# *7*

# *Sockets under VxWorks*

### *7.1    Introduction*

This chapter describes how to use the standard BSD socket interface for stream sockets and datagram sockets on a VxWorks target. It also describes how to use zbuf sockets, an alternative set of socket calls based on a data abstraction called the *zbuf*. These zbuf calls let you share data buffers (or portions of data buffers) between separate software modules.

Using sockets, processes can communicate within a single CPU, across a backplane, across an Ethernet, or across any connected combination of networks. Socket communications can occur between VxWorks tasks and host system processes in any combination. In all cases, the communications appear identical to the application--except, of course, for the speed of the communications.

One of the biggest advantages of socket communication is that it is a homogeneous mechanism: socket communications among processes are the same, regardless of the location of the processes in the network or the operating system where they run. This is true even if you use zbuf sockets, which are fully interoperable with standard BSD sockets.

|  |  |
| --- | --- |
|  |  |
|  | **NOTE:** This chapter focuses on how to use a socket connection between processes. If you are interested in learning how to add socket-support code to a new network service or protocol, see [*11.4 Adding a Socket Interface to Your Service*, p.234](http://www.vxdev.com/docs/vx55man/vxworks/netguide/c-newNetService.html#84741). |
|  |  |
|  | |

For additional information on the socket interface, see the **sockLib** reference entry.

### *7.2    BSD Sockets*

A socket is a communications end-point that is *bound* to a UDP or TCP port within the node. Under VxWorks, your application can use the *sockets* interface to access features of the Internet Protocol suite (features such as multicasting). Depending on the bound port type, a socket is referred to either as a stream socket or a datagram socket. VxWorks sockets are UNIX BSD 4.4 compatible. However, VxWorks does not support signal functionality for VxWorks sockets.

S*tream sockets* use TCP to bind to a particular port number. Another process, on any host in the network, can then create another stream socket and request that it be connected to the first socket by specifying its host Internet address and port number. After the two TCP sockets are connected, there is a *virtual circuit* set up between them, allowing reliable socket-to-socket communications. This style of communication is conversational.

*Datagram socket*s use UDP to bind to a particular port number. Other processes, on any host in the network, can then send messages to that socket by specifying the host Internet address and the port number. Compared to TCP, UDP provides a simpler but less robust communication method. In a UDP communication, data is sent between sockets in separate, unconnected, individually addressed packets called *datagrams*. There is no sense of conversation with a datagram socket. The communication is in the style of a letter. Each packet carries the address of both the destination and the sender. Compared to TCP, UDP is unreliable. Like the mail, packets that are lost or out-of-sequence are not reported.

|  |  |
| --- | --- |
|  |  |
|  | **NOTE:** The complexities of socket programming are beyond the scope of this document. For additional information, consult a socket-programming book, such as those mentioned in the introduction to this manual. |
|  |  |
|  | |

#### *7.2.1   VxWorks-Specific Socket Dependencies*

Although the socket interface is compatible with VxWorks, the environment does affect how you use sockets. Specifically, the globally accessible file descriptors available in the task-independent address space of VxWorks require that you take extra precautions when closing a file descriptor.

You must make sure that one task does not close the file descriptor on which another task is pending during an **accept( )**. Although the **accept( )** on the closed file descriptor sometimes returns with an error, the **accept( )** can also fail to return at all. Thus, if you need to be able to close a socket connection's file descriptor asynchronously, you may need to set up a semaphore-based locking mechanism that prevents the close while an **accept( )** is pending on the file descriptor.

#### *7.2.2   Datagram Sockets (UDP)*

You can use datagram (UDP) sockets to implement a simple client-server communication system. You can also use UDP sockets to handle multicasting.

##### *Using a Datagram Socket to Implement a Client-Server Communication System*

The following code example uses a client-server communication model. The server communicates with clients using datagram-oriented (UDP) sockets. The main server loop, in **udpServer**( ), reads requests and optionally displays the client's message. The client builds the request by prompting the user for input. Note that this code assumes that it executes on machines that have the same data sizes and alignment.

Example 7-1 :   **Datagram Sockets (UDP)**

/\* udpExample.h - header used by both UDP server and client examples \*/   
   
#define SERVER\_PORT\_NUM 5002 /\* server's port number for bind() \*/   
#define REQUEST\_MSG\_SIZE 1024 /\* max size of request message \*/   
   
/\* structure used for client's request \*/   
   
struct request   
 {   
 int display; /\* TRUE = display message \*/   
 char message[REQUEST\_MSG\_SIZE]; /\* message buffer \*/   
 };

/\* udpClient.c - UDP client example \*/   
   
/\*   
DESCRIPTION   
This file contains the client-side of the vxWorks UDP example code.   
The example code demonstrates the usage of several BSD 4.4-style   
socket routine calls.   
\*/   
   
   
/\* includes \*/   
   
#include "vxWorks.h"   
#include "sockLib.h"   
#include "inetLib.h"   
#include "stdioLib.h"   
#include "strLib.h"   
#include "hostLib.h"   
#include "ioLib.h"   
#include "udpExample.h"   
   
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
\*   
\* udpClient - send a message to a server over a UDP socket   
\*   
\* This routine sends a user-provided message to a server over a UDP socket.   
\* Optionally, this routine can request that the server display the message.   
\* This routine may be invoked as follows:   
\* -> udpClient "remoteSystem"   
\* Message to send:   
\* Greetings from UDP client   
\* Would you like server to display your message (Y or N):   
\* y   
\* value = 0 = 0x0   
\*   
\* RETURNS: OK, or ERROR if the message could not be sent to the server.   
\*/   
   
STATUS udpClient   
 (   
 char \* serverName /\* name or IP address of server \*/   
 )   
 {   
 struct request myRequest; /\* request to send to server \*/   
 struct sockaddr\_in serverAddr; /\* server's socket address \*/   
 char display; /\* if TRUE, server prints message \*/   
 int sockAddrSize; /\* size of socket address structure \*/   
 int sFd; /\* socket file descriptor \*/   
 int mlen; /\* length of message \*/   
   
 /\* create client's socket \*/   
   
 if ((sFd = socket (AF\_INET, SOCK\_DGRAM, 0)) == ERROR)   
 {   
 perror ("socket");   
 return (ERROR);   
 }   
   
 /\* explicit bind not required - local port number is dynamic \*/   
   
 /\* build server socket address \*/   
   
 sockAddrSize = sizeof (struct sockaddr\_in);   
 bzero ((char \*) &serverAddr, sockAddrSize);   
 serverAddr.sin\_len = (u\_char) sockAddrSize;   
 serverAddr.sin\_family = AF\_INET;   
 serverAddr.sin\_port = htons (SERVER\_PORT\_NUM);   
   
 if (((serverAddr.sin\_addr.s\_addr = inet\_addr (serverName)) == ERROR) &&   
 ((serverAddr.sin\_addr.s\_addr = hostGetByName (serverName)) == ERROR))   
 {   
 perror ("unknown server name");   
 close (sFd);   
 return (ERROR);   
 }   
   
 /\* build request, prompting user for message \*/   
   
 printf ("Message to send: \n");   
 mlen = read (STD\_IN, myRequest.message, REQUEST\_MSG\_SIZE);   
 myRequest.message[mlen - 1] = '\0';   
   
 printf ("Would you like the server to display your message (Y or N): \n");   
 read (STD\_IN, &display, 1);   
 switch (display)   
 {   
 case 'y':   
 case 'Y': myRequest.display = TRUE;   
 break;   
 default: myRequest.display = FALSE;   
 break;   
 }   
   
 /\* send request to server \*/   
   
 if (sendto (sFd, (caddr\_t) &myRequest, sizeof (myRequest), 0,   
 (struct sockaddr \*) &serverAddr, sockAddrSize) == ERROR)   
 {   
 perror ("sendto");   
 close (sFd);   
 return (ERROR);   
 }   
   
 close (sFd);   
 return (OK);   
 }

/\* udpServer.c - UDP server example \*/   
   
/\*   
DESCRIPTION   
This file contains the server-side of the vxWorks UDP example code.   
The example code demonstrates the usage of several BSD 4.4-style   
socket routine calls.   
\*/   
   
