```
/*
** Note.h - Some generic external references
*/

extern struct Library   *IntuitionBase,
                        *SockBase;


/*
** Amiga System Includes
*/

#include <exec/types.h>
#include <exec/exec.h>
#include <dos/dos.h>
#include <dos/rdargs.h>
#include <dos/dostags.h>
#include <dos/dosextens.h>
#include <intuition/intuition.h>
#include <utility/tagitem.h>

/*
** Amiga System Prototypes
*/

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/intuition_protos.h>

/*
** socket.library Includes
*/

#include <sys/types.h>
#include <sys/socket.h>
#include <ss/socket.h>
/* make sure you rename <ss/socket_pragmas.sas|manx>
to <ss/socket_pragmas.h> */
#include <ss/socket_pragmas.h>
#include <netdb.h>

/*
** ...and some generic ANSI stuff
*/

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

/*
** The definition of the structure which is the message packet passed
** between the client and server.  This has been kept about as minimal
** as possible, but the buffers had to be designated that way to keep
** the code of the handler routines down in size.
**
** The #define's give the valid types that may be in the nn_Code field
** of the NetNote packet.
*/

struct NetNote
{
    int     nn_Code;
    int     nn_Retval;
    char    nn_Text[200],
            nn_Button[40];
};

#define NN_MSG  0
#define NN_ACK  1
#define NN_ERR  2

/*
** This definition is used in both the client and server as the name any
** entries in the INET:DB/SERVICES file will be under.
*/

#define APPNAME "notes"

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }
#endif

/*
** End of note.h
*/
```

```
;/* shownote.c - Execute to compile with SAS 5.10b
LC -b0 -cfistq -v -y -j73 shownote.c
Blink FROM LIB:c.o shownote.o TO shownote LIBRARY LIB:LC.lib LIB:Amiga.lib
quit
*/


/*
** Our Application Prototypes (specific to noter.c file)
*/
void    main( void );
void    TG_Init( void );
int     SS_Init( void );
int     DoER( char *, char *, char * );
void    AppPanic( char *, int );
void    HandleMsg( int );

/*
** Application-specific defines and globals
*/

char Version[] = "\0$VER: ShowNote 1.2 (1.12.91)";

/*
** The library bases...we need em later...
*/

struct Library *IntuitionBase, *SockBase;

/*
** All other includes and protos are indexed off our catch-all file
** note.h which both the client (sendnote.c) and server (shownote.c) include.
*/

#include    "note.h"


VOID    main( VOID )
{
    int     socket;         /* The socket */

    fd_set  sockmask,       /* Mask of open sockets */
            mask;           /* Return value of socketwait() */

    long    umask;          /* AmigaDOS signal mask */



    /*
    ** Call TG_Init to prepare the generic Amiga stuff for use...
    */

    TG_Init();

    /*
    ** ...and SS_Init for the socket-specific arrangements, keeping
    ** track of what it hands back.
    */

    socket = SS_Init();

    /*
    ** First, prepare the various masks for signal processing
    */

    FD_ZERO( &sockmask );
    FD_SET( socket, &sockmask );

    /*
    ** And we enter the event loop itself
    */

    while(1)
    {
        /*
        ** Reset the mask values for another pass
        */

        mask = sockmask;
        umask = SIGBREAKF_CTRL_C;

        /*
        ** selectwait is a combo network and Amiga Wait() rolled into
        ** a single call.  It allows the app to respond to both Amiga
        ** signals (CTRL-C in this case) and to network events.
        **
        ** Here, if the selectwait event is the SIGBREAK signal, we
        ** bail and AppPanic() but otherwise its a network event.
        ** This is a very crude way of handling the exit, but it
        ** is an effective one
        */

        if (selectwait( 2, &mask, NULL, NULL, NULL, &umask ) == -1 )
        {
            AppPanic("CTRL-C:\nProgram terminating!",0);
        }

