be defined
#endif
#endif

int     sockfd = -1;                            /* fd for socket of server */
char    openhost[MAXHOSTNAMELEN] = { 0 };  /* remember host's name */

extern int                      traceflag;      /* TFTP variable */

#ifdef  CLIENT

extern struct sockaddr_in       udp_srv_addr; /* set by udp_open() */
extern struct servent           udp_serv_info;          /* set by udp_open() */
static int                      recv_first;


/*
 * Open the network connection.
 */

int
net_open(host, service, port)
char    *host;          /* name of other system to communicate with */
char    *service;       /* name of service being requested */
int     port;           /* if > 0, use as port#, else use value for service */
{
        struct sockaddr_in      addr;

        /*
         * Call udp_open() to create the socket.  We tell udp_open to
         * not connect the socket, since we'll receive the first response
```

```
         * from a port that's different from where we send our first
         * datagram to.
         */

        if ( (sockfd = udp_open(host, service, port, 1)) < 0)
                return(-1);

        DEBUG2("net_open: host %s, port# %d",
                        inet_ntoa(udp_srv_addr.sin_addr),
                        ntohs(udp_srv_addr.sin_port));

        strcpy(openhost, host);                 /* save the host's name */
        recv_first = 1;                 /* flag for net_recv() */

        return(0);
}

/*
 * Close the network connection.
 */

net_close()
{
        DEBUG2("net_close: host = %s, fd = %d", openhost, sockfd);

        close(sockfd);

        sockfd = -1;
}

/*
 * Send a record to the other end.
 * We use the sendto() system call, instead of send(), since the address
 * of the server changes after the first packet is sent.
 */

net_send(buff, len)
char    *buff;
int     len;
{
        register int    rc;

        DEBUG3("net_send: sent %d bytes to host %s, port# %d",
                        len, inet_ntoa(udp_srv_addr.sin_addr),
                        ntohs(udp_srv_addr.sin_port));
```

```
        rc = sendto(sockfd, buff, len, 0, (struct sockaddr *) &udp_srv_addr,
                                        sizeof(udp_srv_addr));
        if (rc < 0)
                err_dump("send error");
}

/*
 * Receive a record from the other end.
 */

int                             /* return #bytes in packet, or -1 on EINTR */
net_recv(buff, maxlen)
char    *buff;
int     maxlen;
{
        register int            nbytes;
        int                     fromlen;        /* value-result parameter */
        struct sockaddr_in      from_addr;      /* actual addr of sender */

        fromlen = sizeof(from_addr);
        nbytes = recvfrom(sockfd, buff, maxlen, 0,
                                (struct sockaddr *) &from_addr, &fromlen);
        /*
         * The recvfrom() system call can be interrupted by an alarm
         * interrupt, in case it times out.  We just return -1 if the
         * system call was interrupted, and the caller must determine
         * if this is OK or not.
         */

        if (nbytes < 0) {
                if (errno == EINTR)
                        return(-1);
                else
                        err_dump("recvfrom error");
        }

        DEBUG3("net_recv: got %d bytes from host %s, port# %d",
                        nbytes, inet_ntoa(from_addr.sin_addr),
                        ntohs(from_addr.sin_port));

        /*
         * The TFTP client using UDP/IP has a requirement.
         * The problem is that UDP is being used for a
         * "connection-oriented" protocol, which it wasn't really
         * designed for.  Rather than tying up a single well-known
         * port number, the server changes its port after receiving
```

```
         * the first packet from a client.
         *
         * The first packet a client sends to the server (an RRQ or a WRQ)
         * must be sent to its well-known port number (69 for TFTP).
         * The server is then to choose some other port number for all
         * subsequent transfers.  The recvfrom() call above will return
         * the server's current address.  If the port number that we
         * sent the last packet to (udp_srv_addr.sin_port) is still equal to
         * the initial well-known port number (udp_serv_info.s_port), then we
         * must set the server's port for our next transmission to be
         * the port number from the recvfrom().
         *
         * Furthermore, after we have determined the port number that
         * we'll be receiving from, we can verify each datagram to make
         * certain its from the right place.
         */

        if (recv_first) {
                /*
                 * This is the first received message.
                 * The server's port should have changed.
                 */

                if (udp_srv_addr.sin_port == from_addr.sin_port)
                        err_dump("first receive from port %d",
                                        ntohs(from_addr.sin_port));

                udp_srv_addr.sin_port = from_addr.sin_port;
                                /* save the new port# of the server */
                recv_first = 0;

        } else if (udp_srv_addr.sin_port != from_addr.sin_port) {
                err_dump("received from port %d, expected from port %d",
                                ntohs(from_addr.sin_port),
                                ntohs(udp_srv_addr.sin_port));
        }

        return(nbytes);         /* return the actual length of the message */
}

#endif  /* CLIENT */

#ifdef  SERVER

#include        <sys/ioctl.h>
```

```
struct sockaddr_in      udp_srv_addr; /* server's Internet socket addr */
struct sockaddr_in      udp_cli_addr;  /* client's Internet socket addr */
struct servent          udp_serv_info;          /* from getservbyname() */

static int      recv_nbytes = -1;
static int      recv_first = 0;

extern char     recvbuff[];             /* this is declared in initvars.c */

/*
 * Initialize the network connection for the server, when it has *not*
 * been invoked by inetd.
 */

net_init(service, port)
char    *service;       /* the name of the service we provide */
int     port;           /* if nonzero, this is the port to listen on;
                           overrides the standard port for the service */
{
        struct servent  *sp;

        /*
         * We weren't started by a master daemon.
         * We have to create a socket ourselves and bind our well-known
         * address to it.
         */

        if ( (sp = getservbyname(service, "udp")) == NULL)
                err_dump("net_init: unknown service: %s/udp", service);

        if (port > 0)
                sp->s_port = htons(port);       /* caller's value */
        udp_serv_info = *sp;                    /* structure copy */

        if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
                err_dump("net_init: can't create datagram socket");

        /*
         * Bind our local address so that any client can send to us.
         */

        bzero((char *) &udp_srv_addr, sizeof(udp_srv_addr));
        udp_srv_addr.sin_family      = AF_INET;
        udp_srv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
        udp_srv_addr.sin_port = sp->s_port;
```

```
        if (bind(sockfd, (struct sockaddr *) &udp_srv_addr,
                                        sizeof(udp_srv_addr)) < 0)
                err_dump("net_init: can't bind local address");
}

/*
 * Initiate the server's end.
 * We are passed a flag that says whether or not we were started
*by a "master daemon," such as the inetd program
* A master daemon will have already waited for a message to arrive
 * for us, and will have already set up the connection to the client.
 * If we weren't started by a master daemon, then we must wait for a
 * client's request to arrive.
 */

int
net_open(inetdflag)
int     inetdflag;      /* true if inetd started us */
{
        register int    childpid, nbytes;
        int             on;

        on = 1;

        if (inetdflag) {
#ifdef  BSD             /* assumes 4.3BSD inetd */
                /*
                 * We want to first receive the message that's waiting
                 * for us on the socket, and then close the socket.
                 * This will let inetd go back to waiting for another
                 * request on our "well-known port."
                 */

                sockfd = 0;     /* descriptor for net_recv() to recvfrom() */

                /*
                 * Set the socket to nonblocking, since inetd won't invoke
                 * us unless there's a datagram ready for us to read.
                 */

                if (ioctl(sockfd, FIONBIO, (char *) &on) < 0)
                        err_dump("ioctl FIONBIO error");