/\* includes \*/   
#include "vxWorks.h"   
#include "sockLib.h"   
#include "inetLib.h"   
#include "stdioLib.h"   
#include "strLib.h"   
#include "ioLib.h"   
#include "fioLib.h"   
#include "udpExample.h"   
   
   
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
\*   
\* udpServer - read from UDP socket and display client's message if requested   
\*   
\* Example of vxWorks UDP server:   
\* -> sp udpServer   
\* task spawned: id = 0x3a1f6c, name = t2   
\* value = 3809132 = 0x3a1f6c   
\* -> MESSAGE FROM CLIENT (Internet Address 150.12.0.11, port 1028):   
\* Greetings from UDP client   
\*   
\* RETURNS: Never, or ERROR if a resources could not be allocated.   
\*/   
   
STATUS udpServer (void)   
 {   
 struct sockaddr\_in serverAddr; /\* server's socket address \*/   
 struct sockaddr\_in clientAddr; /\* client's socket address \*/   
 struct request clientRequest; /\* request/Message from client \*/   
 int sockAddrSize; /\* size of socket address structure \*/   
 int sFd; /\* socket file descriptor \*/   
 char inetAddr[INET\_ADDR\_LEN];   
 /\* buffer for client's inet addr \*/   
   
 /\* set up the local address \*/   
   
 sockAddrSize = sizeof (struct sockaddr\_in);   
 bzero ((char \*) &serverAddr, sockAddrSize);   
 serverAddr.sin\_len = (u\_char) sockAddrSize;   
 serverAddr.sin\_family = AF\_INET;   
 serverAddr.sin\_port = htons (SERVER\_PORT\_NUM);   
 serverAddr.sin\_addr.s\_addr = htonl (INADDR\_ANY);   
   
 /\* create a UDP-based socket \*/   
   
 if ((sFd = socket (AF\_INET, SOCK\_DGRAM, 0)) == ERROR)   
 {   
 perror ("socket");   
 return (ERROR);   
 }   
   
 /\* bind socket to local address \*/   
   
 if (bind (sFd, (struct sockaddr \*) &serverAddr, sockAddrSize) == ERROR)   
 {   
 perror ("bind");   
 close (sFd);   
 return (ERROR);   
 }   
   
 /\* read data from a socket and satisfy requests \*/   
   
 FOREVER   
 {   
 if (recvfrom (sFd, (char \*) &clientRequest, sizeof (clientRequest), 0,   
 (struct sockaddr \*) &clientAddr, &sockAddrSize) == ERROR)   
 {   
 perror ("recvfrom");   
 close (sFd);   
 return (ERROR);   
 }   
   
 /\* if client requested that message be displayed, print it \*/   
   
 if (clientRequest.display)   
 {   
 /\* convert inet address to dot notation \*/   
   
 inet\_ntoa\_b (clientAddr.sin\_addr, inetAddr);   
 printf ("MSG FROM CLIENT (Internet Address %s, port %d):\n%s\n",   
 inetAddr, ntohs (clientAddr.sin\_port), clientRequest.message);   
 }   
 }   
 }

#### *Using a Datagram (UDP) Socket to Access IP Multicasting*

Multicasting is the delivery of the same packets to multiple IP addresses. Typical multicasting applications include audio and video conferencing, resource discovery tools, and shared white boards. Multicasting is a feature of the IP layer, but to access this function, an application uses a UDP socket.

A VxWorks process must multicast on a network interface driver that supports multicasting (many do not). To review the capabilities of all attached network drivers, use **ifShow( )**. If a network interface supports multicasting, **IFF\_MULTICAST** is listed among the flags for that network interface.

Multicast IP addresses range from 224.0.0.0 to 239.255.255.255. These addresses are also called class D addresses or multicast groups. A datagram with a class D destination address is delivered to every process that has joined the corresponding multicast group.

To multicast a packet, a VxWorks process need do nothing special. The process just sends to the appropriate multicast address. The process can use any normal UDP socket. To set the route to the destination multicast address, use **mRouteAdd( )**.

To receive a multicast packet, a VxWorks process must join a multicast group. To do this, the VxWorks process must set the appropriate socket options on the socket (see [Table 7-1](http://www.vxdev.com/docs/vx55man/vxworks/netguide/c-sockets.html#86943)).

Table 7-1 :   **Multicasting Socket Options1**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | | | | | | | | | | |
| **Command** | **Argument** | **Description** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | | | | | | | | | | | | | | | | | | | |
| **IP\_MULTICAST\_IF** | **struct in\_addr** | Select default interface for outgoing multicasts. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **IP\_MULTICAST\_TTL** | **char** | Select default time to live (TTL) for outgoing multicast packets. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **IP\_MULTICAST\_LOOP** | **char** | Enable or disable loopback of outgoing multicasts. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **IP\_ADD\_MEMBERSHIP** | **struct ip\_mreq** | Join a multicast group. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **IP\_DROP\_MEMBERSHIP** | **struct ip\_mreq** | Leave a multicast group. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | | | | | | | | | | | | | | | | | | | |
| 1:  For more on multicasting socket options, see the **setsockopt( )** reference entry. | | | | | | | | | | | | | | | | | | | |

When choosing an address upon which to multicast, remember that certain addresses and address ranges are already registered to specific uses and protocols. For example, 244.0.0.1 multicasts to all systems on the local subnet. The Internet Assigned Numbers Authority (IANA) maintains a list of registered IP multicast groups. The current list can be found in RFC 1700. For more information about the IANA, see *RFC 1700*. [Table 7-2](http://www.vxdev.com/docs/vx55man/vxworks/netguide/c-sockets.html#86997) lists some of the well known multicast groups.

Table 7-2 :   **Well Known Multicast Groups**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | | | | | | | | | | |
| **Group** | **Constant** | **Description** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | | | | | | | | | | | | | | | | | | | |
| 224.0.0.0 | **INADDR\_UNSPEC\_GROUP** | Reserved for protocols that implement IP unicast and multicast routing mechanisms. Datagrams sent to any of these groups are not forwarded beyond the local network by multicast routers. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 224.0.0.1 | **INADDR\_ALLHOSTS\_GROUP** | All systems on this subnet. This value is automatically added to all network drivers at initialization. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 224.0.0.2 |  | All routers on this subnet. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 224.0.0.3 |  | Unassigned. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 224.0.0.4 |  | DVMRP routers. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 224.0.0.5 |  | OSPF routers. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 224.0.0.6 |  | OSPF designated routers. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 224.0.0.9 |  | All RIP routers. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 224.0.0.255 | **INADDR\_MAX\_LOCAL\_GROUP** | Unassigned. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 224.0.1.1 |  | NTP (Network Time Protocol). |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | | | | | | | | | | | | | | | | | | | |
|  | | | | | | | | | | | | | | | | | | | |

The following code samples define two routines, **mcastSend( )** and **mcastRcv( )**. These routines demonstrate how to use UDP sockets for sending and receiving multicast traffic.

**mcastSend( )** transmits a buffer to the specified multicast address. As input, this routine expects a multicast destination, a port number, a buffer pointer, and a buffer length. For example:

status = mcastSend ("224.1.0.1", 7777, bufPtr, 100);

**mcastRcv( )** receives any packet sent to a specified multicast address. As input, this routine expects the interface address from which the packet came, a multicast address, a port number, and the number of bytes to read from the packet. The returned value of the function is a pointer a buffer containing the read bytes. For example:

buf = mcastRcv (ifAddress, "224.1.0.1", 7777, 100) ;

Example 7-2 :   **Datagram Sockets (UDP) and Multicasting**

/\* includes \*/   
#include "vxWorks.h"   
#include "taskLib.h"   
#include "socket.h"   
#include "netinet/in.h"   
#include "stdio.h"   
#include "stdlib.h"   
#include "string.h"   
#include "sockLib.h"   
#include "inetLib.h"   
#include "ioLib.h"   
#include "routeLib.h"   
   
/\* defines \*/   
/\* globals \*/   
/\* forward declarations \*/   
   
   
STATUS mcastSend (char \* mcastAddr, USHORT mcastPort, char \* sendBuf,   
 int sendLen);   
char \* mcastRcv (char \* ifAddr, char \* mcastAddr, USHORT mcastPort,   
 int numRead);   
   
