7188E/843X/844X/883X/884X

Networking Library User's Manual

Table of Contents

1. Introduction

1.1 Package List & Release Notes

Package List

In addition to this manual, all packages includes the following items:

- One 7188E hardware module
- One user's manual (this manual)
- One release note
- One software utility disk or CD
- One download cable, CA0915

Note: If any of these items are missing or damaged, contact the local distributors for more information. Save the shipping materials and cartons in case you want to ship in the future.

Release Note

It is recommended to read the **release note & README.TXT** first. The release note is given in the shipping. The README.TXT is given in the CD\README.TXT. Some important information are given in the release note & CD\README.TXT

Order Information

Call distributor for details.

1.2 Why! Ethernet Solutions

"Embedded Internet" and "Embedded Ethernet" are hot topics today. Nowadays, the Ethernet protocol has become the de-facto standard for local area networks. Via the Internet, connectivity is spreading into many different areas, from home appliances to vending machines, from testing equipment to UPS...etc. Many embedded designers now face dilemma of adding an Ethernet interface to their products, either for use with local networks or for connecting to the Internet. Solutions to this problem include both hardware and software. Connecting via Ethernet requires a software protocol called TCP/IP. The installed base of Ethernet networks is vast and growing quickly. Most office buildings, factories, and new homes have an installed Ethernet network. With Ethernet, the network is always available. Using Ethernet for network in industrial areas is appealing because the required cabling is already installed.

The 7188E series are a series of embedded controllers designed to meet the most common requirements of Internet/Ethernet applications. They can be used to replace the PC or PLC for the harsh environment.

The 7188E series provide one on-board 10BASE-T port that is directly driven by a NE2000 compatible Ethernet controller. The 10BASE-T port is equipped with a RJ-45 connector. The 10BASE-T interface supports the max. cable length (100 meters) between 7188E & a hub. To link the 7188E & the other device through a 10BASE_T hub, simply use two straight-through cables: one cable connects to 7188E; the other cable connects the hub to the other device.

1.3 The 7188EX, 7188EA & 7188EN Series

The I-7188EX is powered by a 80188-40 processor with 512K bytes of static RAM, and 512K bytes of Flash memory. One serial RS-232 port and one RS-485 port are provided. The Ethernet support is provided by a NE-2000 compatible controller with 16K bytes of on-chip buffer memory and 10Base-T media interface. The I-7188EX also provides 14 user defined I/O lines. A cost-effective I/O expansion board with A/D, D/A, relays drivers and protected inputs are available. The I-7188EX also supports a battery backup SRAM board and Flash-Rom board, providing non-volatile mass storage from 128K bytes megabytes to 64 megabytes. The 10BASE-T port is equipped with a RJ-45 connector. The 10BASE-T interface supports a max. of 100 meters of Cable length between the I-7188EX and the network hub.

Compared to the I-7188EX, the I-7188EA adds seven open-collector output channels and six digital Input channels. The I/O Expansion bus has been occupied by a D/I/O expansion board.

The I-7188EX, Embedded Internet/Ethernet Controller, focuses on embedded control applications while the I-7188EN, Internet Communication Controller, focuses on communication applications and applications dependant on different embedded firmware programs. The Internet Communication Controller can be used as Device Server, Addressable Ethernet to RS-232/485/422 Converter, or Embedded Internet/Ethernet Controller. The user should refer to the comparison table to choose the product that best suit their needs. Currently, we offer a wide range of Internet Communication Controllers, such as the I-7188E1/E2/E3/E4/E5/E8. Except the RTC circuitry, the basic hardware of the I-7188EN is similar to the I-7188EX. Since there are too many Configurations for the I-7188EN series product, an OEM or ODM version is welcomed.

1.4 TCP/IP 4-layer model

S

4 Application Layer	TELNET, FTP, SMTP, DNS
3 Transport Layer	TCP , UDP
2 Network Layer	IP, ICMP, ARP, RARP
1 Physical Layer	Packet radio, Ethernet

Figure 1: TCP/IP protocol suite using a 4-layer model

1.5 Internet Address

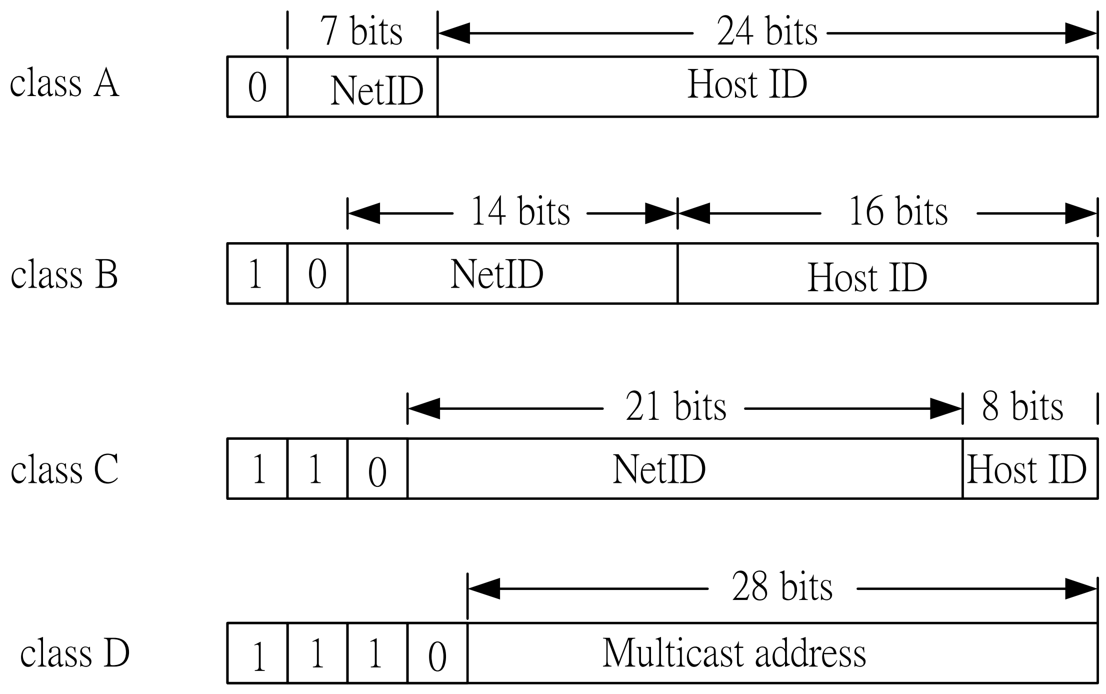


Figure 2: Internet Address.

- **Network Mask:**
  Class A: 255.0.0.0
  Class B: 255.255.0.0
  Class C: 255.255.255.0
  Class D: Multicast address

1.6 Connection-Oriented Protocol

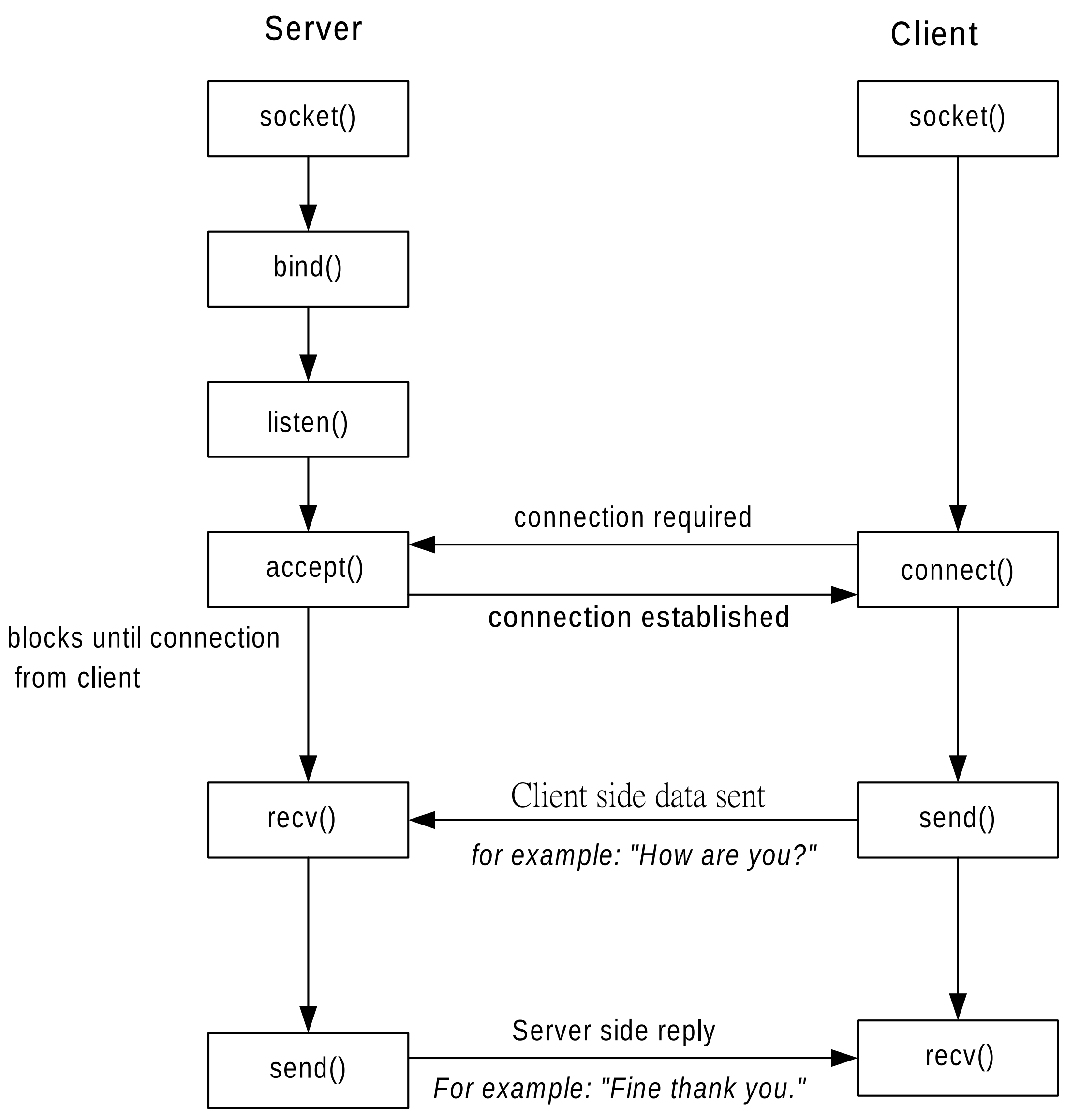


Figure 3: Connection-Oriented Protocol **(TCP)**

1.7 Connectionless Protocol

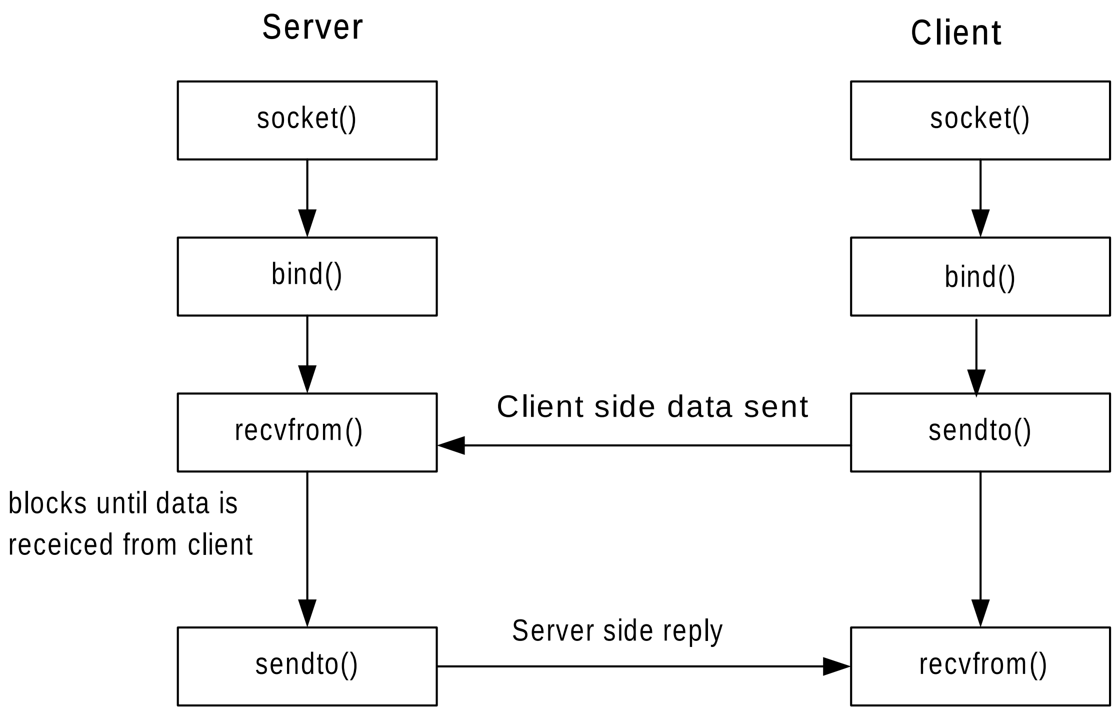


Figure 4: Connectionless Protocol **(UDP)**.