#endif  /* BSD inetd specifics */

        }
```

```
/*
 * Now read the first message from the client.
 * In the inetd case, the message is already here and the call to
 * net_recv() returns immediately.  In the other case, net_recv()
 * blocks until a client request arrives.
 */

recv_first  =  1;          /* tell net_recv to save the address */
recv_nbytes = -1;          /* tell net_recv to do the actual read */
nbytes = net_recv(recvbuff, MAXBUFF);

/*
 * Fork a child process to handle the client's request.
 * In the inetd case, the parent exits, which allows inetd to
 * handle the next request that arrives to this well-known port
 * (inetd's wait mode for a datagram socket).
 * Otherwise the parent returns the child pid to the caller, which
 * is probably a concurrent server that'll call us again, to wait
 * for the next client request to this well-known port.
 */

if ( (childpid = fork()) < 0)
        err_dump("server can't fork");

else if (childpid > 0) {/* parent */
        if (inetdflag)
                exit(0);          /* inetd case; we're done */
        else
                return(childpid);         /* independent server */
}

/*
 * Child process continues here.
 * First close the socket that is bound to the well-known address:
 * the parent will handle any further requests that arrive there.
 * We've already read the message that arrived for us to handle.
 */

if (inetdflag) {
        close(0);
        close(1);
        close(2);
} else {
        close(sockfd);
}
```

```
        errno = 0;                  /* in case it was set by a close() */

        /*
         * Create a new socket.
         * Bind any local port# to the socket as our local address.
         * We don't connect(), since net_send() uses the sendto()
         * system call, specifying the destination address each time.
         */

        if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
                err_dump("net_open: can't create socket");

        bzero((char *) &udp_srv_addr, sizeof(udp_srv_addr));
        udp_srv_addr.sin_family      = AF_INET;
        udp_srv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
        udp_srv_addr.sin_port        = htons(0);
        if (bind(sockfd, (char *) &udp_srv_addr, sizeof(udp_srv_addr)) < 0)
                err_dump("net_open: bind error");

        /*
         * Now we'll set a special flag for net_recv(), so that
         * the next time it's called, it'll know that the recvbuff[]
         * already has the received packet in it (from our call to
         * net_recv() above).
         */

        recv_nbytes = nbytes;

        return(0);
}

/*
 * Close a socket.
 */

net_close()
{
        DEBUG2("net_close: host = %s, fd = %d", openhost, sockfd);

        close(sockfd);

        sockfd = -1;
}

/*
 * Send a record to the other end.
```

```
 * The "struct sockaddr_in cli_addr" specifies the client's address.
 */

net_send(buff, len)
char    *buff;
int     len;
{
        register int    rc;

        DEBUG3("net_send: sent %d bytes to host %s, port# %d",
                        len, inet_ntoa(udp_cli_addr.sin_addr),
                        ntohs(udp_cli_addr.sin_port));

        rc = sendto(sockfd, buff, len, 0, (struct sockaddr *) &udp_cli_addr,
                                sizeof(udp_cli_addr));
        if (rc != len)
                err_dump("sendto error");
}

/*
 * Receive a record from the other end.
 * We're called not only by the user, but also by net_open() above,
 * to read the first datagram after a "connection" is established.
 */

int
net_recv(buff, maxlen)
char    *buff;
int     maxlen;
{
        register int            nbytes;
        int                     fromlen;        /* value-result parameter */
        extern int              tout_flag;      /* set by SIGALRM */
        struct sockaddr_in      from_addr;

        if (recv_nbytes >= 0) {
                /*
                 * First message has been handled specially by net_open().
                 * It's already been read into recvbuff[ ].
                 */

                nbytes = recv_nbytes;
                recv_nbytes = -1;
                return(nbytes);
        }
```

```
again:
        fromlen = sizeof(from_addr);
        nbytes = recvfrom(sockfd, buff, maxlen, 0,
                                (struct sockaddr *) &from_addr, &fromlen);
        /*
         * The server can have its recvfrom() interrupted by either an
         * alarm timeout or by a SIGCLD interrupt.  If it's a timeout,
         * "tout_flag" will be set and we have to return to the caller
         * to let them determine if another receive should be initiated.
         * For a SIGCLD signal, we can restart the recvfrom() ourself.
         */

        if (nbytes < 0) {
                if (errno == EINTR) {
                        if (tout_flag)
                                return(-1);

                        errno = 0;        /* assume SIGCLD */
                        goto again;
                }
                err_dump("recvfrom error");
        }

        DEBUG3("net_recv: got %d bytes from host %s, port# %d",
                        nbytes, inet_ntoa(from_addr.sin_addr),
                        ntohs(from_addr.sin_port));

        /*
         * If "recv_first" is set, then we must save the received
         * address that recvfrom() stored in "from_addr" in the
         * global "udp_cli_addr".
         */

        if (recv_first) {
                bcopy((char *) &from_addr, (char *) &udp_cli_addr,
                                                sizeof(from_addr));
                recv_first = 0;
        }

        /*
         * Make sure the message is from the expected client.
         */

        if (udp_cli_addr.sin_port != 0 &&
           udp_cli_addr.sin_port != from_addr.sin_port)
                err_dump("received from port %d, expected from port %d",
```

```
                ntohs(from_addr.sin_port), ntohs(udp_cli_addr.sin_port));

        return(nbytes);         /* return the actual length of the message */
}

#endif  /* SERVER */
```

7c. maincli.c

```
/*
 * tftp - Trivial File Transfer Program.  Client side.
*/

#include        "defs.h"
#include        <signal.h>

main(argc, argv)
int   argc;
char  **argv;
{
        register int      i;
        register char     *s;
        register FILE     *fp;

        pname = argv[0];

        while (--argc > 0 && (*++argv)[0] == '-')
                for (s = argv[0]+1; *s != '\0'; s++)
                        switch (*s) {

                        case 'h':                       /* specify host name */
                                if (--argc <= 0)
                                  err_quit("-h requires another argument");
                                strcpy(hostname, *++argv);
                                break;

                        case 't':
                                traceflag = 1;
                                break;

                        case 'v':
                                verboseflag = 1;
                                break;

                        default:
```

```
                        err_quit("unknown command line option: %c", *s);
                }

        /*
         * For each filename argument, execute the tftp commands in
         * that file.  If no filename arguments were specified on the
         * command line, we process the standard input by default.
         */

        i = 0;
        fp = stdin;
        do {
                if (argc > 0 && (fp = fopen(argv[i], "r")) == NULL) {
                        err_sys("%s: can't open %s for reading", argv[i]);
                }

                mainloop(fp);           /* process a given file */

        } while (++i < argc);

        exit(0);
}

mainloop(fp)
FILE    *fp;
{
        int             sig_intr();

        if (signal(SIGINT, SIG_IGN) != SIG_IGN)
                signal(SIGINT, sig_intr);

        /*
         * Main loop.  Read a command and execute it.
         * This loop is terminated by a "quit" command, or an
         * end-of-file on the command stream.
         */

        if (setjmp(jmp_mainloop) < 0) {
                err_ret("Timeout");
        }

        if (interactive)
                printf("%s", prompt);

        while (getline(fp)) {
                if (gettoken(command) != NULL)
```

```
                        docmd(command);

                if (interactive)
                        printf("%s", prompt);
        }
}

/*
 * INTR signal handler.  Just return to the main loop above.
 * In case we were waiting for a read to complete, turn off any possible
 * alarm clock interrupts.
 *
 * With TFTP, if the client aborts a file transfer (such as with
 * the interrupt signal), the server is not notified.  The protocol counts
 * on the server eventually timing out and exiting.
 */

int
sig_intr()
{
        signal(SIGALRM, SIG_IGN);          /* first ignore the signal */
        alarm(0);                          /* then assure alarm is off */

        longjmp(jmp_mainloop, 1);
        /* NOTREACHED */
}
```

7d. cmd.h

```
/*
 * Header file for user command processing functions.
 */

#include        "defs.h"

extern char     temptoken[ ];  /* temporary token for anyone to use */

typedef struct Cmds {
  char   *cmd_name;             /* actual command string */
  int    (*cmd_func)();                 /* pointer to function */
} Cmds;

extern Cmds   commands[ ];
extern int    ncmds;          /* number of elements in array */
```

7e. cmdsubr.c