        /*
        ** Since the contact between the client and server is so
        ** quick, an iterative server is adeqaute.  For cases where
        ** extended connections or concurrent connections are needed,
        ** either a state-machine or concurrent server would be a
        ** better choice.
        **/

        /*
        ** Here, we accept the pending connection (the only case
        ** possible with this mechanism) and dispatch to a routine
        ** which actually handles the client-server communication.
        */

        if (FD_ISSET( socket, &mask ))
        {
            HandleMsg( socket );
        }
        else
        {
            AppPanic("Network Signal Error!",0);
        }
    }
}


/*
**  AppPanic() - General Shutdown Routine
**
**  This routine serves to provide both a nice GUI way of indicating error
**  conditions to the user, and to handle all the aspects of shutting the
**  server down.  In a real-world application, you would probably separate
**  the error condition shutdown from the normal shutdown.  Since this is
**  an example, it would add needless complexity to the code.  Further,
**  as far as this server is concerned, shutting down _is_ an error state,
**  since SIGBREAKF_CTRL_C is a process-specific warning of shutdown.
*/
```

```
VOID    AppPanic( char *panictxt, int panicnum )
{
    char buffer[255];

    if (!panicnum)
    {
        DoER( APPNAME, panictxt, "OK" );
    }
    else
    {
        sprintf( (char *)&buffer, "%s\n\n%s", panictxt, strerror(panicnum));
        DoER( APPNAME, (char *)&buffer, "OK" );
    }
    if (SockBase)
    {
        cleanup_sockets();
        CloseLibrary(SockBase);
    }

    if (IntuitionBase)
    {
        CloseLibrary(IntuitionBase);
    }

    exit(RETURN_ERROR);

}

/*
**  DoER() - Attempt at a "generic" wrapper for EasyRequest()
**
**  Since EasyRequest(), though "easy", still requires some initialization
**  before it can be used, this routine acts as a wrapper to EasyRequest.
**  It also catches and provides a means to implement application-generic
**  values for what gets handed to the EasyRequest routine, making coding
**  just a wee bit easier.
*/
int     DoER( char *titletxt, char *msgtxt, char *btntxt )
{
    struct EasyStruct easy =    {
                                sizeof(struct EasyStruct),
                                NULL,
                                NULL,
                                NULL,
                                NULL
                                };

    int retval = 0;

    if (IntuitionBase)
    {
        if (titletxt)
        {
            easy.es_Title = titletxt;
        }
        else
        {
            easy.es_Title = APPNAME;
        }

        if (msgtxt)
        {
            easy.es_TextFormat = msgtxt;
        }
        else
        {
            easy.es_TextFormat = "Null message text!\nIsnt it?";
        }
        if (btntxt)
        {
            easy.es_GadgetFormat = btntxt;
        }
        else
        {
            easy.es_GadgetFormat = "Take off, eh!";
        }

        retval = EasyRequest( NULL, &easy, NULL, NULL );
        return (retval);
    }
}

/*
**  TG_Init() - Initializer of AOS/Intuition
**
**  This routine just handles opening Intuition, but would be a good
**  place to put any other initialization code which is specific to getting
**  an application's Amiga environment properly set up.
*/
VOID    TG_Init( VOID )
{
    IntuitionBase = OpenLibrary("intuition.library",36);
    if (!IntuitionBase)
        exit(RETURN_ERROR);
}
/*
**  SS_Init() - Initializer of shared socket library
**
**  SS_Init() handles the opening of socket.library, the formation of an
**  application-specific socket environment, and the creation of the initial
**  socket for the server.  It returns an identifier to the socket it has
**  prepared, which just happens to represent itself as an int.
*/
int     SS_Init( VOID )
{
    struct sockaddr_in  sockaddr;

    int snum, len = sizeof(sockaddr);

    /*
    ** Attempt to open socket library and initialize socket environ.
    ** If this fails, bail out to the non-returning AppPanic() routine.
    */