2. Software Installation

- ***For the 7188E/843X/883X/844X/884X series***
  The software for the *7188E/843X/883X/844X/884X* consists of one CD-ROM.

  Directories of CD-ROM: Napdos\MiniOS7\tcpip_lib
  For example:

  ```
  D:\8000\Napdos\MiniOS7\tcpip_lib>xcopy/e/s   *.*   c:\tcpip\.
  ```

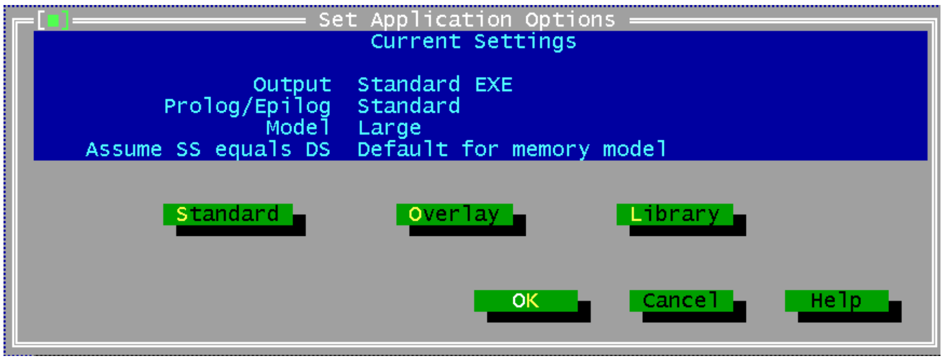
3. Compiling & linking

User must use C Language when writing programs. You can use TC++, BC++, MSC or MSVC++（before 1.52）. Please take care of the following items:

- ➢ Generate a standard DOS executable program.
- ➢ Select CPU=80188/80186.
- ➢ Select EMULATION if floating point computation is required.(can not select 8087)

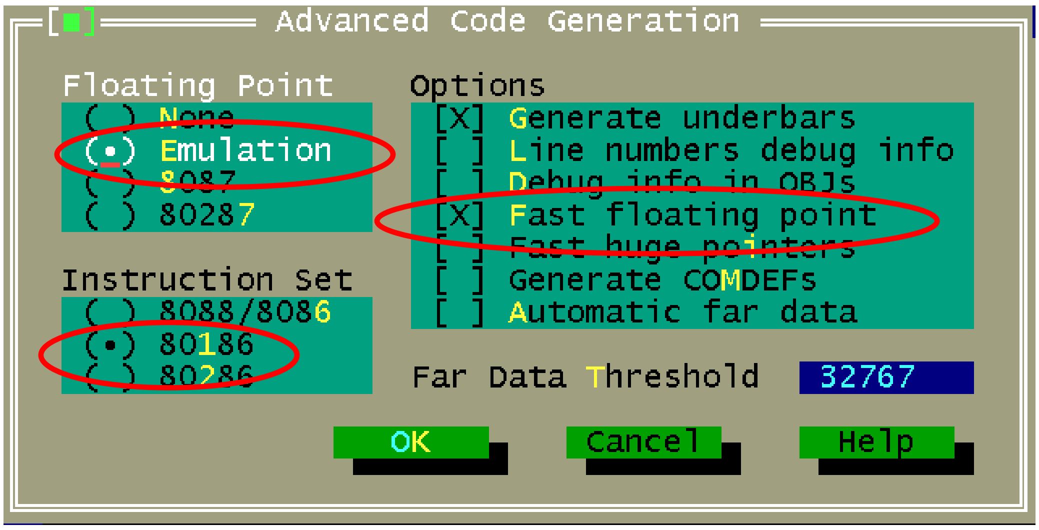
3.1 Using TC++

- Select standard code:
  Options → Application…→



```
Set Application Options
Current Settings

           Output  Standard EXE
     Prolog/Epilog  Standard
             Model  Large
Assume SS equals DS  Default for memory model
```

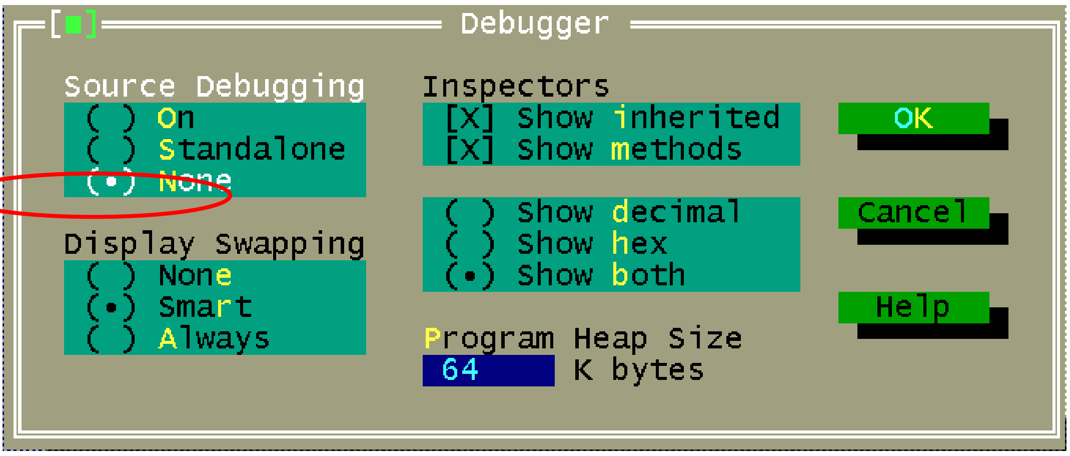
- Select EMULATION , 80186 , Debug info in OBJs cancel:
  Option → Compiler → Advanced Code Generation…→



```
Advanced Code Generation

Floating Point          Options
 ( ) None                [X] Generate underbars
 (•) Emulation           [ ] Line numbers debug info
 ( ) 8087                [ ] Debug info in OBJs
 ( ) 80287               [X] Fast floating point
                         [ ] Fast huge pointers
Instruction Set          [ ] Generate COMDEFs
 ( ) 8088/8086           [ ] Automatic far data
 (•) 80186
 ( ) 80286               Far Data Threshold  32767
```

- Select None Source Debugging

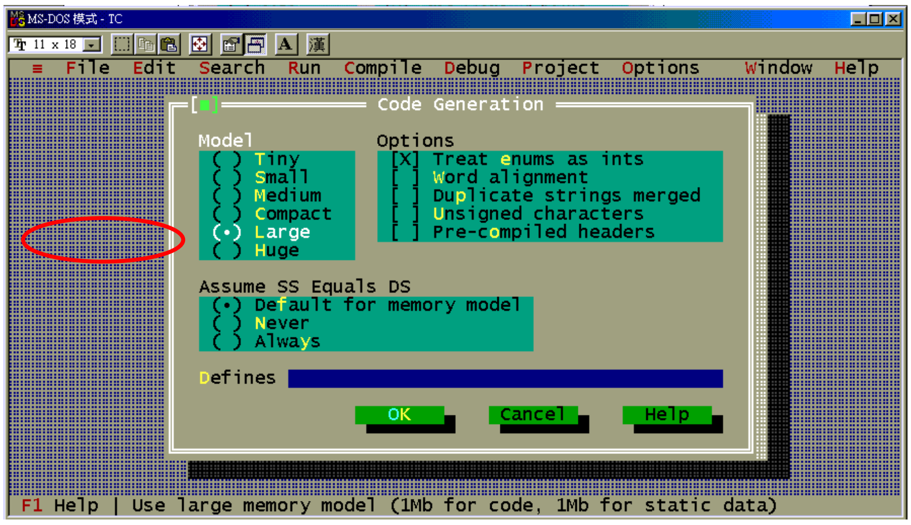
Option → Debugger →



```
[■] Debugger

Source Debugging        Inspectors
 ( ) On                  [X] Show inherited       OK
 ( ) Standalone          [X] Show methods
 (•) None                                          Cancel
                         ( ) Show decimal
Display Swapping         ( ) Show hex
 ( ) None                (•) Show both             Help
 (•) Smart
 ( ) Always             Program Heap Size
                         64        K bytes
```

- Select large mode:

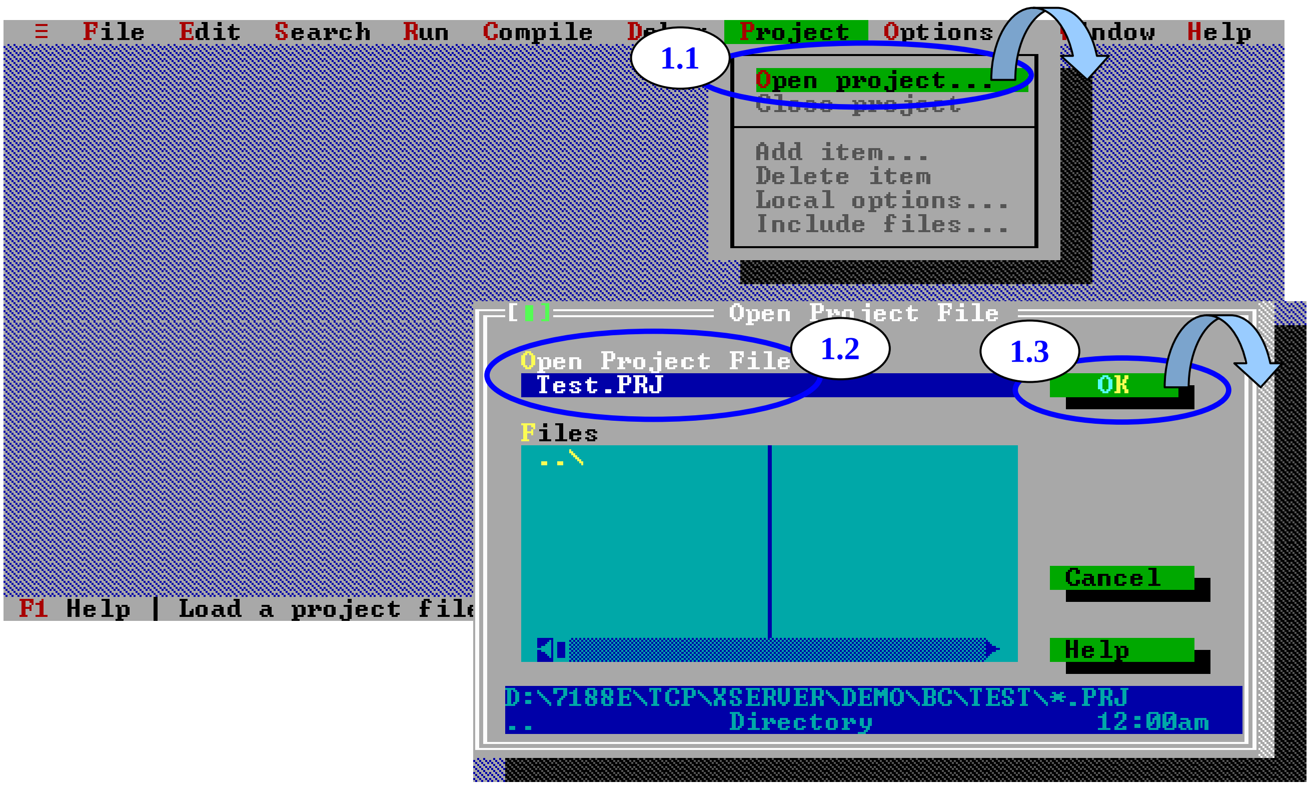
Options → Compiler → Code generation…→



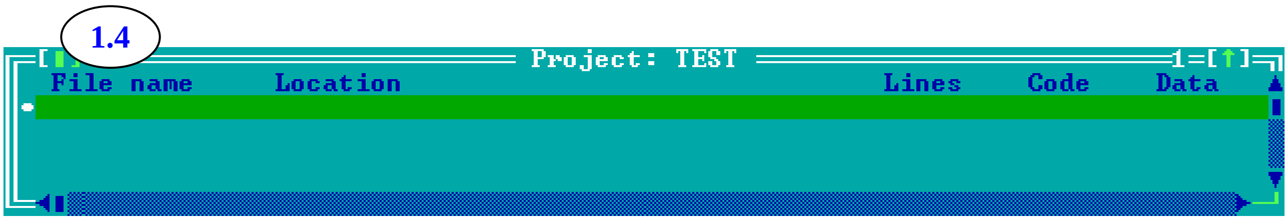
```
MS-DOS 模式 - TC
≡  File  Edit  Search  Run  Compile  Debug  Project  Options   Window  Help

[■] Code Generation

Model                  Options
 ( ) Tiny               [X] Treat enums as ints
 ( ) Small              [ ] Word alignment
 ( ) Medium             [ ] Duplicate strings merged
 ( ) Compact            [ ] Unsigned characters
 (•) Large              [ ] Pre-compiled headers
 ( ) Huge

Assume SS Equals DS
 (•) Default for memory model
 ( ) Never
 ( ) Always

Defines

              OK       Cancel      Help

F1 Help | Use large memory model (1Mb for code, 1Mb for static data)
```

3.2 Using BC ++ 3.1

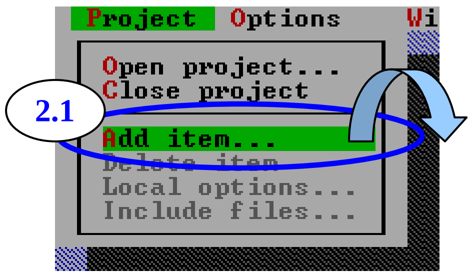
Step 1: Create a new project.

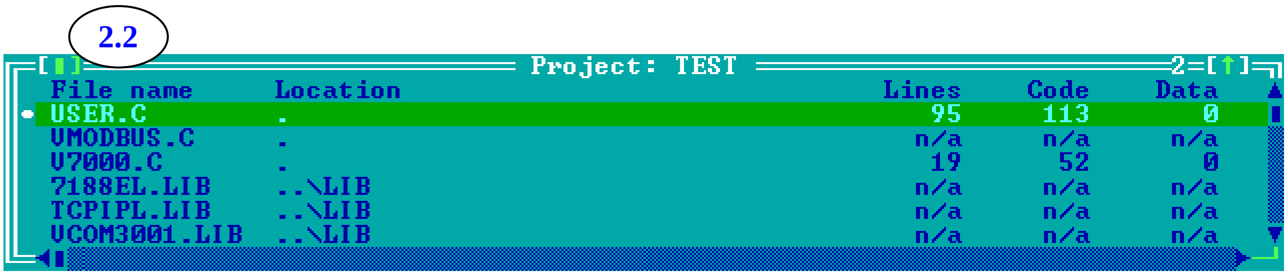


```
F1 Help | Load a project file
D:\7188E\TCP\XSERVER\DEMO\BC\TEST\*.PRJ
..              Directory              12:00am
```



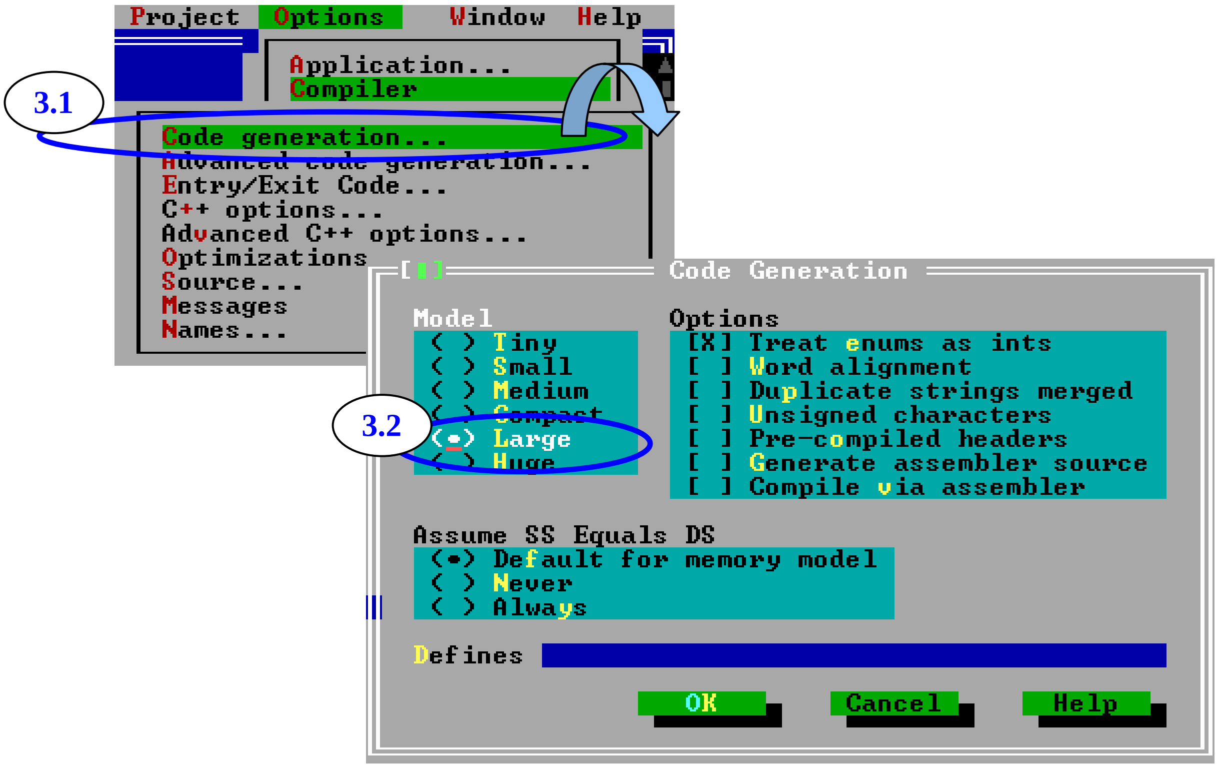
Step 2: Add all necessary files into the project.





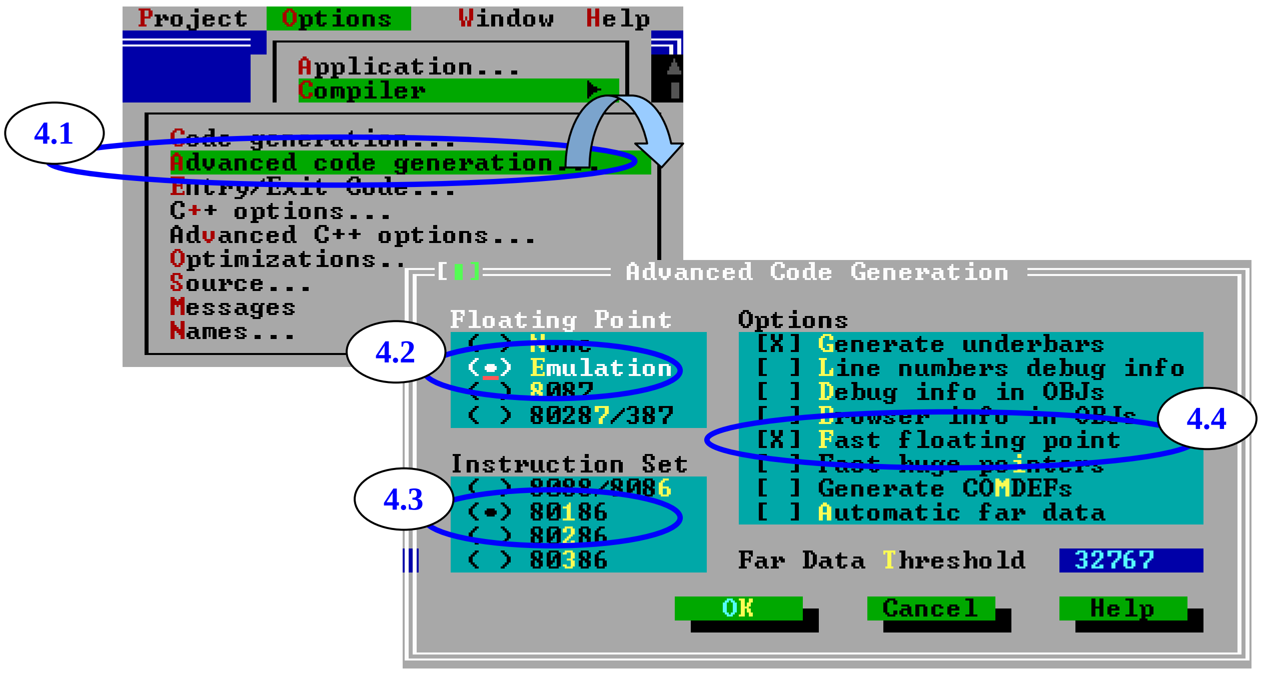
```
Project: TEST                                          2
  File name     Location                    Lines    Code    Data
• USER.C        .                              95     113       0
  VMODBUS.C     .                             n/a     n/a     n/a
  V7000.C       .                              19      52       0
  7188EL.LIB    ..\LIB                        n/a     n/a     n/a
  TCPIPL.LIB    ..\LIB                        n/a     n/a     n/a
  VCOM3001.LIB  ..\LIB                        n/a     n/a     n/a
```

Step 3: Set Code generation options.