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
\* mcastSend - send a message to the multicast address   
\* This function sends a message to the multicast address   
\* The multicast group address to send, the port number, the pointer to the   
\* send buffer and the send buffer length are given as input parameters.   
\* RETURNS: OK if successful or ERROR   
\*/   
   
STATUS mcastSend   
 (   
 char \* mcastAddr, /\* multicast address \*/   
 USHORT mcastPort, /\* udp port number \*/   
 char \* sendBuf, /\* send Buffer \*/   
 int sendLen /\* length of send buffer \*/   
 )   
 {   
 struct sockaddr\_in sin;   
 struct sockaddr\_in toAddr;   
 int toAddrLen;   
 int sockDesc;   
 char \* bufPtr;   
 int len;   
   
 /\* create a send and recv socket \*/   
   
 if ((sockDesc = socket (AF\_INET, SOCK\_DGRAM, 0)) < 0 )   
 {   
 printf (" cannot open send socket\n");   
 return (ERROR);   
 }   
   
 /\* zero out the structures \*/   
 bzero ((char \*)&sin, sizeof (sin));   
 bzero ((char \*)&toAddr, sizeof (toAddr));   
   
 sin.sin\_len = (u\_char) sizeof(sin);   
 sin.sin\_family = AF\_INET;   
 sin.sin\_addr.s\_addr = INADDR\_ANY;   
 sin.sin\_port = htons(0);   
   
 if (bind(sockDesc, (struct sockaddr \*)&sin, sizeof(sin)) != 0)   
 {   
 perror("bind");   
 if (sockDesc)   
 close (sockDesc);   
 return (ERROR);   
 }   
   
 toAddrLen = sizeof(struct sockaddr\_in);   
 toAddr.sin\_len = (u\_char) toAddrLen;   
 toAddr.sin\_family = AF\_INET;   
   
   
 /\* initialize the address to the send \*/   
 toAddr.sin\_addr.s\_addr = inet\_addr (mcastAddr);   
   
 /\* initialize the port to send \*/   
 toAddr.sin\_port = htons(mcastPort);   
   
 bufPtr = sendBuf; /\* initialize the buffer pointer \*/   
   
 /\* send the buffer \*/   
 while (sendLen > 0)   
 {   
 if ((len = sendto (sockDesc, bufPtr, sendLen, 0,   
 (struct sockaddr \*)&toAddr, toAddrLen)) < 0 )   
 {   
 printf("mcastSend sendto errno:0x%x\n", errno );   
 break;   
 }   
   
 sendLen -= len;   
 bufPtr += len;   
   
 taskDelay (1); /\* give a taskDelay \*/   
 }   
   
 close (sockDesc);   
   
 return (OK);   
 }   
   
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
 \* mcastRcv - receive a message from a multicast address   
 \* This function receives a message from a multicast address   
 \* The interface address from which to receive the multicast packet,   
 \* the multicast address to recv from, the port number and the number of   
 \* bytes to read are given as input parameters to this routine.   
 \* RETURNS: Pointer to the Buffer or NULL if error.   
 \*/   
   
char \* mcastRcv   
 (   
 char \* ifAddr, /\* interface address to recv mcast packets \*/   
 char \* mcastAddr, /\* multicast address \*/   
 USHORT mcastPort, /\* udp port number to recv \*/   
 int numRead /\* number of bytes to read \*/   
 )   
 {   
 struct sockaddr\_in fromAddr;   
 struct sockaddr\_in sin;   
 int fromLen;   
 struct ip\_mreq ipMreq;   
 int recvLen;   
 int sockDesc;   
 char \* bufPtr;   
 int status = OK;   
 char \* recvBuf = NULL;   
   
 if ((sockDesc = socket (AF\_INET, SOCK\_DGRAM, 0)) < 0)   
 {   
 printf (" cannot open recv socket\n");   
 return (NULL);   
 }   
   
 bzero ((char \*)&sin, sizeof (sin));   
 bzero ((char \*) &fromAddr, sizeof(fromAddr));   
 fromLen = sizeof(fromAddr);   
   
 if ((recvBuf = calloc (numRead, sizeof (char))) == NULL)   
 {   
 printf (" calloc error, cannot allocate memory\n");   
 status = ERROR;   
 goto cleanUp;   
 }   
   
 sin.sin\_len = (u\_char) sizeof(sin);   
 sin.sin\_family = AF\_INET;   
 sin.sin\_addr.s\_addr = INADDR\_ANY;   
   
 /\* UDP port number to match for the received packets \*/   
 sin.sin\_port = htons (mcastPort);   
   
 /\* bind a port number to the socket \*/   
 if (bind(sockDesc, (struct sockaddr \*)&sin, sizeof(sin)) != 0)   
 {   
 perror("bind");   
 status = ERROR;   
 goto cleanUp;   
 }   
   
 /\* fill in the argument structure to join the multicast group \*/   
 /\* initialize the multicast address to join \*/   
   
 ipMreq.imr\_multiaddr.s\_addr = inet\_addr (mcastAddr);   
   
 /\* unicast interface addr from which to receive the multicast packets \*/   
 ipMreq.imr\_interface.s\_addr = inet\_addr (ifAddr);   
   
 /\* set the socket option to join the MULTICAST group \*/   
 if (setsockopt (sockDesc, IPPROTO\_IP, IP\_ADD\_MEMBERSHIP,   
 (char \*)&ipMreq,   
 sizeof (ipMreq)) < 0)   
 {   
 printf ("setsockopt IP\_ADD\_MEMBERSHIP error:\n");   
 status = ERROR;   
 goto cleanUp;   
 }   
   
 /\* get the data destined to the above multicast group \*/   
 bufPtr = recvBuf;   
   
   
   
 while (numRead > 0)   
 {   
 if ((recvLen = recvfrom (sockDesc, bufPtr, numRead, 0,   
 (struct sockaddr \*)&fromAddr, &fromLen)) < 0)   
 {   
 perror("recvfrom");   
 status = ERROR;   
 break;   
 }   
 numRead -= recvLen; /\* decrement number of bytes to read \*/   
 bufPtr += recvLen; /\* increment the buffer pointer \*/   
 }   
   
 /\* set the socket option to leave the MULTICAST group \*/   
 if (setsockopt (sockDesc, IPPROTO\_IP, IP\_DROP\_MEMBERSHIP,   
 (char \*)&ipMreq,   
 sizeof (ipMreq)) < 0)   
 printf ("setsockopt IP\_DROP\_MEMBERSHIP error:\n");   
   
 cleanUp:   
 close (sockDesc);   
 if ((status != OK) && (recvBuf != NULL))   
 {   
 free (recvBuf);   
 recvBuf = NULL;   
 }   
 return (recvBuf);   
 }

#### *7.2.3   Stream Sockets (TCP)*

The Transmission Control Protocol (TCP) provides reliable, two-way transmission of data. In a TCP communication, two sockets are *connected*, allowing a reliable byte-stream to flow between them in either direction. TCP is referred to as a *virtual circuit* protocol, because it behaves as though a circuit is created between the two sockets.

A good analogy for TCP communications is a telephone system. Connecting two sockets is similar to calling from one telephone to another. After the connection is established, you can write and read data (talk and listen).

[Table 7-3](http://www.vxdev.com/docs/vx55man/vxworks/netguide/c-sockets.html" \l "87090) shows the steps in establishing socket communications with TCP, and the analogy of each step with telephone communications.

Table 7-3 :   **TCP Analogy to Telephone Communication**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | | | | | | | | | | |
| **Task 1  Waits** | **Task 2  Calls** | **Function** | **Analogy** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | | | | | | | | | | | | | | | | | | | |
| **socket**( ) | **socket**( ) | Create sockets. | Hook up telephones. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **bind**( ) |  | Assign address to socket. | Assign telephone number. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **listen**( ) |  | Allow others to connect to socket. | Allow others to call. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | **connect**( ) | Request connection to another socket. | Dial another telephone 's number. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **accept**( ) |  | Complete connection between sockets. | Answer telephone and establish connection. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **write**( ) | **write**( ) | Send data to other socket. | Talk. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **read**( ) | **read**( ) | Receive data from other socket. | Listen. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **close**( ) | **close**( ) | Close sockets. | Hang up. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | | | | | | | | | | | | | | | | | | | |
|  | | | | | | | | | | | | | | | | | | | |

Example 7-3 :   **Stream Sockets (TCP)**

The following code example uses a client-server communication model. The server communicates with clients using stream-oriented (TCP) sockets. The main server loop, in **tcpServerWorkTask**( ), reads requests, prints the client's message to the console, and, if requested, sends a reply back to the client. The client builds the request by prompting for input. It sends a message to the server and, optionally, waits for a reply to be sent back. To simplify the example, we assume that the code is executed on machines that have the same data sizes and alignment.