```
/*
 * Miscellaneous functions for user command processing.
 */

#include        "cmd.h"

        /* all of the following functions are in cmd.c */
int     cmd_ascii(), cmd_binary(), cmd_connect(), cmd_exit(),
        cmd_get(), cmd_help(), cmd_mode(), cmd_put(),
        cmd_status(), cmd_trace(), cmd_verbose();

Cmds  commands[ ] = {       /* keep in alphabetical order for binary search */
                "?",            cmd_help,
                "ascii",cmd_ascii,
                "binary",       cmd_binary,
                "connect",      cmd_connect,
                "exit",         cmd_exit,
                "get",          cmd_get,
                "help",         cmd_help,
                "mode",                 cmd_mode,
                "put",          cmd_put,
                "quit", cmd_exit,
                "status",       cmd_status,
                "trace",        cmd_trace,
                "verbose",      cmd_verbose,
};
```

```
#defineNCMDS        (sizeof(commands) / sizeof(Cmds))

int       ncmds = NCMDS;

static char       line[MAXLINE] = { 0 };
static char       *lineptr = NULL;

/*
 * Fetch the next command line.
 * For interactive use or batch use, the lines are read from a file.
 *
 * Return 1 if OK, else 0 on error or end-of-file.
 */

int
getline(fp)
FILE    *fp;
{
          if (fgets(line, MAXLINE, fp) == NULL)
                    return(0);                /* error or end-of-file */
          lineptr = line;

          return(1);
}

/*
 * Fetch the next token from the input stream.
 * We use the line that was set up in the most previous call to
 * getline().
 *
 * Return a pointer to the token (the argument), or NULL if no more exist.
 */

char *
gettoken(token)
char    token[];
{
          register int      c;
          register char     *tokenptr;

          while ((c = *lineptr++) == ' ' || c == '\t')
                    ;                   /* skip leading white space */

          if (c == '\0' || c == '\n')
                    return(NULL);/* nothing there */
```

```
        tokenptr = token;
        *tokenptr++ = c;        /* first char of token */

        /*
         * Now collect everything up to the next space, tab, newline, or null.
         */

        while ((c = *lineptr++) != ' ' && c != '\t' && c != '\n' && c != '\0')
                *tokenptr++ = c;

        *tokenptr = 0;          /* null terminate token */
        return(token);
}

/*
 * Verify that there aren't any more tokens left on a command line.
 */

checkend()
{
        if (gettoken(temptoken) != NULL)
                err_cmd("trailing garbage");
}

/*
 * Execute a command.
 * Call the appropriate function.  If all goes well, that function will
 * return, otherwise that function may call an error handler, which will
 * call longjmp() and branch back to the main command processing loop.
 */

docmd(cmdptr)
char    *cmdptr;
{
        register int      i;

        if ( (i = binary(cmdptr, ncmds)) < 0)
                err_cmd(cmdptr);

        (*commands[i].cmd_func)();

        checkend();
}

/*
 * Perform a binary search of the command table
```

```
 * to see if a given token is a command.
 */

binary(word, n)
char    *word;
int     n;
{
        register int    low, high, mid, cond;

        low  = 0;
        high = n - 1;
        while (low <= high) {
                mid = (low + high) / 2;
                if ( (cond = strcmp(word, commands[mid].cmd_name)) < 0)
                        high = mid - 1;
                else if (cond > 0)
                        low = mid + 1;
                else
                        return(mid);    /* found it, return index in array */
        }
        return(-1);     /* not found */
}

/*
 * Take a "host:file" character string and separate the "host"
 * portion from the "file" portion.
 */

striphost(fname, hname)
char    *fname;                 /* input:  "host:file" or just "file" */
char    *hname;                 /* store "host" name here, if present */
{
        char            *index();
        register char   *ptr1, *ptr2;

        if ( (ptr1 = index(fname, ':')) == NULL)
                return;         /* there is not a "host:" present */

        /*
         * Copy the entire "host:file" into the hname array,
         * then replace the colon with a null byte.
         */

        strcpy(hname, fname);
        ptr2 = index(hname, ':');
        *ptr2 = 0;      /* null terminates the "host" string */
```

```
        /*
         * Now move the "file" string left in the fname array,
         * removing the "host:" portion.
         */

        strcpy(fname, ptr1 + 1);        /* ptr1 + 1 to skip over the ':' */
}

/*
 * User command error.
 * Print out the command line too, for information.
 */

err_cmd(str)
char    *str;           /* may be a 0-length string, i.e., "" */
{
        fprintf(stderr, "%s: '%s' command error", pname, command);
        if (strlen(str) > 0)
                fprintf(stderr, ": %s", str);
        fprintf(stderr, "\n");
        fflush(stderr);

        longjmp(jmp_mainloop, 1);   /* 1 -> not a timeout, we've already
                                       printed our error message */
}
```

7f. cmd.c

```
/*
 * Command processing functions, one per command.
 * (Only the client side processes user commands.)
 * In alphabetical order.
 */

#include        "cmd.h"

/*
 * ascii
 *
 *      Equivalent to "mode ascii".
 */

cmd_ascii()
{
```

```
        modetype = MODE_ASCII;
}

/*
 * binary
 *
 *      Equivalent to "mode binary".
 */

cmd_binary()
{
        modetype = MODE_BINARY;
}

/*
 * connect <hostname> [ <port> ]
 *
 *      Set the hostname and optional port number for future transfers.
 *      The port is the well-known port number of the tftp server on
 *      the other system.  Normally this will default to the value
 *      specified in /etc/services (69).
 */

cmd_connect()
{
        register int      val;

        if (gettoken(hostname) == NULL)
                err_cmd("missing hostname");

        if (gettoken(temptoken) == NULL)
                return;
        val = atoi(temptoken);
        if (val < 0)
                err_cmd("invalid port number");
        port = val;
}

/*
 * exit
 */

cmd_exit()
{
        exit(0);
}
```

```
/*
 * get <remotefilename> <localfilename>
 */

cmd_get()
{
        char    remfname[MAXFILENAME], locfname[MAXFILENAME];
        char    *index();

        if (gettoken(remfname) == NULL)
                err_cmd("the remote filename must be specified");
        if (gettoken(locfname) == NULL)
                err_cmd("the local filename must be specified");

        if (index(locfname, ':') != NULL)
                err_cmd("can't have 'host:' in local filename");

        striphost(remfname, hostname);        /* check for "host:" and process */
        if (hostname[0] == 0)
                err_cmd("no host has been specified");

        do_get(remfname, locfname);
}

/*
 * help
 */

cmd_help()
{
        register int      i;

        for (i = 0; i < ncmds; i++) {
                printf("  %s\n", commands[i].cmd_name);
        }
}