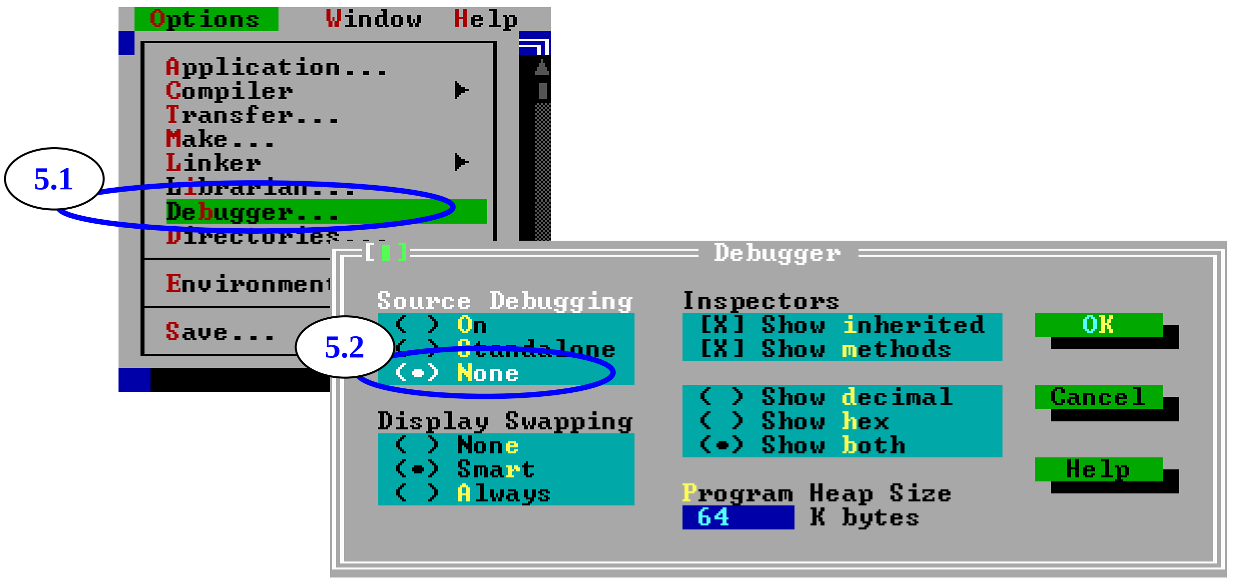
```
Code Generation
Model                Options
( ) Tiny             [X] Treat enums as ints
( ) Small            [ ] Word alignment
( ) Medium           [ ] Duplicate strings merged
( ) Compact          [ ] Unsigned characters
(•) Large            [ ] Pre-compiled headers
( ) Huge             [ ] Generate assembler source
                     [ ] Compile via assembler

Assume SS Equals DS
(•) Default for memory model
( ) Never
( ) Always

Defines
```

Step 4: Set Advanced code generation options.



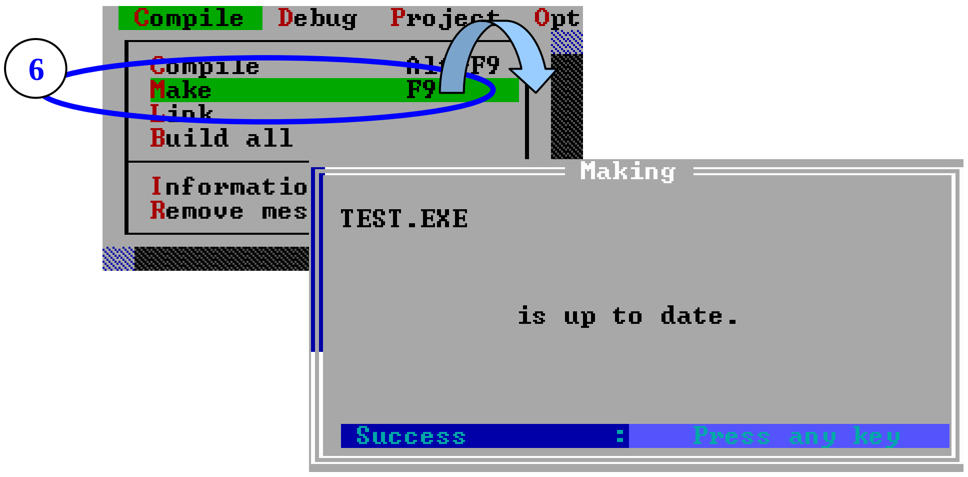
```
Project  Options    Window  Help
          Application...
          Compiler           ►
        Code generation...
        Advanced code generation...
        Entry/Exit Code...
        C++ options...
        Advanced C++ options...
        Optimizations...
        Source...
        Messages...
        Names...

[■]═══════ Advanced Code Generation ═══════
 Floating Point          Options
 ( ) None                 [X] Generate underbars
 (•) Emulation            [ ] Line numbers debug info
 ( ) 8087                 [ ] Debug info in OBJs
 ( ) 80287/387            [ ] Browser info in OBJs
                          [X] Fast floating point
 Instruction Set          [ ] Fast huge pointers
 ( ) 8088/8086            [ ] Generate COMDEFs
 (•) 80186                [ ] Automatic far data
 ( ) 80286
 ( ) 80386                Far Data Threshold  32767

            OK        Cancel        Help
```

Step 5: Set Debugger options.



```
 Options     Window  Help
  Application...
  Compiler           ►
  Transfer...
  Make...
  Linker             ►
  Librarian...
  Debugger...
  Directories...
  Environment
  Save...

[■]══════════════ Debugger ══════════════
 Source Debugging       Inspectors
 ( ) On                  [X] Show inherited        OK
 ( ) Standalone          [X] Show methods
 (•) None                                          Cancel
                         ( ) Show decimal
 Display Swapping        ( ) Show hex
 ( ) None                (•) Show both             Help
 (•) Smart
 ( ) Always              Program Heap Size
                         64     K bytes
```

Step 6: Make the project.



```
TEST.EXE

          is up to date.

Success    :    Press any key
```

4. BSD Socket Interface

4.1 About BSD Sockets

The BSD 4.3 sockets are the closest thing there is to a standard user interface to TCP/IP. However, they can only be approximated on a non-UNIX system, because many UNIX functions interact with sockets.

Writing New Code

For somebody who already knows the BSD sockets, writing any new code using them makes sense. (The Dynamic Protocol Interface (Refer to Chapter 5) needs quite a bit less space, but the difference in speed is not significant.) To support these users, we have made the sockets as similar to 4.3 BSD sockets as reasonably possible. These points may require special attention:

- Symbolic error codes are not perfectly standardized across different UNIX systems. ICPDAS uses the Solaris names.
- The typical UNIX use of errno is not reentrant. If this becomes critical, use ***getsockopt()*** to get the last error code.
- The function ***gethostbyname()*** is not reentrant. Use ***gethostbyname_r()*** instead if this is critical.
- You can't mix files and sockets. For instance, you can't use a ***selectsocket()*** to wait for either a keyboard character or a network packet.
- Avoid non-blocking mode if multitasking is used.

Structures and Definitions

Many of the BSD socket routines use a pointer to structure sockaddr, which specifies network address information. The sockaddr structure is a generic structure that can be used with a number of different communications protocols. ICPDAS only uses the Internet Protocol (IP), and therefore only requires the use of the Internet structure sockaddr_in. Values are assigned to sockaddr_in and passed into the socket routine via the sockaddr parameter. This requires a typecast to sockaddr *. The discussion of the ***connect()*** function provides an example. Here are the structure definitions:

```
struct sockaddr { /* generic socket address */
```

unsigned short sa_family; / address family */*
char sa_data[14]; / up to 14 bytes of address */*
};
In practice, this is used almost as a void pointer. The true Internet address structure is:
struct in_addr { / Internet address */*
unsigned long S_addr;
};
struct sockaddr_in { / Internet socket address */*
short sin_family;
unsigned short sin_port;
struct in_addr sin_addr;
char sin_zero[8];
};

Please refer to tcpip.h for more details. (ICPDAS CD Napdos\MiniOS7\tcpip_lib\)

BSD Socket Interface Functions

The BSD Socket Interface provides these function calls:

- ***accept()*** accepts a connection on a socket.
- ***bind()*** binds a name to a socket.
- ***closesocket()*** closes a socket.
- ***connect()*** initiates a connection on a socket.
- ***fcntlsocket( )*** controls socket flags.
- ***gethostbyname()*** returns the IP address that corresponds to a host name.
- ***getpeername()*** extracts the remote address information for a socket.
- ***getsockname()*** extracts the local address information for a socket.
- ***getsockopt()*** gets options on sockets.
- ***ioctlsocket()*** sets control parameters for a socket.
- ***listen()***listens for connections.
- ***readsocket()*** receives a message from a socket ID.
- ***recv()*** receives a message.
- ***recvfrom()*** receives a message from a connection.
- ***recvmsg()*** establishes a connection and receives a message.
- ***selectsocket()*** waits for activity on a set of sockets.
- ***send()*** sends a message on an established connection.

- ***sendmsg()*** sends a message that can be split between buffers.
- ***sendto()*** establishes a connection and sends a message.
- ***setsockopt()*** sets options on sockets (described with ***getsockopt***).
- ***shutdown()*** shuts down part of a connection.
- ***socket()*** creates a socket.
- ***writesocket()*** sends a message to a socket.

Most functions return a value of -1 in case of an error. The error code is stored in errno, and can also be retrieved using the ***getsockopt()*** function, as in the following example:

```
int errcode, errlen;

i1 = connect(s, (struct sockaddr *)&socka, sizeof(socka));
if (i1 < 0)
{
     i1 = errno;
     if (getsockopt(s, SOL_SOCKET, SO_ERROR, &errcode, &errlen) >= 0)
     i1 = errcode;
     Print("connect: error %d\n", i1);/* additional error handling */
}
```

Here the value of errno is saved before calling ***getsockopt(),*** in case this
call fails and causes errno to be overwritten. The ***getsockopt()*** function should be used when possible in multitasking systems because errno is not reentrant. If a call to ***socket()*** returns -1, there is no socket number to refer to when trying to retrieve the error code. In this case, the error code must be retrieved from errno.
The ***gethostbyname()*** functions return a pointer to a host data structure.
If these functions fail, then a null pointer is returned.

4.1.1 Accept

- **Description**: Accepts a connection on a socket.
- **Syntax** :int accept(int s, struct sockaddr * name, int * namelen);
- **Input Parameter:**

  s Socket identifier.

  name On return, this provides information about the remote end of the connection.

  Namelen On entry, this is a pointer to an integer containing the size of the name structure, and on return this pointer points to the size of the returned structure. This size will not change.

The ***accept()*** call is used by a server application to perform a passive open for a socket. The socket will remain in the LISTEN state until a client establishes a connection with the port offered by the server. The return value from this function is an identifier for a newly created socket over which communication with the remote client can occur. The original socket remains in the LISTEN state, and can be used in a subsequent call to ***accept()*** to provide additional connections.

- **Reference**: ***socket, bind, listen***
- **Return Value**

  -1 Error.

  >= 0 Socket identifier for the established connection.

- **Example**

```
int s1, s2;
int socksz;
struct sockaddr_in socka;
…
socksz = sizeof(socka);
memset(&socka, 0, sizeof(socka));
socka.sin_family = AF_INET;
s2 = accept(s1, (struct sockaddr *)&socka,
&socksz);
if (s2 < 0)
Print(“Error in accept\n”);
```

4.1.2 Bind

- **Description** :Binds a name to a socket.
- **Syntax** :int bind(int s, struct sockaddr * name, int namelen);
- **Input Parameter:**

  s Socket identifier.

  name Structure that identifies the remote end of the connection. The sin_family member of the structure can be left as 0 to accept connections on any attached network interface.

  Namelen Size of name.

A server application uses the ***bind()*** function to specify the local Internet address and port number for a connection. The port number is the port that the server will be listening on. A call to ***bind()*** can also optionally be called by a client application before calling ***connect()***.

- **Reference**: socket, listen, accept, closesocket

- **Return Value**

  -1 Error.