/\* tcpExample.h - header used by both TCP server and client examples \*/   
   
/\* defines \*/   
#define SERVER\_PORT\_NUM 5001 /\* server's port number for bind() \*/   
#define SERVER\_WORK\_PRIORITY 100 /\* priority of server's work task \*/   
#define SERVER\_STACK\_SIZE 10000 /\* stack size of server's work task \*/   
#define SERVER\_MAX\_CONNECTIONS 4 /\* max clients connected at a time \*/   
#define REQUEST\_MSG\_SIZE 1024 /\* max size of request message \*/   
#define REPLY\_MSG\_SIZE 500 /\* max size of reply message \*/   
   
/\* structure for requests from clients to server \*/   
struct request   
 {   
 int reply; /\* TRUE = request reply from server \*/   
 int msgLen; /\* length of message text \*/   
 char message[REQUEST\_MSG\_SIZE]; /\* message buffer \*/   
 };

/\* tcpClient.c - TCP client example \*/   
   
/\*   
DESCRIPTION   
This file contains the client-side of the VxWorks TCP example code.   
The example code demonstrates the usage of several BSD 4.4-style   
socket routine calls.   
\*/   
   
/\* includes \*/   
   
#include "vxWorks.h"   
#include "sockLib.h"   
#include "inetLib.h"   
#include "stdioLib.h"   
#include "strLib.h"   
#include "hostLib.h"   
#include "ioLib.h"   
#include "tcpExample.h"   
   
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
\*   
\* tcpClient - send requests to server over a TCP socket   
\*   
\* This routine connects over a TCP socket to a server, and sends a   
\* user-provided message to the server. Optionally, this routine   
\* waits for the server's reply message.   
\*   
\* This routine may be invoked as follows:   
\* -> tcpClient "remoteSystem"   
\* Message to send:   
\* Hello out there   
\* Would you like a reply (Y or N):   
\* y   
\* value = 0 = 0x0   
\* -> MESSAGE FROM SERVER:   
\* Server received your message   
\*   
\* RETURNS: OK, or ERROR if the message could not be sent to the server.   
\*/   
   
STATUS tcpClient   
 (   
 char \* serverName /\* name or IP address of server \*/   
 )   
 {   
 struct request myRequest; /\* request to send to server \*/   
 struct sockaddr\_in serverAddr; /\* server's socket address \*/   
 char replyBuf[REPLY\_MSG\_SIZE]; /\* buffer for reply \*/   
 char reply; /\* if TRUE, expect reply back \*/   
 int sockAddrSize; /\* size of socket address structure \*/   
 int sFd; /\* socket file descriptor \*/   
 int mlen; /\* length of message \*/   
   
 /\* create client's socket \*/   
 if ((sFd = socket (AF\_INET, SOCK\_STREAM, 0)) == ERROR)   
 {   
 perror ("socket");   
 return (ERROR);   
 }   
   
 /\* bind not required - port number is dynamic \*/   
 /\* build server socket address \*/   
 sockAddrSize = sizeof (struct sockaddr\_in);   
 bzero ((char \*) &serverAddr, sockAddrSize);   
 serverAddr.sin\_family = AF\_INET;   
 serverAddr.sin\_len = (u\_char) sockAddrSize;   
 serverAddr.sin\_port = htons (SERVER\_PORT\_NUM);   
   
 if (((serverAddr.sin\_addr.s\_addr = inet\_addr (serverName)) == ERROR) &&   
 ((serverAddr.sin\_addr.s\_addr = hostGetByName (serverName)) == ERROR))   
 {   
 perror ("unknown server name");   
 close (sFd);   
 return (ERROR);   
 }   
   
 /\* connect to server \*/   
 if (connect (sFd, (struct sockaddr \*) &serverAddr, sockAddrSize) == ERROR)   
 {   
 perror ("connect");   
 close (sFd);   
 return (ERROR);   
 }   
   
 /\* build request, prompting user for message \*/   
 printf ("Message to send: \n");   
 mlen = read (STD\_IN, myRequest.message, REQUEST\_MSG\_SIZE);   
 myRequest.msgLen = mlen;   
 myRequest.message[mlen - 1] = '\0';   
 printf ("Would you like a reply (Y or N): \n");   
 read (STD\_IN, &reply, 1);   
 switch (reply)   
 {   
 case 'y':   
 case 'Y': myRequest.reply = TRUE;   
 break;   
 default: myRequest.reply = FALSE;   
 break;   
 }   
   
 /\* send request to server \*/   
   
 if (write (sFd, (char \*) &myRequest, sizeof (myRequest)) == ERROR)   
 {   
 perror ("write");   
 close (sFd);   
 return (ERROR);   
 }   
   
 if (myRequest.reply) /\* if expecting reply, read and display it \*/   
 {   
 if (read (sFd, replyBuf, REPLY\_MSG\_SIZE) < 0)   
 {   
 perror ("read");   
 close (sFd);   
 return (ERROR);   
 }

printf ("MESSAGE FROM SERVER:\n%s\n", replyBuf);   
 }

close (sFd);   
 return (OK);   
 }

/\* tcpServer.c - TCP server example \*/

/\*   
DESCRIPTION   
This file contains the server-side of the VxWorks TCP example code.   
The example code demonstrates the usage of several BSD 4.4-style   
socket routine calls.   
\*/

/\* includes \*/   
#include "vxWorks.h"   
#include "sockLib.h"   
#include "inetLib.h"   
#include "taskLib.h"   
#include "stdioLib.h"   
#include "strLib.h"   
#include "ioLib.h"   
#include "fioLib.h"   
#include "tcpExample.h"

/\* function declarations \*/

VOID tcpServerWorkTask (int sFd, char \* address, u\_short port);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
\*   
\* tcpServer - accept and process requests over a TCP socket   
\*   
\* This routine creates a TCP socket, and accepts connections over the socket   
\* from clients. Each client connection is handled by spawning a separate   
\* task to handle client requests.   
\*   
\* This routine may be invoked as follows:   
\* -> sp tcpServer   
\* task spawned: id = 0x3a6f1c, name = t1   
\* value = 3829532 = 0x3a6f1c   
\* -> MESSAGE FROM CLIENT (Internet Address 150.12.0.10, port 1027):   
\* Hello out there   
\*   
\* RETURNS: Never, or ERROR if a resources could not be allocated.   
\*/

STATUS tcpServer (void)   
 {   
 struct sockaddr\_in serverAddr; /\* server's socket address \*/   
 struct sockaddr\_in clientAddr; /\* client's socket address \*/   
 int sockAddrSize; /\* size of socket address structure \*/   
 int sFd; /\* socket file descriptor \*/   
 int newFd; /\* socket descriptor from accept \*/   
 int ix = 0; /\* counter for work task names \*/   
 char workName[16]; /\* name of work task \*/

/\* set up the local address \*/

sockAddrSize = sizeof (struct sockaddr\_in);   
 bzero ((char \*) &serverAddr, sockAddrSize);   
 serverAddr.sin\_family = AF\_INET;   
 serverAddr.sin\_len = (u\_char) sockAddrSize;   
 serverAddr.sin\_port = htons (SERVER\_PORT\_NUM);   
 serverAddr.sin\_addr.s\_addr = htonl (INADDR\_ANY);

/\* create a TCP-based socket \*/

if ((sFd = socket (AF\_INET, SOCK\_STREAM, 0)) == ERROR)   
 {   
 perror ("socket");   
 return (ERROR);   
 }

/\* bind socket to local address \*/

if (bind (sFd, (struct sockaddr \*) &serverAddr, sockAddrSize) == ERROR)   
 {   
 perror ("bind");   
 close (sFd);   
 return (ERROR);   
 }