/*
 * mode ascii
 * mode binary
 *
 *      Set the mode for file transfers.
 */

cmd_mode()
```

```
{
        if (gettoken(temptoken) == NULL) {
                err_cmd("a mode type must be specified");
        } else {
                if (strcmp(temptoken, "ascii") == 0)
                        modetype = MODE_ASCII;
                else if (strcmp(temptoken, "binary") == 0)
                        modetype = MODE_BINARY;
                else
                        err_cmd("mode must be 'ascii' or 'binary'");
        }
}

/*
 * put <localfilename> <remotefilename>
 *
 *      Note that the <remotefilename> may be of the form <host>:<filename>
 *      to specify the host also.
 */

cmd_put()
{
        char    remfname[MAXFILENAME], locfname[MAXFILENAME];

        if (gettoken(locfname) == NULL)
                err_cmd("the local filename must be specified");
        if (gettoken(remfname) == NULL)
                err_cmd("the remote filename must be specified");

        if (index(locfname, ':') != NULL)
                err_cmd("can't have 'host:' in local filename");

        striphost(remfname, hostname);        /* check for "host:" and process */
        if (hostname[0] == 0)
                err_cmd("no host has been specified");

        do_put(remfname, locfname);
}

/*
 * Show current status.
 */

cmd_status()
{
        if (connected)
```

```
                printf("Connected\n");
        else
                printf("Not connected\n");

        printf("mode = ");
        switch (modetype) {
        case MODE_ASCII:   printf("netascii");              break;
        case MODE_BINARY:        printf("octet (binary)");        break;
        default:
                err_dump("unknown modetype");
        }

        printf(", verbose = %s", verboseflag ? "on" : "off");
        printf(", trace = %s\n", traceflag ? "on" : "off");
}

/*
 * Toggle debug mode.
 */

cmd_trace()
{
        traceflag = !traceflag;
}

/*
 * Toggle verbose mode.
 */

cmd_verbose()
{
        verboseflag = !verboseflag;
}
```

7g. cmdgetput.c

```
/*
 * File get/put processing.
 *
 * This is the way the client side gets started - either the user
 * wants to get a file (generates a RRQ command to the server)+
 * or the user wants to put a file (generates a WRQ command to the
 * server).  Once either the RRQ or the WRQ command is sent,
 * the finite state machine takes over the transmission.
 */

#include    "defs.h"

/*
 * Execute a get command - read a remote file and store on the local system.
 */

do_get(remfname, locfname)
char       *remfname;
char       *locfname;
{
    if ( (localfp = file_open(locfname, "w", 1)) == NULL) {
            err_ret("can't fopen %s for writing", locfname);
            return;
}
 if (net_open(hostname, TFTP_SERVICE, port) < 0)
     return;
 totnbytes = 0;
  t_start();                        /* start timer for statistics */

send_RQ(OP_RRQ, remfname, modetype);

fsm_loop(OP_RRQ);

t_stop();                                                   /* stop timer for statistics */

net_close();

file_close(localfp);

printf("Received %ld bytes in %.1f seconds\n", totnbytes, t_getrtime());   /* print stastics */
}
```

```
/*
 * Execute a put command - send a local file to the remote system.
 */

do_put(remfname, locfname)
char        *remfname;
char        *locfname;
{
    if ( (localfp = file_open(locfname, "r", 0)) == NULL) {
          err_ret("can't fopen %s for reading", locfname);
          return;
}

    if (net_open(hostname, TFTP_SERVICE, port) < 0)
  return;
  totnbytes = 0;
  t_start();                        /* start timer for statistics */

  lastsend = MAXDATA;
  send_RQ(OP_WRQ, remfname, modetype);

 fsm_loop(OP_WRQ);

 t_stop();                          /* stop timer for statistics */

 net_close();

  file_close(localfp);

  printf("Sent %ld bytes in %.1f seconds\n", totnbytes, t_getrtime());     /* print stastics */
}
```

7h. file.c

```
/*
 * Routines to open/close/read/write the local file.
 * For "binary" (octet) transmissions, we use the UNIX open/read/write
 * system calls (or their equivalent).
 * For "ascii" (netascii) transmissions, we use the UNIX standard i/o routines
 * fopen/getc/putc (or their equivalent).
 */

#include        "defs.h"

/*
 * The following are used by the functions in this file only.
 */

static int      lastcr   = 0;     /* 1 if last character was a carriage-return */
static int      nextchar = 0;

/*
 * Open the local file for reading or writing.
 * Return a FILE pointer, or NULL on error.
 */

FILE *
file_open(fname, mode, initblknum)
char    *fname;
char    *mode;          /* for fopen() - "r" or "w" */
int     initblknum;
{
        register FILE   *fp;

        if (strcmp(fname, "-") == 0)
                fp = stdout;
        else if ( (fp = fopen(fname, mode)) == NULL)
                return((FILE *) 0);

        nextblknum = initblknum;      /* for first data packet or first ACK */
        lastcr     = 0;               /* for file_write() */
        nextchar   = -1;              /* for file_read() */

        DEBUG2("file_open: opened %s, mode = %s", fname, mode);

        return(fp);
}
```

```
/*
 * Close the local file.
 * This causes the standard i/o system to flush its buffers for this file.
 */

file_close(fp)
FILE    *fp;
{
        if (lastcr)
                err_dump("final character was a CR");
        if (nextchar >= 0)
                err_dump("nextchar >= 0");

        if (fp == stdout)
                return;         /* don't close standard output */
        else if (fclose(fp) == EOF)
                err_dump("fclose error");
}

/*
 * Read data from the local file.
 * Here is where we handle any conversion between the file's mode
 * on the local system and the network mode.
 *
 * Return the number of bytes read (between 1 and maxnbytes, inclusive)
 * or 0 on EOF.
 */

int
file_read(fp, ptr, maxnbytes, mode)
FILE            *fp;
register char   *ptr;
register int    maxnbytes;
int             mode;
{
        register int    c, count;

        if (mode == MODE_BINARY) {
                count = read(fileno(fp), ptr, maxnbytes);
                if (count < 0)
                        err_dump("read error on local file");

                return(count);          /* will be 0 on EOF */

        } else if (mode == MODE_ASCII) {
```

```
                /*
                 * For files that are transferred in netascii, we must
                 * perform the reverse conversions that file_write() does.
                 * We have to use the global "nextchar" to
                 * remember if the next character to output is a linefeed
                 * or a null, since the second byte of a 2-byte sequence
                 * may not fit in the current buffer, and may have to go
                 * as the first byte of the next buffer (i.e., we have to
                 * remember this fact from one call to the next).
                 */

                for (count = 0; count < maxnbytes; count++) {
                        if (nextchar >= 0) {
                                *ptr++ = nextchar;
                                nextchar = -1;
                                continue;
                        }

                        c = getc(fp);

                        if (c == EOF) {         /* EOF return means eof or error */
                                if (ferror(fp))
                                  err_dump("read err from getc on local file");
                                return(count);

                        } else if (c == '\n') {
                                c = '\r';       /* newline -> CR,LF */
                                nextchar = '\n';

                        } else if (c == '\r') {
                                nextchar = '\0';/* CR -> CR,NULL */

                        } else
                                nextchar = -1;

                        *ptr++ = c;
                }

                return(count);
        } else
                err_dump("unknown MODE value");

        /* NOTREACHED */
}
```

```
/*
 * Write data to the local file.
 * Here is where we handle any conversion between the mode of the
 * file on the network and the local system's conventions.
 */

file_write(fp, ptr, nbytes, mode)
FILE            *fp;
register char   *ptr;
register int    nbytes;
int             mode;
{
        register int    c, i;

        if (mode == MODE_BINARY) {
                /*
                 * For binary mode files, no conversion is required.
                 */

                i = write(fileno(fp), ptr, nbytes);
                if (i != nbytes)
                        err_dump("write error to local file, i = %d", i);