  0 Success. The Internet address and port number have been associated with the local end of the socket.
- **Example**

```
int rc; /* return code */
int s; /* socket identifier */
struct sockaddr_in socka; /* local port, etc */
…
memset(&socka, 0, sizeof(socka));
socka.sin_family = AF_INET;
socka.sin_port = htons(1100);

rc = bind(s, (struct sockaddr *)&socka,
sizeof(socka));
if (rc < 0)
Print("Error in bind\n");
```

  In this example, 1100 is the local port number to be used. A client performing a connect() to this server would also use port number 1100.

4.1.3 Closesocket

- **Description** :Closes a socket.
- **Syntax** :int closesocket(int s);
- **Input Parameter:**

  s Socket identifier.

The ***closesocket()*** function is used to close a socket. This function is the
same as the regular BSD Sockets ***close()*** function, but it has been renamed to avoid conflicts with the ***close()*** function that operates on file descriptors.

- **Reference**: ***socket***
- **Return Value**

  -1 Error.

  0 Close was successful.

4.1.4 Connect

- **Description** : Initiates a connection on a socket.
- **Syntax** :int connect(int s, struct sockaddr * name, int namelen);
- **Input Parameter:**

s	Socket identifier.
Name	Structure that identifies the remote end of the connection.
Namelen	Size of name.

The ***connect()*** function performs an active open, allowing a client application to establish a connection with a remote server. The name structure is used to specify the Internet address and port number for the remote end of the connection. The Internet address is usually retrieved using the ***gethostbyname_r()*** function.

- **Reference**: ***closesocket***
- **Return Value**

-1	Error.
0	Success. A connection has been established with the remote server.

- **Example**

```
int rc; /* return code */
struct sockaddr_in socka; /* internet address */
/* and port number */
struct hostent hostent; /* for retrieving IP */
/* address from host */
unsigned char buff[BUFFLEN + 1];
…
memset(&socka, 0, sizeof(socka));
socka.sin_family = AF_INET;
gethostbyname_r("host1", &hostent, buff,
sizeof(buff), &rc);
if (rc < 0)
Print("Error: gethostbyname_r\n");
memcpy((char *)&socka.sin_addr,
(char *)hostent.h_addr_list[0], Iid_SZ);
socka.sin_port = htons(1100);
rc = connect(s, (struct sockaddr *)&socka,
sizeof(socka));
if (rc < 0)
```

Print(“Error connecting to remote server\n”);

Here you can see that &socka which is of type sockaddr_in * must be cast to a sockaddr * since this is what is expected by ***connect()***. This refers back to the previous discussion on structures and definitions.

4.1.5 fcntlsocket

- **Description** :Controls socket flags.
- **Syntax** :int fcntlsocket(int s, int cmd, int arg);
- **Input Parameter:**

  s Socket identifier.

The networking commands are:
F_GETFL get flags
F_SETFL set flags
This should of course be ***fcntl***, but we append "***socket***" to this to avoid naming conflicts.
The ***fcntlsocket()*** function allows a socket to be set to use non-blocking semantics, and also allows the current setting to be retrieved.
Networking uses only one flag: FNDELAY (or O_NDELAY; both names seem to be in use) for non-blocking I/O.

- **Reference**: Non-blocking sockets in Chapter 5, *Dynamic Protocol Interface.*
- **Return Value**

  The return value is -1 for error, 0 for successful SETFL, the current value of the flags for successful GETFL.

4.1.6 gethostbyname

- **Description** :Returns the IP address that corresponds to a host name.
- **Syntax** :struct hostent *gethostbyname(char * name);
- **Input Parameter:**
  name The name of the host for which the IP address should be obtained.

The ***gethostbyname()*** function is not reentrant. The ***gethostbyname_r()*** function should be used in situations where reentrancy is a requirement.
The name is normally of the form “hostname”, but “host/network” can be used when you want to talk using a specific port or network connection.

- **Reference**: ***gethostbyname_r***
- **Return Value**
  0   IP address could not be obtained.
  != 0 IP address is in the returned structure.
- **Example**
  hostentp = ***gethostbyname****(“testserver”);*
  if (hostentp != 0)
  *memcpy((char *)&socksav.sin_addr,*
  *(char *)hostentp->h_addr_list[0], 4);*

4.1.7 gethostbyname_r

- **Description** :Returns the IP address that corresponds to a host name.
- **Syntax** :struct hostent * gethostbyname_r(char * name, struct hostent* result, char * buff, int buflen, int * errcod);
- **Input Parameter:**

  name The name of the host for which the IP address should be obtained.

  Result Structure in which the IP address should be stored. buff Scratch buffer, which should provide at least 32 bytes.

  Buflen Size of buff.

  Errcod Return code from function.

The name is normally of the form “hostname”, but “host/network” can be used when you want to talk using a specific port or network connection.

The IP address of the host is placed into the structure hostent. This function is reentrant and is available in many but not all 4.3 BSD implementations.

- **Reference**: ***gethostbyname***
- **Return Value**

  0 IP address could not be obtained.

  != 0 IP address is in the returned structure.

  The hostent structure is defined as follows:

```
struct hostent {
char *h_name; /* name for host */
char **h_aliases; /* alias list */
int h_addrtype; /* host address type */
int h_length; /* length of address */
char **h_addr_list; /* list of addresses */
};
```

- **Example**

```
if (gethostbyname_r(“testserver”, &hostentp,
buff, sizeof(buff),&errval))
memcpy((char *)&socksav.sin_addr,
(char *)hostentp->h_addr_list[0], 4);
```

4.1.8 getpeername

- **Description** :Extracts the remote address information for a socket.
- **Syntax** :int getpeername(int s, struct sockaddr * name, int * namelen);
- **Input Parameter:**

  s Socket identifier.

  Name Structure into which the remote address information should be stored.

  Namelen A pointer to the length of the name structure.

The ***getpeername()*** function retrieves the remote address information and stores it in the supplied structure.

- **Return Value**

  -1 Error.

  0 Remote address was retrieved.

- **Example**

```
struct sockaddr_in socka;
int rc; /* return value */
int s; /* socket identifier */
…
s = socket(PF_INET, SOCK_DGRAM, 0);
…
rc = getpeername(s, (struct sockaddr *)&socka,
&socksize);
if (rc < 0)
Print("Error in getpeername\n");
```

4.1.9 getsockname

- **Description** :Extracts the local address information for a socket.
- **Syntax** :int getsockname(int s, struct sockaddr * name, int * namelen);
- **Input Parameter:**

  s Socket identifier.

  Name Structure into which the local address information should be stored.

  Namelen A pointer to the length of the name structure.

The ***getsockname()*** function retrieves the local address information and stores it in the supplied structure.

- **Return Value**

  -1 Error.

  0 Local address was retrieved.

- **Example**

```
struct sockaddr_in socka;
int rc; /* return value */
int s; /* socket identifier */
…
s = socket(PF_INET, SOCK_DGRAM, 0);
…
rc = getsockname(s, (struct sockaddr *)&socka,
&socksize);
if (rc < 0)
Print("Error in getsockname\n");
```

4.1.10 getsockopt, setsockopt

- **Description** :Gets and sets options on sockets.
- **Syntax** :int getsockopt(int s, int level, int optname, char * optval, int * optlen);

int setsockopt(int s, int level, int optname, char * optval, int * optlen);

- **Input Parameter:**

s	Socket handle.
level	See Table 4-1 below.
Optname	See Table 4-1 below.
Optval	Pointer to option value.
Optlen	Pointer to the size of the data stored in optval.

The functions in the following table manipulate socket options.

Table 4-1: Routines that Manipulate Socket Options

level	**optname**	**Description**
IPPROTO_IP	IP_OPTIONS	Options in IP Header
IPPROTO_TCP	TCP_MAXSEG	get TCP maximum segment
	TCP_NODELAY	don’t delay send
SOL_SOCKET	SO_BROADCAST	permit broadcast
	SO_DEBUG	debug flag
	SO_DONTROUTE	no routing
	SO_ERROR	get and clear error code
	SO_KEEPALIVE	keepalive probing
	SO_LINGER	linger on close
	SO_OOBINLINE	leave URG data inline
	SO_RCVBUF	receive buffer size
	SO_SNDBUF	send buffer size
	SO_REUSEADDR	local address reuse
	SO_TYPE	get socket type

- **Reference**: ***fctlsocket, ioctlsocket***

- **Return Value**

-1 Error.
0 Success. The optval pointer contains the option value for ***getsockopt()***; the option was set for ***setsockopt()***.

- **Example**

  rc = ***setsockopt****(s, SOL_SOCKET, SO_KEEPALIVE, 0, 0);*
  if (rc < 0)
  Print("Error in setsockopt\n");

4.1.11 ioctlsocket

- **Description** :Sets control parameters for a socket.
- **Syntax** :int ioctlsocket(int s, int request, char * arg);
- **Input Parameter:**

  s

  Socket identifier.

  request

  Request type

  SIOCATMARK checks out-of-bound mark.

  Arg

  Optional argument.

  arg is assigned 1 if the socket read is

  at the out-of-bound mark, 0 otherwise. arg is of type “int *”.

The ***ioctlsocket()*** function behaves the same as the regular BSD Sockets ***ioctl()*** function, except that it only accepts socket identifiers. The optional third argument is used as a pointer for the result. There is some variation in how this function is defined in BSD sockets: The second argument may be “unsigned long”, and of course the variable arguments are treated differently in non-ANSI C.

- **Reference**: ***getsockopt, setsockopt***
- **Return Value**

  -1 Error.

  0 Operation successful.

4.1.12 listen

- **Description** :Listens for connections.
- **Syntax** :int **listen**(int s, int backlog);
- **Input Parameter:**

  s

  Socket identifier.

  backlog

  Specifies the number of connections that will be held in a queue waiting to be accepted. This value includes connections that are in the SYN_RCVD state and connections that are in the ESTABLISHED state that have not yet been accepted by the application.

The ***listen()*** function is part of the sequence of functions that are called to perform a passive open. This call puts the socket into the LISTEN state.

- **Reference**: ***socket, bind, accept***
- **Return Value**

  -1 Error.

  0 Success.
- **Example**

```
int rc; /* return code */
int s; /* socket identifier */
…
rc = listen(s, 5);
if (rc < 0)
Print("Error calling listen\n");
```

4.1.13 readsocket

Description :Receives a message from a socket ID.

Syntax :int readsocket(int s, char * buf, int len);

- **Input Parameter:**

  s Socket identifier.

  buf Buffer into which received data will be stored.

  len Maximum number of bytes to be received.

  The ***readsocket()*** function behaves the same as the regular BSD Sockets

read() function, except that it only accepts socket identifiers.

- **Reference**: ***recv, recvfrom, recvmsg***
- **Return Value**

  -1 Error.

  >= 0 Number of bytes received.

4.1.14 recv

- **Description** :Receives a message.
- **Syntax** :int recv(int s, char * buf, int len, int flags);
- **Input Parameter:**

  s Socket identifier.

buf	Buffer into which received data will be stored.
len	Maximum number of bytes to be received.
flags	Allows for these options:

  MSG_OOB returns urgent data.
  MSG_PEEK returns information, allowing it to be read again on a subsequent call.The flag MSG_WAITALL is not supported.
- **Reference**: ***recvfrom, recvmsg***
- **Return Value**

  -1 Error.
  >= 0 Number of bytes received.

The following error codes could be returned in errno or through ***getsockopt()*** if ***recv()*** returns indicating an error:

EWOULDBLOCK
Only returns if the socket is set up as non-blocking. If thisis the case, then a call to ***recv()*** can check for EWOULDBLOCK and try again later, effectively polling.

EWTIMEDOUT
Would only be returned if previously the macro *SOCKET_RXTOUT* was used to adjust the receive timeout of the socket. The application could call ***recv()***again later.

EOPNOTSUPP
1. The call to ***recv()*** asked for out-of-band data (the flags parameter had MSG_OOB set), and none was available.
2. The call to ***recv()*** didn't ask for out-of-band data, and there is some that needs to be received.