/\* create queue for client connection requests \*/

if (listen (sFd, SERVER\_MAX\_CONNECTIONS) == ERROR)   
 {   
 perror ("listen");   
 close (sFd);   
 return (ERROR);   
 }

/\* accept new connect requests and spawn tasks to process them \*/

FOREVER   
 {   
 if ((newFd = accept (sFd, (struct sockaddr \*) &clientAddr,   
 &sockAddrSize)) == ERROR)   
 {   
 perror ("accept");   
 close (sFd);   
 return (ERROR);   
 }

sprintf (workName, "tTcpWork%d", ix++);   
 if (taskSpawn(workName, SERVER\_WORK\_PRIORITY, 0, SERVER\_STACK\_SIZE,   
 (FUNCPTR) tcpServerWorkTask, newFd,   
 (int) inet\_ntoa (clientAddr.sin\_addr), ntohs (clientAddr.sin\_port),   
 0, 0, 0, 0, 0, 0, 0) == ERROR)   
 {   
 /\* if taskSpawn fails, close fd and return to top of loop \*/   
   
 perror ("taskSpawn");   
 close (newFd);   
 }   
 }   
 }   
   
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
\*   
\* tcpServerWorkTask - process client requests   
\*   
\* This routine reads from the server's socket, and processes client   
\* requests. If the client requests a reply message, this routine   
\* will send a reply to the client.   
\*   
\* RETURNS: N/A.   
\*/   
   
VOID tcpServerWorkTask   
 (   
 int sFd, /\* server's socket fd \*/   
 char \* address, /\* client's socket address \*/   
 u\_short port /\* client's socket port \*/   
 )   
 {   
 struct request clientRequest; /\* request/message from client \*/   
 int nRead; /\* number of bytes read \*/   
 static char replyMsg[] = "Server received your message";   
   
 /\* read client request, display message \*/   
   
 while ((nRead = fioRead (sFd, (char \*) &clientRequest,   
 sizeof (clientRequest))) > 0)   
 {   
 printf ("MESSAGE FROM CLIENT (Internet Address %s, port %d):\n%s\n",   
 address, port, clientRequest.message);   
   
 free (address); /\* free malloc from inet\_ntoa() \*/   
   
 if (clientRequest.reply)   
 if (write (sFd, replyMsg, sizeof (replyMsg)) == ERROR)   
 perror ("write");   
 }   
   
 if (nRead == ERROR) /\* error from read() \*/   
 perror ("read");   
   
 close (sFd); /\* close server socket connection \*/   
 }

### *7.3    Zbuf Sockets*

VxWorks includes an alternative set of socket calls based on a data abstraction called a *zbuf*, a zero-copy buffer. Using the zbuf socket interface, applications can read and write UNIX BSD sockets without copying data between application buffers and network buffers. You can use zbufs with either UDP or TCP applications. The TCP subset of the zbuf interface is sometimes called *zero-copy TCP*.

Zbuf-based socket calls are *interoperable* with the standard BSD socket interface: the other end of a socket has no way of telling whether your end is using zbuf-based calls or traditional calls. However, zbuf-based socket calls are *not source-compatible* with the standard BSD socket interface: you must call different socket functions to use the zbuf interface. Applications that use the zbuf interface are thus less portable.

|  |  |
| --- | --- |
|  |  |
|  | **WARNING:** The send socket buffer size must exceed that of any zbufs sent over the socket. To set the send socket buffer size, use either the **TCP\_SND\_SIZE\_DFLT** or **UDP\_SND\_SIZE\_DFLT** configuration parameter. |
|  |  |
|  | |

To include zbuf functionality in your image, use the **INCLUDE\_ZBUF\_SOCK** configuration parameter.

#### *7.3.1   Zbuf Sockets and Protection Domains*

If you are using zbufs within the VxWorks AE protection domain model, you must do so within the kernel domain.

#### *7.3.2   Zbuf Calls to Send Existing Data Buffers*

The simplest way to use zbuf sockets is to call either **zbufSockBufSend( )** (in place of **send( )** for a TCP connection) or **zbufSockBufSendto( )** (in place of **sendto( )** for a UDP datagram). In either case, you supply a pointer to your application's data buffer containing the data or message to send, and the network protocol uses that same buffer rather than copying the data out of it.

|  |  |
| --- | --- |
|  |  |
|  | **WARNING:** Using zbufs allows different modules to share the same buffers. This lets your application avoid the performance hit associated with copying the buffer. To make this work, your application must not modify (let alone free!) the data buffer while network software is still using it. Instead of freeing your buffer explicitly, you can supply a free-routine callback: a pointer to a routine that knows how to free the buffer. The zbuf library keeps track of how many zbufs point to a data buffer and calls the free routine when the data buffer is no longer in use. |
|  |  |
|  | |

To receive socket data using zbufs, see the following sections. [*7.3.3 Manipulating the Zbuf Data Structure*, p.143](http://www.vxdev.com/docs/vx55man/vxworks/netguide/c-sockets.html#87247) describes the routines to create and manage zbufs, and [*7.3.4 Zbuf Socket Calls*, p.152](http://www.vxdev.com/docs/vx55man/vxworks/netguide/c-sockets.html#87537) introduces the remaining zbuf-specific socket routines. See also the reference entries for **zbufLib** and **zbufSockLib**.

#### *7.3.3   Manipulating the Zbuf Data Structure*

A zbuf has three essential properties:

* A zbuf holds a sequence of bytes.
* The data in a zbuf is organized into one or more *segments* of contiguous data. Successive zbuf segments are not usually contiguous to each other.
* Zbuf segments refer to data buffers through pointers. The underlying data buffers can be shared by more than one zbuf segment.

Zbuf segments are at the heart of how zbufs minimize data copying; if you have a data buffer, you can incorporate it (by reference, so that only pointers and lengths move around) into a new zbuf segment. Conversely, you can get pointers to the data in zbuf segments, and examine the data there directly.

#### *Zbuf Byte Locations*

You can address the contents of a zbuf by *byte locations*. A zbuf byte location has two parts, an *offset* and a *segment ID*.

An *offset* is a signed integer (type **int**): the distance in bytes to a portion of data in the zbuf, relative to the beginning of a particular segment. Zero refers to the first byte in a segment; negative integers refer to bytes in previous segments; and positive integers refer to bytes after the start of the current segment.

A *segment ID* is an arbitrary integer (type **ZBUF\_SEG**) that identifies a particular segment of a zbuf. You can always use **NULL** to refer to the first segment of a zbuf.

[Figure 7-1](http://www.vxdev.com/docs/vx55man/vxworks/netguide/c-sockets.html" \l "87268) shows a simple zbuf with data organized into two segments. The offsets are relative to the first segment. This is the most efficient addressing scheme to use to refer to bytes a, b, or c in the figure.

Figure 7-1 :   **Zbuf Addressing Relative to First Segment (NULL)**

[Figure 7-2](http://www.vxdev.com/docs/vx55man/vxworks/netguide/c-sockets.html" \l "87305) shows the same zbuf, but it is labeled with offsets relative to the second segment. This is the most efficient addressing scheme to refer to bytes d, e, f, or g in the figure.

Figure 7-2 :   **Zbuf Addressing Relative to Second Segment**

Two special shortcuts give the fastest access to either the beginning or the end of a zbuf. The constant **ZBUF\_END** refers to the position after all existing bytes in the zbuf. Similarly, **ZBUF\_BEGIN** refers to the position before all existing bytes. These constants are the only offsets with meanings not relative to a particular segment.

When you insert data in a zbuf, the new data is always inserted *before* the byte location you specify in the call to an insertion routine. That is, the byte location you specify becomes the address of the newly inserted data.

#### *Creating and Destroying Zbufs*

To create a new zbuf, call **zbufCreate( )**. The routine takes no arguments, and returns a zbuf identifier (type **ZBUF\_ID**) for a zbuf containing no segments. After you have the zbuf ID, you can attach segments or otherwise insert data. While the zbuf is empty, **NULL** is the only valid segment ID, and 0 the only valid offset.

When you no longer need a particular zbuf, call **zbufDelete( )**. Its single argument is the ID for the zbuf to delete. The **zbufDelete( )** routine calls the free routine associated with each segment in the zbuf, for segments that are not shared by other zbufs. After you delete a zbuf, its zbuf ID is meaningless; any reference to a deleted zbuf ID is an error.