        } else if (mode == MODE_ASCII) {
                /*
                 * For files that are transferred in netascii, we must
                 * perform the following conversions:
                 *
                 *      CR,LF           ->  newline = '\n'
                 *      CR,NULL         ->  CR      = '\r'
                 *      CR,anything_else  ->  undefined (we don't allow this)
                 *
                 * We have to use the global "lastcr" to remember
                 * if the last character was a carriage-return or not,
                 * since if the last character of a buffer is a CR, we have
                 * to remember that when we're called for the next buffer.
                 */

                for (i = 0; i < nbytes; i++) {
                        c = *ptr++;
                        if (lastcr) {
                                if (c == '\n')
                                        c = '\n';
                                else if (c == '\0')
                                        c = '\r';
                                else
```

```
                                err_dump("CR followed by 0x%02x", c);
                        lastcr = 0;

                } else if (c == '\r') {
                        lastcr = 1;
                        continue;       /* get next character */
                }

                if (putc(c, fp) == EOF)
                        err_dump("write error from putc to local file");
            }
        } else
            err_dump("unknown MODE value");
}
```

5i. fsm.c

```
/*
 * Finite state machine routines.
 */

#include        "defs.h"
#include        <signal.h>

#include        "rtt.h"                 /* for RTT timing */

#ifdef          CLIENT
int             recv_ACK(), recv_DATA(), recv_RQERR();
#endif

#ifdef          SERVER
int             recv_RRQ(), recv_WRQ(), recv_ACK(), recv_DATA();
#endif

int             fsm_error(), fsm_invalid();

/*
 * Finite state machine table.
 * This is just a 2-d array indexed by the last opcode sent and
 * the opcode just received.  The result is the address of a
 * function to call to process the received opcode.
 */

int                     (*fsm_ptr [ OP_MAX + 1 ] [ OP_MAX + 1 ] ) () = {
```

```
#ifdef          CLIENT

fsm_invalid,                /* [sent = 0]       [recv = 0]*/
fsm_invalid,              /* [sent = 0]       [recv = OP_RRQ]*/
fsm_invalid,              /* [sent = 0]       [recv = OP_WRQ]*/
fsm_invalid,             /* [sent = 0]       [recv = OP_DATA]*/
fsm_invalid,              /* [sent = 0]       [recv = OP_ACK]*/
fsm_invalid,            /* [sent = 0]       [recv = OP_ERROR]*/
fsm_invalid,              /* [sent = OP_RRQ]   [recv = 0]*/
fsm_invalid,            /* [sent = OP_RRQ]   [recv = OP_RRQ]*/
fsm_invalid,            /* [sent = OP_RRQ]   [recv = OP_WRQ]*/
recv_DATA,             /* [sent = OP_RRQ]   [recv = OP_DATA]*/
fsm_invalid,           /* [sent = OP_RRQ]   [recv = OP_ACK]*/
recv_RQERR,          /* [sent = OP_RRQ]   [recv = OP_ERROR]*/

fsm_invalid,           /* [sent = OP_WRQ]   [recv = 0]*/
fsm_invalid,             /* [sent = OP_WRQ]   [recv = OP_RRQ]*/
fsm_invalid,             /* [sent = OP_WRQ]   [recv = OP_WRQ]*/
fsm_invalid,           /* [sent = OP_WRQ]   [recv = OP_DATA]*/
recv_ACK,              /* [sent = OP_WRQ]   [recv = OP_ACK]*/
recv_RQERR,          /* [sent = OP_WRQ]   [recv = OP_ERROR]*/

fsm_invalid,              /* [sent = OP_DATA]  [recv = 0]*/
fsm_invalid,            /* [sent = OP_DATA]  [recv = OP_RRQ]*/
fsm_invalid,            /* [sent = OP_DATA]  [recv = OP_WRQ]*/
fsm_invalid,           /* [sent = OP_DATA]  [recv = OP_DATA]*/
recv_ACK,              /* [sent = OP_DATA]  [recv = OP_ACK]*/
fsm_error,           /* [sent = OP_DATA]  [recv = OP_ERROR]*/

fsm_invalid,              /* [sent = OP_ACK]   [recv = 0]*/
fsm_invalid,            /* [sent = OP_ACK]   [recv = OP_RRQ]*/
fsm_invalid,            /* [sent = OP_ACK]   [recv = OP_WRQ]*/
recv_DATA,             /* [sent = OP_ACK]   [recv = OP_DATA]*/
fsm_invalid,            /* [sent = OP_ACK]   [recv = OP_ACK]*/
fsm_error,           /* [sent = OP_ACK]   [recv = OP_ERROR]*/

fsm_invalid,             /* [sent = OP_ERROR] [recv = 0]*/
fsm_invalid,           /* [sent = OP_ERROR] [recv = OP_RRQ]*/
fsm_invalid,           /* [sent = OP_ERROR] [recv = OP_WRQ]*/
fsm_invalid,          /* [sent = OP_ERROR] [recv = OP_DATA]*/
fsm_invalid,           /* [sent = OP_ERROR] [recv = OP_ACK]*/
fsm_error            /* [sent = OP_ERROR] [recv = OP_ERROR]*/
#endif                          /* CLIENT */

#ifdef                          SERVER
fsm_invalid,            /* [sent = 0]       [recv = 0]*/
```

```
recv_RRQ,               /* [sent = 0]       [recv = OP_RRQ]                    */
recv_WRQ,               /* [sent = 0]       [recv = OP_WRQ]                    */
fsm_invalid,            /* [sent = 0]       [recv = OP_DATA]                   */
fsm_invalid,            /* [sent = 0]       [recv = OP_ACK]                    */
fsm_invalid,            /* [sent = 0]       [recv = OP_ERROR]                  */
fsm_invalid,            /* [sent = OP_RRQ]  [recv = 0]
fsm_invalid,            /* [sent = OP_RRQ]  [recv = OP_RRQ]                    */
fsm_invalid,            /* [sent = OP_RRQ]  [recv = OP_WRQ]                    */
fsm_invalid,            /* [sent = OP_RRQ]  [recv = OP_DATA]                   */
fsm_invalid,            /* [sent = OP_RRQ]  [recv = OP_ACK]                    */
fsm_invalid,            /* [sent = OP_RRQ]  [recv = OP_ERROR]                  */
fsm_invalid,            /* [sent = OP_WRQ]  [recv = 0]
fsm_invalid,            /* [sent = OP_WRQ]  [recv = OP_RRQ]                    */
fsm_invalid,            /* [sent = OP_WRQ]  [recv = OP_WRQ]                    */
fsm_invalid,            /* [sent = OP_WRQ]  [recv = OP_DATA]                   */
fsm_invalid,            /* [sent = OP_WRQ]  [recv = OP_ACK]                    */
fsm_invalid,            /* [sent = OP_WRQ]  [recv = OP_ERROR]                  */
fsm_invalid,            /* [sent = OP_DATA] [recv = 0]
fsm_invalid,            /* [sent = OP_DATA] [recv = OP_RRQ]                    */
fsm_invalid,            /* [sent = OP_DATA] [recv = OP_WRQ]                    */
fsm_invalid,            /* [sent = OP_DATA] [recv = OP_DATA]                   */
recv_ACK,                                    /* [sent = OP_DATA] [recv = OP_ACK]
fsm_error,              /* [sent = OP_DATA] [recv = OP_ERROR]                  */
fsm_invalid,            /* [sent = OP_ACK]  [recv = 0]
fsm_invalid,            /* [sent = OP_ACK]  [recv = OP_RRQ]                    */
fsm_invalid,            /* [sent = OP_ACK]  [recv = OP_WRQ]                    */
recv_DATA,                                   /* [sent = OP_ACK]  [recv = OP_DATA]
fsm_invalid,            /* [sent = OP_ACK]  [recv = OP_ACK]                    */
fsm_error,              /* [sent = OP_ACK]  [recv = OP_ERROR]                  */
fsm_invalid,            /* [sent = OP_ERROR] [recv = 0]
fsm_invalid,            /* [sent = OP_ERROR] [recv = OP_RRQ]                   */
fsm_invalid,            /* [sent = OP_ERROR] [recv = OP_WRQ]                   */
fsm_invalid,            /* [sent = OP_ERROR] [recv = OP_DATA]                  */
fsm_invalid,            /* [sent = OP_ERROR] [recv = OP_ACK]                   */
fsm_error               /* [sent = OP_ERROR] [recv = OP_ERROR]                 */
#endif                                  /* SERVER */
};