EBADF
Invalid socket handle. No need to close, since that call would return an error as well.

ECONNABORTED
A definite fatal error. Usually results from a retransmission timeout or

reception of a RST segment. Time to close the socket.

- **Example**

  int rc; / return code */*
  int s1, s2; / socket identifiers */*
  unsigned char buff[BUFFLEN]; / read buffer */*
  …
  *s2 = accept(s1, (struct sockaddr *)&socka,*
  &socksize);
  …
  rc = ***recv****(s2, buff, 2, 0);*
  if (rc < 0)
  Print(“Error receiving data.\n”);
  else if (rc == 2)
  Print(“Success: read 2 bytes\n”);
  else
  Print(Error: did not retrieve 2 bytes\n”);

  Notice in this example that **recv()** uses the second socket identifier, the one returned from the **accept()**, not the original socket which is used as an argument to **accept()**.

4.1.15 recvfrom

- **Description** :Receives a message from a connection.
- **Syntax** :int recvfrom(int s, char * buf, int len, int flags,

struct sockaddr * from, int * fromlen);

- **Input Parameter:**

s	Socket identifier.
buf	Buffer in which information will be stored.
len	Number of bytes to receive.

Flags

Specifies optional behavior:

MSG_OOB returns urgent data.

MSG_PEEK returns information, allowing it to be read gain on a subsequent call.

From

Specifies the remote host to which the connection should be made.

Fromlen

Size of the from data structure.

The ***recvfrom()*** function allows a connection to be made and a message to be read from the connection. The flag MSG_WAITALL is not supported.

- **Reference**: ***recv, recvmsg***
- **Return Value**

-1 Error.

>= 0 Number of bytes received.

- **Example**

*The **accept()** or **connect()** call is not needed here since **recvfrom()** establishes the connection before reading.*

```
int s1, s2; /* socket identifiers */
int rc; /* return code */
unsigned char buff[BUFFLEN]; /* read buffer */
struct sockaddr_in socka; /* remote host address */
…
memset(&socka, 0, sizeof(socka));
socka.sin_family = AF_INET;
gethostbyname_r(hnp, &hostent, buff, sizeof(buff),
&i1);
if (i1 < 0)
{
Print("%s not known\n", hnp);
closesocket(s2);
return -1;
}
memcpy((char *)&socka.sin_addr,
(char *)hostent.h_addr_list[0], Iid_SZ);
socka.sin_port = htons(1100);
rc = recvfrom(s2, buff, 8, 0, (struct sockaddr
*)&socka, &socksize);
if (rc != 8)
Print("Error in recvfrom\n");
```

4.1.16 recvmsg

- **Description** :Receives a message.
- **Syntax** :int recvmsg(int s, msghdr * msg, int flags);
- **Input Parameter:**

  s Socket identifier.

  msg Pointer to structure that describes how received data should be stored. This structure is shown below.

  flags Specifies optional behavior:

    MSG_OOB returns urgent data.

    MSG_PEEK returns information, allowing it to be read again on a subsequent call.

The ***recvmsg()*** function is the most general of the ***recv*** functions. This function allows a connection to be established and read with one call. The flag MSG_WAITALL is not supported.

Here is the definition of the msghdr structure:

```
struct msghdr { /* Message header for recvmsg */
char *msg_name; /* optional address */
int msg_namelen; /* size of address */
struct iovec *msg_iov; /* scatter/gather arra */
int msg_iovlen; /* num of elems in msg_iov */
char *msg_accrights; /* access rights */
int msg_accrightslen;
};
struct iovec { /* address and length */
char *iov_base; /* base */
int iov_len; /* size */
};
```

- **Reference**: ***recv, recvfrom***
- **Return Value**

  -1 Error.

  >= 0 Number of bytes received.

4.1.17 selectsocket

- **Description** :Waits for activity on a set of sockets.
- **Syntax** :int selectsocket(int nfds, fd_set * readfds, fd_set * writefds, fd_set * exceptfds, struct timeval * timeout);
- **Input Parameter:**

  nfds

  Number of sockets. Watch out for "off by one" errors.
  For example, if the highest value of the descriptors that should be evaluated is n, nfds should be set to n+1.

  readfds

  Socket identifiers for which ***selectsocket()*** should return if data becomes available or the state of the socket changes.

  writefds

  Socket identifiers for which ***selectsocket()*** should return if the socket can accept more data or if there is an error.

  exceptfds

  Socket identifiers for which ***selectsocket()*** should return if out-of-band data is available.

  timeout

  Specifies time after which ***selectsocket()*** will return if none of the specified conditions occurs.

  This is a general UNIX routine, but handles sockets as well as files.
  The fd_set structures specify which sockets (range 0 to nfds-1) are considered.These macros can be used to manipulate fd_set:
  FD_ZERO(&fd_set) clears the socket list
  FD_SET(s, &fd_set) adds socket s
  FD_CLR(s, &fd_set) removes socket s
  FD_ISSET(s, &fd_set) non-zero if s included

  When ***selectsocket()*** returns, there are bits in the fd-set structures only for those sockets that satisfied the condition.
  Structure timeval gives the timeout value:

  struct timeval { / Time-out format for select() */*
  long tv_sec; / seconds */*
  long tv_usec; / microseconds */*
  };

A NULL pointer means an infinite timeout. If the structure contains the value 0, then the descriptors will be checked once and the call to ***selectsocket()*** will return without delay. This is useful for application-level polling.

- **Return Value**
  -1 Error. Note that this should not occur in the current implementation.
  0 Timeout occurred.
  >0 This number of sockets are ready for the requested operations.
- **Example**

```
int s1, s2, s3; /* sockets */
int rc; /* return code */
fd_set socket_set1, socket_set2;
…
FD_ZERO(&socket_set1);
FD_ZERO(&socket_set2);
FD_SET(s1, &socket_set1);
FD_SET(s3, &socket_set1);
FD_SET(s2, &socket_set2);
rc = selectsocket(3, socket_set1, socket_set2, 0,
NULL);

if (rc < 0)
    Print("Error, no sockets ready.\n");
else
    Print("%d sockets ready.\n", rc);
if (FD_ISSET(s1, &socket_set1))
    Print("Socket 1 is ready to be read.\n");
else if (FD_ISSET(s2, &socket_set2))
    Print("Socket 2 is ready to be written\n");
else if (FD_ISSET(s3, &socket_set3))
    Print("Socket 3 is ready to be read.\n");
else
    Print("Error.\n");
```

4.1.18 send

- **Description** :Sends a message on an established connection.
- **Syntax** :int send(int s, char * buf, int len, int flags);
- **Input Parameter:**

  s Socket identifier.

  buf Pointer to data to be sent.

  len Number of bytes to send.

  flags

  Allows for these options:

  MSG_OOB sends the data as urgent data

  MSG_DONTROUTE ensures that the message is not sent through a default router.

  The ***send()*** function can be used with sockets for which the connection has previously been established.
- **Reference**: ***sendto, sendmsg***
- **Return Value**

  -1 Error.

  >= 0 Number of bytes sent.

  If ***send()*** returns indicating an error, the following error codes could be returned in errno or through ***getsockopt()***:

  EBADF

  The socket descriptor is invalid, or another process is using the socket at the moment.

  ESHUTDOWN

  The application has already requested that the sending side of the socket be shut down. No further data can be sent through this socket.

  ECONNABORTED

  An error has occurred on this socket. The socket should be closed.

  EMSGSIZE

  A non-stream socket has been asked to send more information than can be written at once through the socket.

  ENOBUFS

  The system is out of buffers for sending data. The call to ***send()*** can be retried later.

- **Example**

```
int s2; /* socket identifier */
int rc; /* return code */
unsigned char buff[BUFFLEN];
…
rc = send(s2, buff, sizeof(buff), 0);
if (rc < 0)
Print("Error sending data\n");
```

4.1.19 sendmsg

- **Description** :Sends a message that can be split between buffers.
- **Syntax** :int sendmsg(int s, msghdr * msg, int flags);
- **Input Parameter:**

  s Socket identifier.

  msg Pointer to structure that describes the data to be sent. This structure is shown below.

  flags Specifies optional behavior:

  MSG_OOB sends the data as urgent data

  MSG_DONTROUTE ensures that the message is not sent through a default router.

  The ***sendmsg()*** function is a send function that allows the data to be sent to be split into an array of buffers.

  Here is the definition of the msghdr structure:

  struct msghdr { / Message header for recvmsg */*
  *char *msg_name; /* optional address */*
  int msg_namelen; / size of address */*
  *struct iovec *msg_iov; /* scatter/gather arra */*
  int msg_iovlen; / num of elems in msg_iov */*
  *char *msg_accrights; /* access rights */*
  int msg_accrightslen;
  };
  struct iovec { / address and length */*
  *char *iov_base; /* base */*
  int iov_len; / size */*
  };
- **Reference**: ***send, sendto***
- **Return Value**

  -1 Error.

  >= 0 Number of bytes sent

4.1.20 sendto

- **Description** :Send a message.
- **Syntax** :int sendto(int s, char * buf, int len, int flags, struct sockaddr * to, int tolen);
- **Input Parameter:**
  s Socket identifier.
  buf Buffer from which information will be sent.
  len Number of bytes to send.
  flags Specifies optional behavior:
    MSG_OOB sends the data as urgent data.
    MSG_DONTROUTE ensures that the message is
    not sent through a default router.
  to Specifies the remote host to which the connection should
  be made.
  tolen Size of the to data structure.
  The ***sendto()*** function allows a connection to be made and a
  message to be written to the connection.
- Reference: ***send, sendmsg***
- **Return Value**
    -1 Error.
    >= 0 Number of bytes sent.

- **Example**
  rc = ***sendto****(s, "HIJKLMNO", 8, 0,*
  *(struct sockaddr *)&socka, sizeof(socka));*
  if (rc < 0)
  Print("Error sending\n");

4.1.21 shutdown

- **Description** :Shuts down part of a connection.
- **Syntax** :int shutdown(int s, int how);
- **Input Parameter:**
  s Socket identifier.
  how Describes type of shutdown:
  0 shuts down receive data path
  1 shuts down send data path, TCP sends FIN
  2 shuts down send and receive path

The ***shutdown()*** function is useful for fully specifying the limited closure of a connection. Normally the ***closesocket()*** function is used to fully close a connection.

- **Reference**: ***closesocket***
- **Return Value**
  -1 Error.
  0 Shutdown successful.

4.1.22 socket

- **Description** :Creates a socket.
- **Syntax** :int socket(int domain, int type, int protocol);
- **Input Parameter:**

  Domain

  this should always be PF_INET.

  type

  one of three constants is for this parameter:
  SOCK_STREAM stream socket (TCP/IP)
  SOCK_DGRAM datagram socket (UDP/IP)
  SOCK_RAW raw-protocol interface

  protocol

  This can be specified as 0.

A call to ***socket()*** will create a socket of the specified type. A socket must be created before any other socket calls are used.

- **Reference**: ***closesocket***
- **Return Value**

  -1 Error.
  >= 0 The newly created socket can be accessed through this handle.

  If ***socket()*** returns with an error indication, the value in errno or obtained through ***getsockopt()*** can be interpreted as follows:

  EPROTONOSUPPORT

  The requested protocol is not available. Perhaps SOCK_STREAM was specified, but TCP support is not configured for the underlying stack.

- **Example**

  int s; / a socket */*
  …
  s = ***socket****(PF_INET, SOCK_DGRAM, 0);*
  if (s < 0)
  Print("Error opening socket\n");

4.1.23 writesocket

- **Description** :Sends a message to a socket.
- **Syntax** :int writesocket(int s, char * buf, int len);
- **Input Parameter:**

  s Socket identifier.

  buf Pointer to data to be sent.

  len Number of bytes to send.