    /*
    ** The errno variable is a part of ANSI, and is defined in the c.o
    ** startup code.  Essentially, its where ANSI functions put their
    ** error codes when they fail.  For more information, consult a
    ** reference to ANSI C.
    */

    if (SockBase = OpenLibrary("inet:libs/socket.library",0L))
    {
        setup_sockets( 3, &errno );
    }
    else
    {
        AppPanic("Can't open socket.library!",0);
    }

    /*
    ** Open the initial socket on which incoming messages will queue for
    ** handling.  While the server is iterative, I do it this way so that
    ** SIGBREAKF_CTRL_C will continue to function.
    */
```

```
    if ((snum = socket( AF_INET, SOCK_STREAM, 0 )) == -1)
    {
        AppPanic("Socket Creation:",errno);
    }

    /*
    ** Here we clear and prepare the information to give our socket
    ** a real address on the system.
    */

    memset( &sockaddr, 0, len );
    sockaddr.sin_family = AF_INET;

      /*
      **  Following is commented out for ease of testing purposes!
      **
      **  {
      **      struct servent *servptr;
      **      char *serv = APPNAME;
      **
      **      if ((servptr = getservbyname( serv, "tcp" )) == NULL)
      **      {
      **          AppPanic("Service not in inet:db/services list!",0);
      **      }
      **      sockaddr.sin_port = servptr->s_port;
      **  }
      */

    sockaddr.sin_port = 8769;

    sockaddr.sin_addr.s_addr = INADDR_ANY;

    /*
    ** Having everything set up, we now attempt to allocate the port number
    ** for our socket.  If this fails, we bail.
    */

    if ( bind( snum, (struct sockaddr *)&sockaddr, len ) < 0 )
    {
        AppPanic("Socket Binding:",errno);
    }

    /*
    ** Okay, the socket is as ready as it gets.  Now all we need to do is to
    ** tell the system that the socket is open for business.  In an ideal
    ** world, this needs to be checked for errors, but for the scope of the
    ** example, it isnt necessary.  By the way, the '5' in the listen() call
    ** indicates the "queue size" for number of outstanding requests.
    */

    listen( snum, 5 );

    /*
    ** And last, we pass the socket number back to the main routine.
    */

    return snum;
}

/*
**  HandleMsg() - Handles client connection and message display
**
**  This is where 90% of the "function" of the program occurs.  This routine
**  connects the server to the client socket, gets the incoming message pkt,
**  acknowledges it, displays it, then terminates the client connection.
**  For doing all that, its small, a testament to how easily the actual work
**  can be done.
*/
void    HandleMsg( int sock )
```

```
{
    struct NetNote in;              /* Buffer for incoming packets */

    struct sockaddr_in saddr;       /* Socket address from accept() */
    struct in_addr sad;             /* Internet address component */

    struct hostent *hent;           /* Internet host information */

    int nsock,                      /* New socket from accept() */
        len,                        /* Length of addr from accept() */
        retv;                       /* Return value from DoER call */

    char rname[80],                 /* Buffer for titlebar string */
         *hname,                    /* Ptr to the hostname */
         *dd_addr;                  /* Ptr to the dotted-decimal address */

    /*
    ** We accept() the attempted connection on socket 'sock'
    ** which also yields the addr of the remote machine.  Then we
    ** attempt to convert the name to something meaningful.
    **
    ** First, we clear the stuff...
    */


    bzero( (char *)&rname, 80);
    bzero( (char *)&saddr, sizeof(struct sockaddr_in) );
    bzero( (char *)&sad, sizeof(struct in_addr) );
    len = sizeof(struct sockaddr_in);


    /*
    ** Then we accept the connection on the socket
    */

    if (!(nsock = accept( sock, (struct sockaddr *)&saddr, &len )))
    {
        AppPanic("Accept:",errno);
    }

    /*
    ** Break the internet address out of the sockaddr_in structure and then
    ** create a dotted-decimal format string from it, for later use
    */

    sad = saddr.sin_addr;
    dd_addr = inet_ntoa(sad.s_addr);

    /*
    ** Use the internet address to find out the machine's name
    */

    if ( !(hent =
            gethostbyaddr( (char *)&sad.s_addr,
                           sizeof(struct in_addr),
                           AF_INET )))
    {
        AppPanic("Client resolution:\nAddress not in hosts db!", 0 );
    }
    hname = hent->h_name;

    /*
    ** Form the string which goes into the title bar using name & address
    */

    sprintf( rname, "FROM: %s (%s)", hname, dd_addr );
```