#ifdef                  DATAGRAM
static struct rtt_struct   rttinfo;/* used by the rtt_XXX() functions */
static int              rttfirst = 1;

int                     tout_flag;          /* set to 1 by SIGALRM handler */
#endif                                  /* DATAGRAM */

/*
```

```
 * Main loop of finite state machine.
 *
 * For the client, we're called after either an RRQ or a WRQ has been
 * sent to the other side.
 *
 * For the server, we're called after either an RRQ or a WRQ has been
 * received from the other side.  In this case, the argument will be a
 * 0 (since nothing has been sent) but the state table above handles
 * this.
 */

int                     /* return 0 on normal termination, -1 on timeout */
fsm_loop(opcode)
int             opcode;                                 /* for client: RRQ or WRQ */
                                                                /* for server: 0 */
{
register int            nbytes;

 op_sent = opcode;


#ifdef                                  DATAGRAM

      if (rttfirst) {
                        rtt_init(&rttinfo);
                        rttfirst = 0;
                      }

rtt_newpack(&rttinfo);                                  /* initialize for a new packet */
for ( ; ; ) {
       int          func_timeout();                                 /* our signal handler */
      signal(SIGALRM, func_timeout);
      tout_flag = 0;
      alarm(rtt_start(&rttinfo));               /* calc timeout & start timer */

     if ( (nbytes = net_recv(recvbuff, MAXBUFF)) < 0) {
              if (tout_flag) {
/*
* The receive timed out.  See if we've tried
* enough, and if so, return to caller.
*/

if (rtt_timeout(&rttinfo) < 0) {
#ifdef                      CLIENT
printf("Transfer timed out\n");
#endif
```

```
return(-1);
}
if (traceflag)
 rtt_debug(&rttinfo);
} else
err_dump("net_recv error");
/*
 * Retransmit the last packet.
 */
net_send(sendbuff, sendlen);
continue;
}
alarm(0);                                                    /* stop signal timer */
tout_flag = 0;
rtt_stop(&rttinfo);          /* stop RTT timer, calc new values */
if (traceflag)
   rtt_debug(&rttinfo);
#else        /* else we have a connection-oriented protocol (makes life easier) */
for ( ; ; ) {
     if ( (nbytes = net_recv(recvbuff, MAXBUFF)) < 0)
           err_dump("net_recv error");
#endif                                  /* DATAGRAM */
if (nbytes < 4)
     err_dump("receive length = %d bytes", nbytes);
  op_recv = ldshort(recvbuff);
  if (op_recv < OP_MIN || op_recv > OP_MAX)
          err_dump("invalid opcode received: %d", op_recv);
/*
 * We call the appropriate function, passing the address
 * of the receive buffer and its length.  These arguments
 * ignore the received-opcode, which we've already processed.
 * We assume the called function will send a response to the
 * other side.  It is the called function's responsibility to
 * set op_sent to the op-code that it sends to the other side.
 */

if ((*fsm_ptr[op_sent][op_recv])(recvbuff + 2, nbytes - 2) < 0){
/*
 * When the called function returns -1, this loop
 * is done.  Turn off the signal handler for
 * timeouts and return to the caller.
 */

signal(SIGALRM, SIG_DFL);
return(0);
}
```

```
}
}

#ifdef                                  DATAGRAM
/*
* Signal handler for timeouts.
 * Just set the flag that is looked at above when the net_recv()
 * returns an error (interrupted system call).
 */
int  func_timeout()
{
tout_flag = 1;                                                  /* set flag for function above */
}
#endif                                  /* DATAGRAM */
/*
 * Error packet received and we weren't expecting it.
 */
/*ARGSUSED*/
int  fsm_error(ptr, nbytes)
char                                    *ptr;
int                                     nbytes;
{
                err_dump("error received: op_sent = %d, op_recv = %d",
                op_sent, op_recv);
}
/*
 * Invalid state transition.  Something is wrong.
 */
/*ARGSUSED*/
int  fsm_invalid(ptr, nbytes)
char                                    *ptr;
int                                     nbytes;
{
        err_dump("protocol botch: op_sent = %d, op_recv = %d",
        op_sent, op_recv);
}
```

5j. sendrecv.c

```
/*
 * Send and receive packets.
 */

#include                                "defs.h"
#include                               <sys/stat.h>
```

```
#include                    <ctype.h>

#ifdef                      CLIENT

/*
 * Send a Read-Request or a Write-Request to the other system.
 * These two packets are only sent by the client to the server.
 * This function is called when either the "get" command or the
 * "put" command is executed by the user.
 */

send_RQ(opcode, fname, mode)
int         opcode;                         /* OP_RRQ or OP_WRQ */
char        *fname;
int         mode;
{
register int                    len;
char                           *modestr;
DEBUG2("sending RRQ/WRQ for %s, mode = %d", fname, mode);
stshort(opcode, sendbuff);
strcpy(sendbuff+2, fname);
len = 2 + strlen(fname) + 1;/* +1 for null byte at end of fname */
switch(mode) {
case MODE_ASCII:        modestr = "netascii";                   break;
case MODE_BINARY:         modestr = "octet";                    break;
default:
err_dump("unknown mode");
}
strcpy(sendbuff + len, modestr);
len += strlen(modestr) + 1;         /* +1 for null byte at end of modestr */
sendlen = len;
net_send(sendbuff, sendlen);
op_sent = opcode;
}

/*
 * Error packet received in response to an RRQ or a WRQ.
 * Usually means the file we're asking for on the other system
 * can't be accessed for some reason.  We need to print the
 * error message that's returned.
 * Called by finite state machine.
 */

int         recv_RQERR(ptr, nbytes)
char         *ptr;                          /* points just past received opcode */
int          nbytes;                        /* doesn't include received opcode */
```

```
{
 register int                                    ecode;
ecode = ldshort(ptr);
ptr += 2;
nbytes -= 2;
ptr[nbytes] = 0;                /* assure it's null terminated ... */
DEBUG2("ERROR received, %d bytes, error code %d", nbytes, ecode);
fflush(stdout);
fprintf(stderr, "Error# %d: %s\n", ecode, ptr);
fflush(stderr);
return(-1);                     /* terminate finite state loop */
}
#endif                                  /* CLIENT */
/*
* Send an acknowledgment packet to the other system.
* Called by the recv_DATA() function below and also called by
* recv_WRQ().
*/
send_ACK(blocknum)
int                                     blocknum;
{
DEBUG1("sending ACK for block# %d", blocknum);

stshort(OP_ACK, sendbuff);
stshort(blocknum, sendbuff + 2);
sendlen = 4;
net_send(sendbuff, sendlen);
#ifdef                                  SORCERER
/*
 * If you want to see the Sorcerer's Apprentice syndrome,
 * #define SORCERER, then run this program as the client and
 * get a file from a server that doesn't have the bug fixed
 * (such as the 4.3BSD version).
 * Turn on the trace option, and you'll see the duplicate
 * data packets sent by the broken server, starting with
 * block# 2.  Yet when the transfer is complete, you'll find
 * the file was received correctly.
 */
if (blocknum == 1)
net_send(sendbuff, sendlen);  /* send the first ACK twice */
#endif
op_sent = OP_ACK;
}
/*
* Send data to the other system.
 * The data must be stored in the "sendbuff" by the caller.
```