  The ***writesocket()*** function behaves the same as the regular BSD Sockets

  write() function, except that it only accepts socket identifiers.
- **Reference**: ***send, sendto, sendmsg***
- **Return Value**

  -1 Error.

  >= 0 Number of bytes sent.

5. Dynamic Protocol Interface

5.1 Overview

This Chapter details the usage of ICPDAS Dynamic Protocol Interface. The Dynamic Protocol Interface provides a simple and efficient interface to implement the Networking program.

5.2 Blocking Versus Non-Blocking Operation

There are two modes of operation that affect how your application deals with network events in a non-multitasking system: Blocking and non-blocking.
Blocking is the default mode. This mode will halt processing while waiting for a network event to complete or timeout. An example of this would be a wait for a return from a TCP open. Blocking mode would halt processing until the open returned a connection number or timed out. This behavior is usually unsatisfactory for most embedded systems.
Non-blocking allows processing to continue while polling the status of the network event. Non-blocking is desirable in a non-multitasking system because it makes efficient use of CPU time while waiting for network events to complete.
Non-blocking issues are addressed in the appropriate sections in this chapter. An example of non-blocking is also given at the end of this chapter.

5.3 Initialization and Termination

Ninit() performs general initialization, such as initialization of tables and buffers. It must be the first network function called and can't be called again unless the function ***Nterm()*** has been called first.
Portinit() and ***Portterm()*** are used to initialize and shut down the system's network interfaces.
Detailed descriptions of these functions follow.

5.3.1 Ninit

- **Description**: Performs general network initialization.
- **Syntax**: int Ninit(void);
  Ninit() takes no parameters.
- **Reference**: ***Nterm, Portinit, Portterm***
- **Return Value**

  0 Success.

  ENOBUFS

  No buffers configured. Check tcpip**.h** variables

  NCONFIGS

  and NNETS.

  USER

  User-defined error return from ***LOCALSETUP()*** found in **tcpip.h**.
- **Example**

```
main()
{
/* initialize all connections */
if (Ninit() < 0)
/* process error */
}
```

5.3.2 Nterm

- **Description** :Shuts down networking.
- **Syntax** :int Nterm(void);

Nterm() takes no parameters. Any open network interfaces will be shut down, so ***Portterm()*** does not need to be called before ***Nterm()***. Network support can be restarted by making a call to ***Ninit()***.

- **Reference**: ***Ninit, Portinit, Portterm***
- **Return Value**

  0 Always returns 0.
- **Example**

  / shut down all network connections */*

  ***Nterm**();*

5.3.3 Portinit

- **Description** :Initializes one or more network interfaces.
- **Syntax:** int Portinit(char * name);
- **Input Parameter:**
  name If "*****", then all network interfaces for this system will be initialized.
  Portinit() initializes the specified network interfaces. Note that all interfaces can be initialized all at once, or individually. The initialization routine will prepare the device driver to transmit and receive network frames, and will install and enable the interrupt service routine for the network device driver.
- **Reference**: ***Ninit(), Nterm(), Portterm()***
- **Return Value**
  NE_PARAM
  Parameter error. The device driver did not accept the initialization string specified in **tcpip.lib**.
  EHOSTUNREACH
  The specified port (when "*****" is not used) is not in **tcpip.lib** for this host. This could also mean that the host name is wrong.
  NE_HWERR
  A hardware error occurred. Generally, this indicates an error with the network controller.

- **Examples**

```
/* initialize all network interfaces */
main()
{
  if (Ninit() < 0)
  /* process error */
  if (Portinit("*") < 0)
  /* process error */
}

/* Initialize a specific network interface */
main()
{
   if (Ninit() < 0)
   /* process error */
   if (Portinit("com1") < 0)
   /* process error */
}
```

5.3.4 Portterm

- **Description** :Shuts down one or more network interfaces.
- **Syntax:** int Portterm(char * name);
- **Input Parameter:**
  name If “*****”, then all network interfaces for this system will be shut down.
  Shuts down the specified network interfaces. Note that all interfaces can be shut down at once, or individually. The shut down routine will put the network controller into an idle state, and restore the interrupt vector associated with the network device driver to its original state. The shutdown is reversible: Just make another call to ***Portinit()***. A call to ***Portterm()*** can be omitted prior to calling ***Nterm()***, because ***Nterm()*** automatically calls ***Portterm()***.
- **Reference**: ***Ninit(), Nterm(), Portinit()***
- **Return Value**
  0 Always returns 0.
- **Examples**
  /* shut down all network connections */
  Portterm(“*”);
  /* shut down a specific network connection */
  Portterm(“com1”);

5.4 Connections

Connections behave very much like files: You can open and close a connection, you can read data from it, and write data to it. The main difference is that a connection has a user at each end, and a file has only one user. The data you read is the data the other user wrote, and vice versa.

The library offers the user two basic kinds of connections: TCP and UDP. There are two primary differences:

- TCP performs error correction and flow control, and UDP does not. You can read TCP like a local disk file: You want to check for errors, but they should not occur and if they do you quit. Doing this with UDP would be difficult, and writing applications using UDP is quite cumbersome. It is best to leave UDP for prewritten
  applications, such as TFTP and BOOTP.
- UDP is a packet protocol, and TCP is a byte-stream protocol. With TCP, you can't predict with certainty how many bytes a read will return, or how many reads you'll need for a given amount of data.

Port numbers are used to match the two ends of the connection. If your local port number is my remote port and vice versa, then we have a connection.

Normally one end performs an active open and the other a passive open. The system performing a passive open is typically running a server application. This system will wait until it receives an indication from a client application performing an active open.

Connections

5.5 Open, Close, Read, and Write

These four routines (plus the startup and shutdown) are the only user level network functions required to write an application using the Library. This might surprise you, especially if you have seen network packages that go something like:

call TCPwrite
call Ipwrite
call DRIVERwrite
...

The library uses a table-driven protocol stack structure. Each protocol level has only one public symbol: The name of the protocol table.
The library performs all necessary calls through these protocol tables.
The user only has to call a general high-level function that is the same for all protocol configurations.
The open function specifies which protocols, and in which order, are to be used. There are no restrictions on the protocol stack as such, but of course not all combinations make sense.
Connections

5.5.1 Nopen

- **Description** :Opens a connection.
- **Syntax:** int Nopen(char * to, char * protoc, int lp, int rp, int flags);
- **Input Parameter:**

to

String specifying the name of the remote system. This can take one of the following forms:

"host"	Remote host, shortest route.
"host/network"	Remote host, using named network.
"*"	Any host, used for passive open or broadcast.
"*/network"	Any host, using named network.
"n1.n2.n3.n4"	IP address of remote system.

protoc

String specifying the transport and network layer protocols, separated by a slash. Typical values would be "TCP/IP", "UDP/IP" or "ICMP/IP".

lp

Local port number. For an active open, this is often an ephemeral port, and a suitable random value can be obtained using the utility function ***Nportno()***. For a passive open, the well-known port number should be used.

rp

Remote port number. For an active open, this should be the well-known port for the service used in the connection. For a passive open, this value should be specified as 0, and any remote port will be accepted for the connection.

flags

Normally 0, but for a non-blocking open, you can specify the flag S_NOWA, and the call will return without blocking. In order to determine if the connection is established, use the macro ***SOCKET_ISOPEN()***. Also, for UDP connections, you can use the value S_NOCON to cause the connection to behave in a connectionless manner.
When you specify S_NOCON, the connection will accept all UDP messages directed to the local port, regardless of the originating IP

address or UDP port.
This information is stored so that a call to ***Nread()***
followed by a call to ***Nwrite()*** will respond to the source of the message that was just read.

Nopen() is used for both active and passive opens. The behavior is determined by the parameters supplied to the function. Several examples follow to further illustrate the use of the function.
A passive open will wait indefinitely. An active open for TCP will return when the connection has been made, but it times out in a couple of minutes if there is no answer.

- **Reference**: ***Nclose(), Nread(), Nwrite()***
- **Return Value**

  conno
  A return value >= 0 is a connection number.
  This is the handle for further communication on the connection.

  EHOSTUNREACH
  Could not access the remote system.

  ENOBUFS
  NCONNS in **tcpip.h** is not large enough.

  ETIMEDOUT
  Timeout.

  ECONNABORTED
  Remote host refused the connection.

- **Examples**

  / An active open from host1 that causes TCP to send out open requests to port 1000. The local port number is dynamically and randomly assigned with the function Nportno(). */*
  / host1 */*
  int conno, myport; / connection and port number */*
  myport = ***Nportno****();*
  conno = ***Nopen****("host2", "TCP/IP", myport, 1000, 0);*
  if (conno < 0)
  / process error */*
  / A passive open at host2 that waits for and accepts calls from anyone who asks for port number 1000. This type of*

*open would be done by a server */*
/ host2 */*
int conno; / connection number */*
conno = ***Nopen****("*", "TCP/IP", 1000, 0, 0);*
if (conno < 0)
/ process error */*
/ A UDP open at host1 for hostA through port com1 would look like this: */*
/ host1 */*
conno = ***Nopen****("hostA/com1", "UDP/IP", 1000, 1010, 0);*
/ The port "com1" is a host1 port, not a hostA port. This form of open may be needed if there are two connections between host1 and hostA. In this manner, "com1" serves to identify which connection is being opened. Note "com1" References field 2 in the network configuration table in* ***tcpip.lib*** **/*
/ To send and receive ICMP messages, you can use the form: */*
/ host1 */*
conno = ***Nopen****("host2", "ICMP/IP", 1000, 1010, 0);*
Connections

/ This is a special situation; see, for instance, PING.C for the use of ICMP. */*
/ Perform a non-blocking OPEN and do some processing while polling for the OPEN connection. */*
conno = ***Nopen****("*", "TCP/IP", 1000, 0, S_NOWA);*
if (conno < 0)
/ handle error condition */*
while (!SOCKET_ISOPEN(conno))
/ perform other processing */*

5.5.2 Nclose

- **Description** :Closes a connection.
- **Syntax:** int Nclose(int conno);
- **Input Parameter:**
  conno The connection number previously returned from a call to ***Nopen()***.

Nclose closes a connection, possibly waiting for a complete close handshake. In no case should the application retry the close. In some cases (as with TCP), the connection block will actually be freed after a minute or so, but this is automatic, and the application should not touch the connection after the close.

- **Reference**: ***Nopen(), Nread(), Nwrite()***
- **Return Value**
  0 Normal close.
  EBADF The connection number is invalid. No closing was performed.
  ECONNABORTED Protocol problem. If you have been writing data to the other system, consider the dataunsafe. Connection is closed.
- **Example**
  int error; / error code */*
  int conno; / connection number */*
  error = ***Nclose****(conno); /* close the connection */*
  if (error < 0)
  / process error */*

5.5.3 Nread

- **Description** :Reads a message from a connection.
- **Syntax:** int Nread(int conno, char * buff, int len);
- **Input Parameter:**

  conno Connection number.
  buff Buffer to store message.
  len Size of the buffer.

Reads a message from a connection into the specified buffer. For a blocking socket, the call will block until information is available to be read, or until a timeout occurs. The timeout can be adjusted using the ***SOCKET_RXTOUT()*** macro.
For TCP connections, ***Nread()*** may return up to the maximum amount of information that will fit in one internal message buffer. This will be less than MAXBUF bytes. For UDP connections, the data from the next UDP message will be returned.