Table 7-4 :   **Zbuf Creation and Deletion Routines**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | | | | | | | | | | |
| **Call** | **Description** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | | | | | | | | | | | | | | | | | | | |
| **zbufCreate( )** | Create an empty zbuf. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **zbufDelete( )** | Delete a zbuf and free any associated segments. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | | | | | | | | | | | | | | | | | | | |
|  | | | | | | | | | | | | | | | | | | | |

#### *Getting Data In and Out of Zbufs*

The usual way to place data in a zbuf is to call **zbufInsertBuf( )**. This routine builds a zbuf segment pointing to an existing data buffer, and inserts the new segment at whatever byte location you specify in a zbuf. You can also supply a callback pointer to a free routine, which the zbuf library calls when no zbuf segments point to that data buffer.

Because the purpose of the zbuf socket interface is to avoid data copying, the need to actually copy data into a zbuf (rather than designating its location as a shareable buffer) occurs much less frequently. When that need does arise, however, the routine **zbufInsertCopy( )** is available. This routine does not require a callback pointer to a free routine, because the original source of the data is not shared.

Similarly, the most efficient way to examine data in zbufs is to read it in place, rather than to copy it to another location. However, if you must copy some of the data out of a zbuf (for example, to guarantee the data is contiguous, or to place it in a data structure required by another interface), call **zbufExtractCopy( )**. Within the call, specify what to copy (zbuf ID, byte location, and the number of bytes) and where to put it (an application buffer).

Table 7-5 :   **Zbuf Data Copying Routines**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | | | | | | | | | | |
| **Call** | **Description** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | | | | | | | | | | | | | | | | | | | |
| **zbufInsertBuf( )** | Create a zbuf segment from a buffer and insert into a zbuf. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **zbufInsertCopy( )** | Copy buffer data into a zbuf. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **zbufExtractCopy( )** | Copy data from a zbuf to a buffer. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | | | | | | | | | | | | | | | | | | | |
|  | | | | | | | | | | | | | | | | | | | |

#### *Operations on Zbufs*

The routines listed in [Table 7-6](http://www.vxdev.com/docs/vx55man/vxworks/netguide/c-sockets.html#87397) perform several fundamental operations on zbufs.

Table 7-6 :   **Zbuf Operations**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | | | | | | | | | | |
| **Call** | **Description** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | | | | | | | | | | | | | | | | | | | |
| **zbufLength( )** | Determine the length of a zbuf, in bytes. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **zbufDup( )** | Duplicate a zbuf. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **zbufInsert( )** | Insert a zbuf into another zbuf. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **zbufSplit( )** | Split a zbuf into two separate zbufs. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **zbufCut( )** | Delete bytes from a zbuf. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | | | | | | | | | | | | | | | | | | | |
|  | | | | | | | | | | | | | | | | | | | |

The routine **zbufLength( )** reports how many bytes are in a zbuf.

The routine **zbufDup( )** provides the simplest mechanism for sharing segments between zbufs: it produces a new zbuf ID that refers to some or all of the data in the original zbuf. You can exploit this sort of sharing to get two different views of the same data. For example, after duplicating a zbuf, you can insert another zbuf into one of the two duplicates, with **zbufInsert( )**. None of the data in the original zbuf segments moves, yet after some byte location (the byte location where you inserted data) addressing the two zbufs gives completely different data.

The **zbufSplit( )** routine divides one zbuf into two; you specify the byte location for the split, and the result of the routine is a new zbuf ID. The new zbuf's data begins after the specified byte location. The original zbuf ID also has a modified view of the data: it is truncated to the byte location of the split. However, none of the data in the underlying segments moves through all this. If you duplicate the original zbuf before splitting it, three zbuf IDs share segments. The duplicate permits you to view the entire original range of data, another zbuf contains a leading fragment, and the third zbuf holds the trailing fragment.

Similarly, if you call **zbufCut( )** to remove some range of bytes from within a zbuf, the effects are visible only to callers who view the data through the same zbuf ID you used for the deletion. Other zbuf segments can still address the original data through a shared buffer.

For the most part, these routines do not free data buffers or delete zbufs, but there are two exceptions:

* **zbufInsert( )** deletes the zbuf ID it inserts. No segments are freed, because they now form part of the larger zbuf.
* If the bytes you remove with **zbufCut( )** span one or more complete segments, the free routines for those segments can be called (if no other zbuf segment refers to the same data).

The data-buffer free routine runs only when *none* of the data in a segment is part of any zbuf. To avoid data copying, zbuf manipulation routines such as **zbufCut( )** record which parts of a segment are currently in a zbuf, postponing the deletion of a segment until no part of its data is in use.

#### *Segments of Zbufs*

The routines in [Table 7-7](http://www.vxdev.com/docs/vx55man/vxworks/netguide/c-sockets.html#87449) give your applications access to the underlying segments in a zbuf.

Table 7-7 :   **Zbuf Segment Routines**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | | | | | | | | | | |
| **Call** | **Description** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | | | | | | | | | | | | | | | | | | | |
| **zbufSegFind( )** | Find the zbuf segment containing a specified byte location. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **zbufSegNext( )** | Get the next segment in a zbuf. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **zbufSegPrev( )** | Get the previous segment in a zbuf. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **zbufSegData( )** | Determine the location of data in a zbuf segment. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **zbufSegLength( )** | Determine the length of a zbuf segment. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | | | | | | | | | | | | | | | | | | | |
|  | | | | | | | | | | | | | | | | | | | |

By specifying a **NULL** segment ID, you can address the entire contents of a zbuf as offsets from its very first data byte. However, it is always more efficient to address data in a zbuf relative to the closest segment. Use **zbufSegFind( )** to translate any zbuf byte location into the most local form.

The pair **zbufSegNext( )** and **zbufSegPrev( )** are useful for going through the segments of a zbuf in order, perhaps in conjunction with **zbufSegLength( )**.

Finally, **zbufSegData( )** allows the most direct access to the data in zbufs: it gives your application the address where a segment's data begins. If you manage segment data directly using this pointer, bear the following restrictions in mind:

* Do not change data if any other zbuf segment is sharing it.
* As with any other direct memory access, it is up to your own code to restrict itself to meaningful data: remember that the next segment in a zbuf is usually not contiguous. Use **zbufSegLength( )** as a limit, and **zbufSegNext( )** when you exceed that limit.

#### *Example: Manipulating Zbuf Structure*

The following interaction illustrates the use of some of the previously described **zbufLib** routines, and their effect on zbuf segments and data sharing. To keep the example manageable, the zbuf data used is artificially small, and the execution environment is the Tornado shell (for details on this shell, see the *Tornado User's Guide: Shell*).

To begin with, we create a zbuf, and use its ID **zId** to verify that a newly created zbuf contains no data; **zbufLength( )** returns a result of 0.

-> **zId = zbufCreate()**new symbol "zId" added to symbol table.   
zId = 0x3b58e8: value = 3886816 = 0x3b4ee0   
-> **zbufLength (zId)**value = 0 = 0x0

Next, we create a data buffer **buf1**, insert it into zbuf **zId**, and verify that **zbufLength( )** now reports a positive length. To keep the example simple, **buf1** is a literal string, and therefore we do not supply a free-routine callback argument to **zbufInsertBuf( )**.

-> **buf1 = "I cannot repeat enough!"**new symbol "buf1" added to symbol table.   
buf1 = 0x3b5898: value = 3889320 = 0x3b58a8 = buf1 + 0x10   
-> **zbufInsertBuf (zId, 0, 0, buf1, strlen(buf1), 0, 0)**value = 3850240 = 0x3ac000   
-> **zbufLength (zId)**value = 23 = 0x17

To examine the effect of other zbuf operations, it is useful to have a zbuf-display routine. The remainder of this example uses a routine called **zbufDisplay( )** for that purpose; for the complete source code, see [Example 7-4](http://www.vxdev.com/docs/vx55man/vxworks/netguide/c-sockets.html#87515).

For each zbuf segment, **zbufDisplay( )** shows the segment ID, the start-of-data address, the offset from that address, the length of the segment, and the data in the segment as a character string. The following display of **zId** illustrates that the underlying data in its only segment is still at the **buf1** address (0x3b58a8), because **zbufInsertBuf( )** incorporates its buffer argument into the zbuf without copying data.