```
 * Called by the recv_ACK() function below.
 */

send_DATA(blocknum, nbytes)
int         blocknum;
int         nbytes;                                     /* #bytes of actual data to send */
{
        DEBUG2("sending %d bytes of DATA with block# %d", nbytes, blocknum);
        stshort(OP_DATA, sendbuff);
        stshort(blocknum, sendbuff + 2);
      sendlen = nbytes + 4;
      net_send(sendbuff, sendlen);
      op_sent = OP_DATA;
}

/*
 * Data packet received.  Send an acknowledgment.
 * Called by finite state machine.
 * Note that this function is called for both the client and the server.
 */

int     recv_DATA(ptr, nbytes)
register char       *ptr;                               /* points just past received opcode */
register int        nbytes;                             /* doesn't include received opcode */
{
register int                                recvblknum;
recvblknum = ldshort(ptr);
ptr += 2;
nbytes -= 2;
DEBUG2("DATA received, %d bytes, block# %d", nbytes, recvblknum);
if (nbytes > MAXDATA)
err_dump("data packet received with length = %d bytes", nbytes);
if (recvblknum == nextblknum) {
/*
 * The data packet is the expected one.
 * Increment our expected-block# for the next packet.
 */
nextblknum++;
totnbytes += nbytes;
if (nbytes > 0) {
/*
 * Note that the final data packet can have a
 * data length of zero, so we only write the
 * data to the local file if there is data.
 */
```

```
file_write(localfp, ptr, nbytes, modetype);
}
#ifdef                                  SERVER
/*
 * If the length of the data is between 0-511, this is
 * the last data block.  For the server, here's where
 * we have to close the file.  For the client, the
 * "get" command processing will close the file.
 */
if (nbytes < MAXDATA)
file_close(localfp);
#endif

} else if (recvblknum < (nextblknum - 1)) {
/*
 * We've just received data block# N (or earlier, such as N-1,
 * N-2, etc.) from the other end, but we were expecting data
 * block# N+2.  But if we were expecting N+2 it means we've
 * already received N+1, so the other end went backwards from
 * N+1 to N (or earlier).  Something is wrong.
 */
                        err_dump("recvblknum < nextblknum - 1");
                           } else if (recvblknum > nextblknum) {
/*
 * We've just received data block# N (or later, such as N+1,
 * N+2, etc.) from the other end, but we were expecting data
 * block# N-1.  But this implies that the other end has
 * received an ACK for block# N-1 from us.  Something is wrong.
 */
err_dump("recvblknum > nextblknum");
}
/*
 * The only case not handled above is "recvblknum == (nextblknum - 1)".
 * This means the other end never saw our ACK for the last data
 * packet and retransmitted it.  We just ignore the retransmission
 * and send another ACK.
 *
 * Acknowledge the data packet.
 */
send_ACK(recvblknum);
/*
 * If the length of the data is between 0-511, we've just
 * received the final data packet, else there is more to come.
 */
return( (nbytes == MAXDATA) ? 0 : -1 );
}
```

```
/*
* ACK packet received.  Send some more data.
* Called by finite state machine.  Also called by recv_RRQ() to
* start the transmission of a file to the client.
* Note that this function is called for both the client and the server.
*/

int  recv_ACK(ptr, nbytes)
register char        *ptr;                          /* points just past received opcode */
register int         nbytes;                        /* doesn't include received opcode */
{
    register int                        recvblknum;
    recvblknum = ldshort(ptr);
    if (nbytes != 2)
          err_dump("ACK packet received with length = %d bytes",
        nbytes + 2);
       DEBUG1("ACK received, block# %d", recvblknum);
       if (recvblknum == nextblknum) {
 /*
 * The received acknowledgment is for the expected data
 * packet that we sent.
 * Fill the transmit buffer with the next block of data
 * to send.
 * If there's no more data to send, then we might be
 * finished.  Note that we must send a final data packet
 * containing 0-511 bytes of data.  If the length of the
 * last packet that we sent was exactly 512 bytes, then we
 * must send a 0-length data packet.
 */
if ( (nbytes = file_read(localfp, sendbuff + 4,
MAXDATA, modetype)) == 0) {
if (lastsend < MAXDATA)
return(-1);                                 /* done */
/* else we'll send nbytes=0 of data */
}
lastsend = nbytes;
nextblknum++;                                       /* incr for this new packet of data */
totnbytes += nbytes;
send_DATA(nextblknum, nbytes);
return(0);
} else if (recvblknum < (nextblknum - 1)) {
/*
 * We've just received the ACK for block# N (or earlier, such
 * as N-1, N-2, etc) from the other end, but we were expecting
 * the ACK for block# N+2.  But if we're expecting the ACK for
 * N+2 it means we've already received the ACK for N+1, so the
```

```
 * other end went backwards from N+1 to N (or earlier).
 * Something is wrong.
 */
err_dump("recvblknum < nextblknum - 1");
} else if (recvblknum > nextblknum) {
/*
 * We've just received the ACK for block# N (or later, such
 * as N+1, N+2, etc) from the other end, but we were expecting
 * the ACK for block# N-1.  But this implies that the other
 * end has already received data block# N-1 from us.
 * Something is wrong.
 */
err_dump("recvblknum > nextblknum");
} else {
/*
 * Here we have "recvblknum == (nextblknum - 1)".
 * This means we received a duplicate ACK.  This means either:
 * (1) the other side never received our last data packet;
 * (2) the other side's ACK got delayed somehow.
 *
 * If we were to retransmit the last data packet, we would start
 * the "Sorcerer's Apprentice Syndrome."  We'll just ignore this
 * duplicate ACK, returning to the FSM loop, which will initiate
 * another receive.
 */
return(0);
}
/* NOTREACHED */
}
#ifdef                                  SERVER
/*
* RRQ packet received.
* Called by the finite state machine.
* This (and receiving a WRQ) are the only ways the server gets started.
*/
int  recv_RRQ(ptr, nbytes)
char                                    *ptr;
int                                     nbytes;
{
                                        char                                    ackbuff[2];

recv_xRQ(OP_RRQ, ptr, nbytes);/* verify the RRQ packet */
/*
 * Set things up so we can just call recv_ACK() and pretend we
 * received an ACK, so it'll send the first data block to the
 * client.
```