```
    /*
    ** Okay, now the waiting packet needs to be removed from the connected
    ** socket that accept() gave back to us.  Verify its of type NN_MSG and
    ** if not, set return type to NN_ERR.  If it is, then display it and
    ** return an NN_ACK message.
    */

    recv( nsock, (char *)&in, sizeof(struct NetNote), 0 );
    if (in.nn_Code == NN_MSG)
    {
        DisplayBeep(NULL);
        DisplayBeep(NULL);
        retv = DoER( rname, (char *)&in.nn_Text, (char *)&in.nn_Button );
        in.nn_Code = NN_ACK;
        in.nn_Retval = retv;
    }
    else
    {
        in.nn_Code = NN_ERR;
    }

    /*
    ** Having dealt with the message one way or the other, send the message
    ** back at the remote, then disconnect from the remote and return.
    */

    send( nsock, (char *)&in, sizeof(struct NetNote), 0 );
    s_close( nsock );
```

```
;/* sendnote.c - Execute to compile with SAS 5.10b
LC -b0 -cfistq -v -y -j73 sendnote.c
Blink FROM LIB:c.o sendnote.o TO sendnote LIBRARY LIB:LC.lib LIB:Amiga.lib
quit
*/


/*
** Our Application Prototypes (specific to notes.c file)
*/

void    main( int, char ** );
void    FinalExit( int );

/*
** Application-specific defines and globals
*/

char Version[] = "\0$VER: SendNote 1.2 (1.12.91)";

#define TEMPLATE    "Host/A,Text,Button"
#define OPT_HOST    0
#define OPT_TEXT    1
#define OPT_BUTTON  2
#define OPT_COUNT   3

/*
** The library bases...we need em later...
*/

struct Library *IntuitionBase, *SockBase;

/*
** All other includes and protos are indexed off our catch-all file
** note.h which both the client (sendnote.c) and server (shownote.c)
** include.
*/

#include    "note.h"


void    main(int argc, char **argv)
{
    struct RDArgs *rdargs;      /* ReadArgs() return information */

    struct sockaddr_in serv;    /* Server’s Internet Address */

    struct hostent *host;       /* The located host info */

    struct NetNote out;         /* Message packet for send/recv */

    long opts[OPT_COUNT] =  {
                            0L,
                            (long)"== PING! ==",
                            (long)"OK"
                            };

    int sock;                   /* The working socket */

    char *hostnam,                  /* Arg of hostname */
         *text,                 /* Arg of text to be sent */
         *button;               /* Arg of button text */

    /*
    ** Process arguments using new (2.0) dos calls.
    */

    rdargs = (struct RDArgs *)ReadArgs( (UBYTE *)TEMPLATE, opts, NULL );
    if(rdargs == NULL)
```

```
    {
        puts("Command line not accepted!");
        FinalExit( RETURN_ERROR );
    }

    hostnam = (char *)opts[OPT_HOST];
    text = (char *)opts[OPT_TEXT];
    button  = (char *)opts[OPT_BUTTON];

    /*
    **  Open socket.library and intialize socket space
    */

    if (SockBase = OpenLibrary("inet:libs/socket.library", 0L))
    {
        setup_sockets( 1, &errno );
    }
    else
    {
        puts("Can't open socket.library!");
        FinalExit( RETURN_ERROR );
    }