- **Reference**: ***Nclose(), Nopen(), Nwrite()***
- **Return Value**

  0
  The remote system has closed the connection.
  count
  Values > 0 indicate the number of bytes read.
  EBADF
  The connection number is not valid.
  EWOULDBLOCK
  Non-blocking connection can't proceed.
  Read would be retried.
  ETIMEDOUT
  Timeout. Read can be retried.
  ECONNABORTED
  Protocol problem. Normally the application should close the connection.
  EMSGSIZE
  The message is too long for the supplied buffer.

Example

```
/* user defined input buffer size */
#define MAX_BUFFER_SIZE 80
```

```
int error; /* error code */
int conno; /* connection Number */
char buff[MAX_BUFFER_SIZE]; /* data input buffer */
/* read data into “buff” from connection number “conno” */
error = Nread(conno, buff, sizeof(buff));
if (error < 0)
/* process error */
```

The constant MAX_BUFFER_SIZE could be replaced with the constant MAXBUF defined in file **tcpip.h**. A call to ***Nread()*** cannot return more than MAXBUF bytes.

5.5.4 Nwrite

- **Description** :Writes a message to a connection.
- **Syntax:** int Nwrite(int conno, char * buff, int len);
- **Input Parameter:**
  conno Connection number.
  buff Buffer containing message.
  len Number of bytes to write.

Nwrite() writes a message to a connection from the specified buffer. The largest buffer passed to ***Nwrite()*** should not exceed the value given by the ***SOCKET_MAXDAT()*** macro. For TCP connections, this will reflect the maximum segment size that is indicated by the remote TCP when the connection is established. For UDP connections, this value will reflect the MTU imposed by the link layer. These values will generally be at least 256 bytes, so it is reasonable to write out small buffers directly.

- **Reference**: ***Nclose(), Nopen(), Nread()***
- **Return Value**
  count
  Values >= 0 indicate the number of byteswritten.
  EBADF
  The connection number is not valid.
  ETIMEDOUT
  Timeout. With TCP in blocking mode, this probably means the other end did not send acknowledgments as expected. It could also mean an extremely heavy system load and that a timeout occurred before the acknowledgment could be received. The connection should be closed. In nonblocking mode, the write should be retried.
  ECONNABORTED
  Protocol problem. Normally the application should close the connection.
  EMSGSIZE
  The message is too large for the internal buffer.
- **Example**

```
/* user defined output buffer size */
#define MAX_BUFFER_SIZE 80
int error; /* error code */
int conno; /* connection Number */
char buff[MAX_BUFFER_SIZE]; /* data output buffer */
/* write data stored in “buff” to connection number “conno”
*/
error = Nwrite(conno, buff, sizeof(buff));
if (error < 0)
/* process error */
/* dynamically sized write buffer */
int error; /* error code */
int conno; /* connection Number */
int maxwrite; /* maximum write size */
char buff[MAXBUF]; /* data buffer */
/* write data stored in “buff” to connection number “conno”
*/
conno = Nopen(“host”, “TCP/IP”, Nportno(), 1050, 0);
if (conno < 0)
/* process error */
maxwrite = SOCKET_MAXDAT(conno);
error = Nwrite(conno, buff, maxwrite);
if (error < 0)
/* process error */
```

5.6 Dynamic Protocol Interface Macros

The following macros are useful for obtaining additional information or setting control information for a connection, and are described in this section:

- ***SOCKET_NOBLOCK*** sets the connection for non-blocking operation.
- ***SOCKET_BLOCK*** sets the connection for blocking operation.
- ***SOCKET_ISOPEN*** checks to see if a connection has entered the ESTABLISHED state.
- ***SOCKET_HASDATA*** checks to see if a message is available on a connection.
- ***SOCKET_CANSEND*** checks to see if a connection can accept data to be written.
- ***SOCKET_TESTFIN*** checks to see if the remote end of the connection has closed.
- ***SOCKET_MAXDAT*** provides the maximum size of a buffer than can be written to a connection.
- ***SOCKET_RXTOUT*** sets the receive timeout for a connection.
- ***SOCKET_IPADDR*** provides the IP address of the remote end of a connection.
- ***SOCKET_OWNIPADDR*** provides the IP address of the local end of a connection.
- ***SOCKET_PUSH*** sets the PSH flag on the next outgoing TCP segment.
- ***SOCKET_FIN*** sets the FIN flag on the next outgoing TCP segment. Connections

5.6.1 SOCKET_NOBLOCK

- **Description**: Sets the connection for non-blocking operation.
- **Syntax:** SOCKET_NOBLOCK(conno)
- **Input Parameter:**
  conno The connection for which non-blocking operation should be set.

When non-blocking operation is set, calls to network functions that normally would need to wait for network activity in order to be completed will return the negative value EWOULDBLOCK when such a condition is encountered.

5.6.2 SOCKET_BLOCK

- **Description**:Sets the connection for blocking operation.
- **Syntax:**SOCKET_BLOCK(conno)
- **Input Parameter:**
  conno The connection for which blocking operation should be set.

When blocking operation is set, calls to network functions run to completion, or return a timeout error if an associated time limit is exceeded. Blocking operation is the default behavior for network functions, and this call will only be needed to return a non-blocking connection to blocking operation.

5.6.3 SOCKET_ISOPEN

- **Description**:Checks to see if a connection has entered the ESTABLISHED state.
- **Syntax:**SOCKET_ISOPEN(conno)
- **Input Parameter:**
  conno The connection that should be checked for the ESTABLISHED state.

This macro will evaluate as 0 if the connection is not in the ESTABLISHED state, and 1 if the connection is in the ESTABLISHED state. This macro is useful for connections that call ***Nopen()*** with the S_NOWA flag, so that after requesting a connection, the connection can be checked to see if it has been established.

5.6.4 SOCKET_HASDATA

- **Description**: Checks to see if a message is available on a connection.
- **Syntax:** SOCKET_HASDATA(conno)
- **Input Parameter:**
  conno The connection that should be checked for an available message.

This macro will evaluate as 0 if no information is available, or nonzero if data is available.

5.6.5 SOCKET_CANSEND

- **Description**: Checks to see if a connection can accept data to be written.
- **Syntax:** SOCKET_HASDATA(conno, len)
- **Input Parameter:**
  conno
  The connection that should be checked for room for writing.
  len The amount of data to be written.

This macro will evaluate as 0 if the amount of data is more than can be written out immediately, or non-zero if the data length specified can be written.

5.6.6 SOCKET_TESTFIN

- **Description**: Checks to see if the remote end of the connection has closed.
- **Syntax:** SOCKET_TESTFIN(conno)
- **Input Parameter:**
  conno The connection that should be checked for a close from the remote end.

This macro will evaluate as 0 if the remote end of the connection has not yet closed, or non-zero if the remote system has closed.

5.6.7 SOCKET_MAXDAT

- **Description**:Provides the maximum size of a buffer than can be written to a connection.

- **Syntax:** SOCKET_MAXDAT(conno)
- **Input Parameter:**
  conno The connection for which the maximum buffer size should be determined

This macro will evaluate to the maximum number of bytes that can be accepted by the connection in a call to ***Nwrite()***.

5.6.8 SOCKET_RXTOUT

- **Description**: Sets the receive timeout for a connection.
- **Syntax:** SOCKET_RXTOUT(conno, tout)
- **Input Parameter:**

Conno	The connection for which the timeout is to be adjusted.
Tout	The new timeout, in milliseconds.

5.6.9 SOCKET_IPADDR

- **Description**: Provides the IP address of the remote end of a connection.
- **Syntax:** SOCKET_IPADDR(conno)
- **Input Parameter:**
  conno The connection for which the remote IP address is to be returned. The data type of the result is Iid.

5.6.10 SOCKET_OWNIPADDR

- **Description**: Provides the IP address of the local end of a connection.
- **Syntax:** SOCKET_OWNIPADDR(conno)
- **Input Parameter:**
  conno The connection for which the local IP address is to be returned.

The data type of the result is Iid. This macro is useful for systems that have more than one network interface. The IP address returned will be that of the interface that is used for the connection.

5.6.11 SOCKET_PUSH

- **Description**: Sets the PSH flag on the next outgoing TCP segment.

- **Syntax:** SOCKET_PUSH(conno)
- **Input Parameter:**
  conno The connection for which the next outgoing segment should include the PSH flag.

The next TCP segment to be written following a call to this macro will have the PSH flag set in the TCP header. This is useful for indicating to the TCP on the remote system that all internally buffered segments up through this segment should be delivered to the application as soon as possible.

5.6.12 SOCKET_FIN

- **Description**: Sets the FIN flag on the next outgoing TCP segment.
- **Syntax:** SOCKET_FIN(conno)
- **Input Parameter:**
  conno The connection for which the next outgoing segment should include the PSH flag.

The next TCP segment to be written following a call to this macro will have the FIN flag set in the TCP header. This is useful for shutting down a connection at the same time that the last segment is sent. Following the write, call ***Nclose()*** to finish closing the connection. ***Nclose()*** will not send a FIN segment in this case.

- **Examples**
  The following text provides examples of:
  • Broadcasting
  • Non-Blocking Operations

Broadcasting Example

For broadcasting messages to all hosts on the network, use host name "*" in the active open, and then, do an ***Nwrite()***. For instance:

host1:

conno = Nopen("/enet", "UDP/IP", 1010, 1000, 0);*

.....

stat = Nwrite(conno, buf, len);

In this case, "enet" is the portname for the network, and "" represents all hosts. The receiving hosts'* ***open()*** *would generally be a passive open.*

```
host2:
conno = Nopen("*", "UDP/IP", 1000, 0, 0);
....
stat = Nread(conno, buf, len);
```
The receiving hosts must be listening on the same port number that the broadcasting host is sending to (e.g., 1000 in this case). Broadcasting should only be used for data links that support it in hardware, such as Ethernet. It should not be done at the TCP level. If the broadcasting host connects to several networks, the open call must specify the network name. Broadcasting is done to one network only Examples

Non-Blocking Operations Example

The following example shows how to read using non-blocking operations. Non-blocking writes will complicate an application quite a bit. If no multi-task is used, there is really no alternative to non-blocking operations. With multitasking, a heavy use (perhaps even any use) of non-blocking mode is not recommended.

```
conno = Nopen("*", "TCP/IP", 1001, 0, S_NOWA);
if (conno < 0) /* ERROR */
while (!SOCKET_ISOPEN(conno))
{
/* perform other work */
    YIELD();
}
SOCKET_NOBLOCK(conno);
for (;;)
{
    YIELD();
    len = Nread(conno, buf, sizeof(buf));
    if (len < 0)
    if (len != EWOULDBLOCK)
        break; /* error */
    else
    /* perform other work */
```

```
        else if (len == 0)
            break; /* other end closed */
        else
        {
        /* process message */
        }
    }
    stat = Nclose(conno);
    if (stat < 0) /* ERROR */
```

Appendix

Glossary

CHAP Challenge Handshake Authentication Protocol. A user and password authentication method used by a PPP connection. Both the user name and password are encrypted.

DHCP Dynamic Host Configuration Protocol. The protocol used by a host to request an IP address from a DHCP server based on the host's name.

DNS Dynamic Name Server. This is a machine which tells remote hosts what their names are, based on their IP addresses.

DPI Dynamic Protocol Interface. This is primary interface using stream I/O-like function calls.

FTP File Transport Protocol. FTP is used to transfer files using TCP connections through port 21 on an FTP server.

Passive Open A passive open means a host attempts to open a connection to any remote host wishing to establish a connection. The host will remain in the *Nopen()* function indefinitely until a connection is established.

TCP Transmission Control Protocol. TCP is a reliable protocol that insures data is actually received at the remote site.

TFTP Trivial File Transport Protocol. TFTP is used to transfer files via a UDP connection through port 69 on a TFTP server.

UDP User Datagram Protocol. UDP is a protocol designed to send data packets to the remote site without guaranteeing reception.