```
 */
lastsend = MAXDATA;
stshort(0, ackbuff);          /* pretend its an ACK of block# 0 */
recv_ACK(ackbuff, 2);         /* this sends data block# 1 */
return(0);              /* the finite state machine takes over from here */
}
/*
 * WRQ packet received.
 * Called by the finite state machine.
 * This (and receiving an RRQ) are the only ways the server gets started.
 */

int    recv_WRQ(ptr, nbytes)
char                                    *ptr;
int                                    nbytes;
{
  recv_xRQ(OP_WRQ, ptr, nbytes);/* verify the WRQ packet */
/*
 * Call send_ACK() to acknowledge block# 0, which will cause
 * the client to send data block# 1.
 */

nextblknum = 1;
send_ACK(0);
return(0);              /* the finite stat machine takes over from here */
}
/*
 * Process an RRQ or WRQ that has been received.
 * Called by the 2 routines above.
 */

int                 recv_xRQ(opcode, ptr, nbytes)
int                opcode;                                  /* OP_RRQ or OP_WRQ */
register char      *ptr;                                /* points just past received opcode */
register int       nbytes;                              /* doesn't include received opcode */
{
register int            i;
register char           *saveptr;
char                    filename[MAXFILENAME], dirname[MAXFILENAME],
mode[MAXFILENAME];
struct stat                             statbuff;
/*
 * Assure the filename and mode are present and
 * null-terminated.
 */
saveptr = ptr;                                          /* points to beginning of filename */
```

```
for (i = 0; i < nbytes; i++)
if (*ptr++ == '\0')
goto FileOK;
err_dump("Invalid filename");

FileOK:
strcpy(filename, saveptr);
saveptr = ptr;                                          /* points to beginning of Mode */
for ( ; i < nbytes; i++)
if (*ptr++ == '\0')
goto ModeOK;
err_dump("Invalid Mode");

ModeOK:
strlccpy(mode, saveptr);    /* copy and convert to lower case */

if (strcmp(mode, "netascii") == 0)
                                    modetype = MODE_ASCII;
else if (strcmp(mode, "octet") == 0)
                                  modetype = MODE_BINARY;
else
                send_ERROR(ERR_BADOP, "Mode isn't netascii or octet");

/*
 * Validate the filename.
 * Note that as a daemon we might be running with root
 * privileges.  Since there are no user-access checks with
 * tftp (as compared to ftp, for example) we will only
 * allow access to files that are publicly accessible.
 *
 * Also, since we're running as a daemon, our home directory
 * is the root, so any filename must have it's full
 * pathname specified (i.e., it must begin with a slash).
 */

if (filename[0] != '/')
                send_ERROR(ERR_ACCESS, "filename must begin with '/'");

                                    if (opcode == OP_RRQ) {
/*
 * Read request - verify that the file exists
 * and that it has world read permission.
 */

if (stat(filename, &statbuff) < 0)
send_ERROR(ERR_ACCESS, sys_err_str());
```

```
if ((statbuff.st_mode & (S_IREAD >> 6)) == 0)
send_ERROR(ERR_ACCESS,
"File doesn't allow world read permission");
} else if (opcode == OP_WRQ) {
/*
* Write request - verify that the directory
 * that the file is being written to has world
 * write permission.  We've already verified above
 * that the filename starts with a '/'.
 */
char       *rindex();
strcpy(dirname, filename);
*(rindex(dirname, '/') + 1) = '\0';
if (stat(dirname, &statbuff) < 0)
send_ERROR(ERR_ACCESS, sys_err_str());
if ((statbuff.st_mode & (S_IWRITE >> 6)) == 0)
send_ERROR(ERR_ACCESS,
 "Directory doesn't allow world write permission");
} else
err_dump("unknown opcode");
localfp = file_open(filename, (opcode == OP_RRQ) ? "r" : "w", 0);
if (localfp == NULL)
send_ERROR(ERR_NOFILE, sys_err_str());                    /* doesn't return */
}
/*
 * Send an error packet.
 * Note that an error packet isn't retransmitted or acknowledged by
 * the other end, so once we're done sending it, we can exit.
 */

send_ERROR(ecode, string)
int          ecode;                                    /* error code, ERR_xxx from defs.h */
char         *string;                  /* some additional info */
/* can't be NULL, set to "" if empty */
{
              DEBUG2("sending ERROR, code = %d, string = %s", ecode, string);
              stshort(OP_ERROR, sendbuff);
              stshort(ecode, sendbuff + 2);
              strcpy(sendbuff + 4, string);
               sendlen = 4 + strlen(sendbuff + 4) + 1;                  /* +1 for null at end */
               net_send(sendbuff, sendlen);
              net_close();
               exit(0);
}

/*
```

```
 * Copy a string and convert it to lower case in the process.
 */

strlccpy(dest, src)
register char                   *dest, *src;
{
register char                   c;

while ( (c = *src++) != '\0') {
if (isupper(c))
c = tolower(c);
*dest++ = c;
}
*dest = 0;
}
#endif  /* SERVER */
```

5k. mainserv.c

```
/*
 * tftp - Trivial File Transfer Program.  Server side.
 *
 *-I                        says we were *not* started by inted.
 *-p port#                  specifies a different port# to listen on.
 *-t                        turns on the traceflag - writes to a file.
 */

#include                                "defs.h"

main(argc, argv)
int   argc;
char  **argv;
{
int                                     childpid;
register char                               *s;

err_init("rich's tftpd");
while (--argc > 0 && (*++argv)[0] == '-')
for (s = argv[0]+1; *s != '\0'; s++)
      switch (*s) {

      case 'i':
         inetdflag = 0;  /* turns OFF the flag */          /* (it defaults to 1) */
         break;
```

```
        case 'p':                                   /* specify server's port# */
         if (--argc <= 0)
         err_quit("-p requires another argument");
         port = atoi(*++argv);
         break;

         case 't':
         traceflag = 1;
         break;

       default:
       err_quit("unknown command line option: %c", *s);
}

if (inetdflag == 0) {
/*
 * Start us up as a daemon process (in the background).
 * Also initialize the network connection - create the socket
 * and bind our well-known address to it.
 */

daemon_start(1);

net_init(TFTP_SERVICE, port);
}
/*
 * If the traceflag is set, open a log file to write to.
 * This is used by the DEBUG macros.  Note that all the
 * err_XXX() functions still get handled by syslog(3).
 */

if (traceflag) {
                    if (freopen(DAEMONLOG, "a", stderr) == NULL)
                       err_sys("can't open %s for writing", DAEMONLOG);
                       DEBUG2("pid = %d, inetdflag = %d", getpid(), inetdflag);
                }

/*
 * Concurrent server loop.
 * The child created by net_open() handles the client's request.
 * The parent waits for another request.  In the inetd case,
 * the parent from net_open() never returns.
 */

for ( ; ; ) {
      if ( (childpid = net_open(inetdflag)) == 0) {
```

```
            fsm_loop(0);          /* child processes client's request */
            net_close();                    /* then we're done */
             exit(0);
               }
/* parent waits for another client's request */
}
/* NOTREACHED */
}
```

5l. initvars.c

```
/*
 * Initialize the external variables.
 * Some link editors (on systems other than UNIX) require this.
 */

#include                "defs.h"

char                    command[MAXTOKEN]       = { 0 };
int                     connected               = 0;
char                    hostname[MAXHOSTNAME]   = { 0 };
int                     inetdflag               = 1;
int                     interactive             = 1;
jmp_buf                 jmp_mainloop            = { 0 };
int                     lastsend                = 0;
fILE                    *localfp                = NULL;
int                     modetype                = MODE_ASCII;
int                     nextblknum              = 0;
int                     op_sent                 = 0;
int                     op_recv                 = 0;
int                     port                    = 0;
char                    *prompt                 = "tftp: ";
char                    recvbuff[MAXBUFF]       = { 0 };
char                    sendbuff[MAXBUFF]       = { 0 };
int                     sendlen                 = 0;
char                    temptoken[MAXTOKEN]     = { 0 };
long                    totnbytes               = 0;
int                     traceflag               = 0;
int                     verboseflag             = 0;
```