-> **ld </usr/jane/zbuf-examples/zbufDisplay.o**value = 3890416 = 0x3b5cf0 = zbufDisplay.o\_bss + 0x8   
-> **zbufDisplay zId**segID 0x3ac000 at 0x3b58a8 + 0x0 (23 bytes): I cannot repeat enough!   
value = 0 = 0x0

When we copy the zbuf, the copy has its own IDs, but still uses the same data address:

-> **zId2 = zbufDup (zId,0,0,23)**new symbol "zId2" added to symbol table.   
zId2 = 0x3b5ff0: value = 3886824 = 0x3b4ee8   
-> **zbufDisplay zId2**segID 0x3abf80 at 0x3b58a8 + 0x0 (23 bytes): I cannot repeat enough!   
value = 0 = 0x0

If we insert a second buffer into the middle of the existing data in **zId**, there is still no data copying. Inserting the new buffer gives us a zbuf made up of three segments--but notice that the address of the first segment is still the start of **buf1**, and the third segment points into the middle of **buf1**:

-> **buf2 = " this"**new symbol "buf2" added to symbol table.   
buf2 = 0x3b5fb0: value = 3891136 = 0x3b5fc0 = buf2 + 0x10   
-> **zbufInsertBuf (zId, 0, 15, buf2, strlen(buf2), 0, 0)**value = 3849984 = 0x3abf00   
-> **zbufDisplay zId**segID 0x3ac000 at 0x3b58a8 + 0x0 (15 bytes): I cannot repeat   
segID 0x3abf00 at 0x3b5fc0 + 0x0 ( 5 bytes): this   
segID 0x3abe80 at 0x3b58b7 + 0x0 ( 8 bytes): enough!   
value = 0 = 0x0

Because the underlying buffer is not modified, both **buf1** and the duplicate zbuf **zId2** still contain the original string, rather than the modified one now in **zId**:

-> **printf ("%s\n", buf1)**I cannot repeat enough!   
value = 24 = 0x18   
-> **zbufDisplay zId2**segID 0x3abf80 at 0x3b58a8 + 0x0 (23 bytes): I cannot repeat enough!   
value = 0 = 0x0

The **zbufDup( )** routine can also select part of a zbuf without copying, for instance to incorporate some of the same data into another zbuf--or even into the same zbuf, as in the following example:

-> **zTmp = zbufDup (zId, 0, 15, 5)**new symbol "zTmp" added to symbol table.   
zTmp = 0x3b5f70: value = 3886832 = 0x3b4ef0

-> **zbufInsert (zId, 0, 15, zTmp)**value = 3849728 = 0x3abe00   
-> **zbufDisplay zId**segID 0x3ac000 at 0x3b58a8 + 0x0 (15 bytes): I cannot repeat   
segID 0x3abe00 at 0x3b5fc0 + 0x0 ( 5 bytes): this   
segID 0x3abf00 at 0x3b5fc0 + 0x0 ( 5 bytes): this   
segID 0x3abe80 at 0x3b58b7 + 0x0 ( 8 bytes): enough!   
value = 0 = 0x0

After **zbufInsert( )** combines two zbufs, the second zbuf ID (**zTmp** in this example) is automatically deleted. Thus, **zTmp** is no longer a valid zbuf ID--for example, **zbufLength( )** returns ERROR:

-> **zbufLength (zTmp)**value = -1 = 0xffffffff = zId2 + 0xffc4a00f

However, you must still delete the remaining two zbuf IDs explicitly when they are no longer needed. This releases all associated zbuf-structure storage. In a real application, with free-routine callbacks filled in, it also calls the specified free routine on the data buffers, as follows:

-> **zbufDelete (zId)**value = 0 = 0x0   
-> **zbufDelete (zId2)**value = 0 = 0x0

Example 7-4 :   **Zbuf Display Routine**

The following is the complete source code for the **zbufDisplay( )** utility used in the preceding example:

/\* zbufDisplay.c - zbuf example display routine \*/   
   
/\* includes \*/   
   
#include "vxWorks.h"   
#include "zbufLib.h"   
#include "ioLib.h"   
#include "stdio.h"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
\*   
\* zbufDisplay - display contents of a zbuf   
\*   
\* RETURNS: OK, or ERROR if the specified data could not be displayed.   
\*/

STATUS zbufDisplay   
 (   
 ZBUF\_ID zbufId, /\* zbuf to display \*/   
 ZBUF\_SEG zbufSeg, /\* zbuf segment base for <offset> \*/   
 int offset, /\* relative byte offset \*/   
 int len, /\* number of bytes to display \*/   
 BOOL silent /\* do not print out debug info \*/   
 )   
 {   
 int lenData;   
 char \* pData;

/\* find the most-local byte location \*/

if ((zbufSeg = zbufSegFind (zbufId, zbufSeg, &offset)) == NULL)   
 return (ERROR);

if (len <= 0)   
 len = ZBUF\_END;

while ((len != 0) && (zbufSeg != NULL))   
 {   
 /\* find location and data length of zbuf segment \*/

pData = zbufSegData (zbufId, zbufSeg) + offset;   
 lenData = zbufSegLength (zbufId, zbufSeg) - offset;   
 lenData = min (len, lenData); /\* print all of seg ? \*/

if (!silent)   
 printf ("segID 0x%x at 0x%x + 0x%x (%2d bytes): ",   
 (int) zbufSeg, (int) pData, offset, lenData);   
 write (STD\_OUT, pData, lenData); /\* display data \*/   
 if (!silent)   
 printf ("\n");

zbufSeg = zbufSegNext (zbufId, zbufSeg); /\* update segment \*/   
 len -= lenData; /\* update length \*/   
 offset = 0; /\* no more offset \*/   
 }

return (OK);   
 }

#### *Limitations of the Zbuf Implementation*

The following zbuf limitations are due to the current implementation; they are not inherent to the data abstraction. They are described because they can have an impact on application performance.

* With the current implementation, references to data in zbuf segments before a particular location (whether with **zbufSegPrev( )**, or with a negative offset in a byte location) are significantly slower than references to data after a particular location.
* The data in small zbuf segments (less than 512 bytes) is sometimes copied, rather than having references propagated to it.

#### *7.3.4   Zbuf Socket Calls*

The zbuf socket calls listed in [Table 7-8](http://www.vxdev.com/docs/vx55man/vxworks/netguide/c-sockets.html#87546) are named to emphasize parallels with the standard BSD socket calls: thus, **zbufSockSend( )** is the zbuf version of **send( )**, and **zbufSockRecvfrom( )** is the zbuf version of **recvfrom( )**. The arguments also correspond directly to those of the standard socket calls.

Table 7-8 :   **Zbuf Socket Library Routines**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | | | | | | | | | | |
| **Call** | **Description** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | | | | | | | | | | | | | | | | | | | |
| **zbufSockLibInit( )** | Initialize socket libraries (called automatically if the configuration has zbuf sockets enabled. The relevant configuration parameter is **INCLUDE\_SOCK\_ZBUF**). |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **zbufSockSend( )** | Send zbuf data through a TCP socket. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **zbufSockSendto( )** | Send a zbuf message through a UDP socket. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **zbufSockBufSend( )** | Create a zbuf and send it as TCP socket data. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **zbufSockBufSendto( )** | Create a zbuf and send it as a UDP socket message. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **zbufSockRecv( )** | Receive data in a zbuf from a TCP socket. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **zbufSockRecvfrom( )** | Receive a message in a zbuf from a UDP socket. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | | | | | | | | | | | | | | | | | | | |
|  | | | | | | | | | | | | | | | | | | | |

For a detailed description of each routine, see the corresponding reference entry.