    /*
    ** First we need to try and resolve the host machine as a
    ** normal IP/Internet address.  If that fails, fall back to seaching
    ** the hosts file for it.  Before anything, we need to clear out
    ** the buffer (serv) where the information will be placed, using
    ** the bzero() call (actually a macro in sys/types.h).
    */

    bzero( &serv, sizeof(struct sockaddr_in) );
    if ( (serv.sin_addr.s_addr = inet_addr(hostnam)) == INADDR_NONE )
    {
        /*
        ** Okay, the program wasnt handed a dotted decimal address,
        ** so we check and see if it was handed a machine name.
        **
        ** NOTE:  Grab the information you need before you use the
        **        gethostbyname() call again.  Subsequent calls
        **        will overwrite the buffer it hands back.
        */

        if ( (host = gethostbyname(hostnam)) == NULL )
        {
            printf("Host not found: %s\n",host);
            FinalExit( RETURN_ERROR );
        }

        /*
        ** It does indeed have a name, so copy the addr field from the
        ** hostent structure into the sockaddr structure.
        */

        bcopy( host->h_addr, (char *)&serv.sin_addr, host->h_length );
    }

/*
**  Following is commented out for ease of testing purposes!  Normally, apps
**  should obtain server numbers using this type of code and a matching entry
**  in the inet:db/services file.
**
**  {
**      struct servent *servptr;
**      char *servnam = APPNAME;
**
**      *
**      ** Open the INET:DB/SERVICES file and locate the server info
**      ** by matching the name and service.
**      *
**
**      if ((servptr = getservbyname( servnam, "tcp" )) == NULL)
**      {
**          printf("%s not in inet:db/services list!",servnam);
**          FinalExit( RETURN_ERROR );
**      }
**      serv.sin_port = servptr->s_port;
**  }
**
*/


    /*
    ** If you used the above code, you would remove this line:
    */

    serv.sin_port = 8769;

    /*
    ** This tells the system the socket in question is an Internet socket
    */

    serv.sin_family = AF_INET;

    /*
    ** Initialize the socket
    */

    if ( (sock = socket( AF_INET, SOCK_STREAM, 0 )) < 0 )
    {
        printf("socket gen: %s\n", strerror(errno));
        FinalExit( RETURN_ERROR );
    }

    /*
    ** Connect the socket to the remote socket, which belongs to the
    ** server, and which will "wake up" the server.
    */

    if ( connect( sock,
                  (struct sockaddr *)&serv,
                  sizeof(struct sockaddr) ) < 0 )
    {
        printf("connect: %s\n", strerror(errno));
        s_close( sock );
        FinalExit( RETURN_ERROR );
    }

    /*
    ** Compose the message packet for transmission
    */

    out.nn_Code = NN_MSG;
    strcpy( (char *)&out.nn_Text, text );
    strcpy( (char *)&out.nn_Button, button );

    /*
    ** Send the packet to the remote system
    */

    send( sock, (char *)&out, sizeof(struct NetNote), 0 );

    printf("\nMessage sent to %s...waiting for answer...\n", hostnam );

    /*
    ** Wait for either acknowledge or error.  This is a potential hang
    ** location if the server is mortally wounded.
    */
```

```
    recv( sock, (char *)&out, sizeof(struct NetNote), 0 );

    /*
    ** Evaluate the packet returned to us
    */

    if (out.nn_Code == NN_ACK)
    {
        printf("Response:  Button %ld pressed.\n\n", out.nn_Retval );
    }
    else
    {
        puts("Error during message send...please try again later!");
        FinalExit( RETURN_ERROR );
    }

    /*
    ** Since ReadArgs() was called inside the main() function, the pointer
    ** to the buffer it created needs to be deallocated inside main().
    */

    if (rdargs)
    {
        FreeArgs( rdargs );
    }

    FinalExit( RETURN_OK );

}

/*
**  FinalExit() - Non-returning routine which handles exits and cleanups.
**
**
*/

void    FinalExit( int retcode )
{

      /*
      ** If SockBase is non-null, it means that socket.library was opened,
      ** and a socket environment was initialized.  Remove the environment
      ** and close the library.
      */

    if (SockBase)
    {
        cleanup_sockets();
        CloseLibrary(SockBase);
    }

      /*
      ** Terminate program, handing return code out to the shell handler
      */

    exit( retcode );
}
```