#### *Standard Socket Calls and Zbuf Socket Calls*

The zbuf socket calls are particularly useful when large data transfer is a significant part of your socket application. For example, many socket applications contain sections of code like the following fragment:

pBuffer = malloc (BUFLEN);   
while ((readLen = read (fdDevice, pBuffer, BUFLEN)) > 0)   
 write (fdSock, pBuffer, readLen);

You can eliminate the overhead of copying from the application buffer **pBuffer** into the internal socket buffers by changing the code to use zbuf socket calls. For example, the following fragment is a zbuf version of the preceding loop:

pBuffer = malloc (BUFLEN \* BUFNUM); /\* allocate memory \*/   
for (ix = 0; ix < (BUFNUM - 1); ix++, pBuffer += BUFLEN)   
 appBufRetn (pBuffer); /\* fill list of free bufs \*/   
   
while ((readLen = read (fdDevice, pBuffer, BUFLEN)) > 0)   
 {   
 zId = zbufCreate (); /\* insert into new zbuf \*/   
 zbufInsertBuf (zId, NULL, 0, pBuffer, readLen, appBufRetn, 0);   
 zbufSockSend (fdSock, zId, readLen, 0); /\* send zbuf \*/   
 pBuffer = appBufGet (WAIT\_FOREVER); /\* get a fresh buffer \*/   
 }

The **appBufGet( )** and **appBufRetn( )** references in the preceding code fragment stand for application-specific buffer management routines, analogous to **malloc( )** and **free( )**. In many applications, these routines do nothing more than manipulate a linked list of free fixed-length buffers.

Example 7-5 :   **The TCP Example Server Using Zbufs**

For a small but complete example that illustrates the mechanics of using the zbuf socket library, consider the conversion of the client-server example in [Example 7-3](http://www.vxdev.com/docs/vx55man/vxworks/netguide/c-sockets.html#87170) to use zbuf socket calls.

No conversion is needed for the client side of the example; the client operates the same regardless of whether or not the server uses zbufs. The next example illustrates the following changes to convert the server side to use zbufs:

* Instead of including the header file **sockLib.h**, include **zbufSockLib.h**.
* The data processing component must be capable of dealing with potentially non-contiguous data in successive zbuf segments. In the TCP example, this component displays a message using **printf( )**; we can use the **zbufDisplay( )** routine from [Example 7-4](http://www.vxdev.com/docs/vx55man/vxworks/netguide/c-sockets.html#87515) instead.
* The original TCP example exploits **fioRead( )** to collect the complete message, rather than calling **recv( )** directly. To achieve the same end while avoiding data copying by using zbufs, the following example defines a **zbufFioSockRecv( )** subroutine to call **zbufSockRecv( )** repeatedly until the complete message is received.
* A new version of the worker routine **tcpServerWorkTask( )** must tie together these separate modifications, and must explicitly extract the **reply** and **msgLen** fields from the client's transmission to do so. When using zbufs, these fields cannot be extracted by reference to the C structure in **tcpExample.h** because of the possibility that the data is not contiguous.

The following example shows the auxiliary **zbufFioSockRecv( )** routine and the zbuf version of **tcpServerWorkTask( )**. To run this code:

1. Start with **tcpServer.c** as defined in [Example 7-3](http://www.vxdev.com/docs/vx55man/vxworks/netguide/c-sockets.html#87170).
2. Include the header file **zbufSockLib.h**.
3. Insert the **zbufDisplay( )** routine from [Example 7-4](http://www.vxdev.com/docs/vx55man/vxworks/netguide/c-sockets.html#87515).
4. Replace the **tcpServerWorkTask( )** definition with the following two routines:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
\*   
\* zbufFioSockRecv - receive <len> bytes from a socket into a zbuf   
\*   
\* This routine receives a specified amount of data from a socket into a   
\* zbuf, by repeatedly calling zbufSockRecv() until <len> bytes   
\* are read.   
\*   
\* RETURNS:   
\* The ID of the zbuf containing <len> bytes of data,   
\* or NULL if there is an error during the zbufSockRecv() operation.   
\*   
\* SEE ALSO: zbufSockRecv()   
\*/

ZBUF\_ID zbufFioSockRecv   
 (   
 int fd, /\* file descriptor of file to read \*/   
 int len /\* maximum number of bytes to read \*/   
 )   
 {   
 BOOL first = TRUE; /\* first time thru ? \*/   
 ZBUF\_ID zRecvTotal = NULL; /\* zbuf to return \*/   
 ZBUF\_ID zRecv; /\* zbuf read from sock \*/   
 int nbytes; /\* number of recv bytes \*/

for (; len > 0; len -= nbytes)   
 {   
 nbytes = len; /\* set number of bytes wanted \*/

/\* read a zbuf from the socket \*/

if (((zRecv = zbufSockRecv (fd, 0, &nbytes)) == NULL) ||   
 (nbytes <= 0))   
 {   
 if (zRecvTotal != NULL)   
 zbufDelete (zRecvTotal);   
 return (NULL);   
 }

/\* append recv'ed zbuf onto end of zRecvTotal \*/

if (first)   
 zRecvTotal = zRecv;         /\* cannot append to empty zbuf \*/   
 first = FALSE;              /\* can append now... \*/   
 else if (zbufInsert (zRecvTotal, NULL, ZBUF\_END, zRecv) == NULL)   
 {   
 zbufDelete (zRecv);   
 zbufDelete (zRecvTotal);   
 return (NULL);   
 }   
 }

return (zRecvTotal);   
 }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
\*   
\* tcpServerWorkTask - process client requests   
\*   
\* This routine reads from the server's socket, and processes client   
\* requests. If the client requests a reply message, this routine   
\* sends a reply to the client.   
\*   
\* RETURNS: N/A.   
\*/

VOID tcpServerWorkTask   
 (   
 int sFd, /\* server's socket fd \*/   
 char \* address, /\* client's socket address \*/   
 u\_short port /\* client's socket port \*/   
 )   
 {   
 static char replyMsg[] = "Server received your message";   
 ZBUF\_ID zReplyOrig; /\* original reply msg \*/   
 ZBUF\_ID zReplyDup; /\* duplicate reply msg \*/   
 ZBUF\_ID zRequest; /\* request msg from client \*/   
 int msgLen; /\* request msg length \*/   
 int reply; /\* reply requested ? \*/

/\* create original reply message zbuf \*/

if ((zReplyOrig = zbufCreate ()) == NULL)   
 {   
 perror ("zbuf create");   
 free (address); /\* free malloc from inet\_ntoa() \*/   
 return;   
 }

/\* insert reply message into zbuf \*/

if (zbufInsertBuf (zReplyOrig, NULL, 0, replyMsg,   
 sizeof (replyMsg), NULL, 0) == NULL)   
 {   
 perror ("zbuf insert");   
 zbufDelete (zReplyOrig);   
 free (address); /\* free malloc from inet\_ntoa() \*/   
 return;   
 }

/\* read client request, display message \*/

while ((zRequest = zbufFioSockRecv (sFd, sizeof(struct request))) != NULL)   
 {   
 /\* extract reply field into <reply> \*/

(void) zbufExtractCopy (zRequest, NULL, 0,   
 (char \*) &reply, sizeof (reply));   
 (void) zbufCut (zRequest, NULL, 0, sizeof (reply));

/\* extract msgLen field into <msgLen> \*/

(void) zbufExtractCopy (zRequest, NULL, 0,   
 (char \*) &msgLen, sizeof (msgLen));   
 (void) zbufCut (zRequest, NULL, 0, sizeof (msgLen));

/\* duplicate reply message zbuf, preserving original \*/

if ((zReplyDup = zbufDup (zReplyOrig, NULL, 0, ZBUF\_END)) == NULL)   
 {   
 perror ("zbuf duplicate");   
 zbufDelete (zRequest);   
 break;   
 }

printf ("MESSAGE FROM CLIENT (Internet Address %s, port %d):\n",   
 address, port);   
   
 /\* display request message zbuf \*/

(void) zbufDisplay (zRequest, NULL, 0, msgLen, TRUE);   
 printf ("\n");   
 if (reply)   
 {   
 if (zbufSockSend (sFd, zReplyDup, sizeof (replyMsg), 0) < 0)   
 perror ("zbufSockSend");   
 }

/\* finished with request message zbuf \*/

zbufDelete (zRequest);   
 }

free (address); /\* free malloc from inet\_ntoa() \*/   
 zbufDelete (zReplyOrig);   
 close (sFd);   
 }

|  |  |
| --- | --- |
|  |  |
|  | **CAUTION:** In the interests of brevity, the **STATUS** return values for several zbuf socket calls are discarded with casts to **void**. In a real application, check these return values for possible errors. |
|  |  |
|  | |