Multitasking Servers on MVS with SAS/C

Greg Hester
Elliot Poger
Jeremy Smith
Alan Beale

Abstract

Many MVS applications are being moved to a client/server model which utilizes multitasking servers to service multiple clients at one time. TCP/IP networks are often chosen as the transport mechanism and the MVS ATTACH facility is used to accomplish multitasking.

The goal of this paper is to demonstrate how to use the SAS/C Compiler product to create multitasking servers which utilize TCP/IP and BSD sockets to communicate. The information in this paper is based on Release 5.50C of the SAS/C Compiler product and Release 2.2.1 of IBM TCP/IP.

This paper will discuss multitasking on MVS, BSD Sockets, and then will give a complete C sample of a multitasking server on MVS using SAS/C.

Multitasking on MVS

From an application programming point of view, multitasking is a way to increase turnaround by making use of the multiple processors that are available on an machine. By dividing an application into tasks that can be executed simultaneously, total turnaround time for an application can be reduced.

Additionally, by using multitasking in the design of a server, one contact point can be given to potential clients so as to obtain multiple services which can run simultaneously on the hardware. Thus, one server can service multiple clients simultaneously without requiring program logic to accommodate multiple concurrent service requests.

MVS Overview
Multitasking is achieved on MVS by making use of the ATTACH facility. The ATTACH macro causes the control program to create a new task and indicates the entry point in the program to be given control when the task is activated. MVS provides the following macros which aid in writing a multitasking application:

ATTACH	Create a new subtask
CHAP	Change a Tasks Priority
DETACH	Remove a Subtask from the system
DEQ	Release an enqueued resource
ENQ	Wait for a resource to become available
POST	Post an ECB
STATUS	Halt or resume a subtask
WAIT	Wait for the POST of one or more ECB's

SAS/C Specifics

The SAS/C library at release 5.50C provides equivalent C macros or inline C functions for calling the MVS multitasking facilities. However, the SAS/C library, in general, does not support Multitasking. What this means is that the library as a whole does not keep track of any information about other tasks (possibly related) that may be executing in the same address space or other address spaces. Because of this, a certain amount of care must be taken when writing a SAS/C program that will multitask. Following are a few "rules of thumb" for writing a SAS/C program that will multitask.

parameters to subtasks
Do not use automatic variables as parameters to an ATTACHed program. Automatic variables may get changed before the attached program accesses them.

When "ATTACH-ing" the same program more than once ensure that the program is compiled RENT. Also, avoid passing the same parameters to both instances of the task, unless all the parameters, and all the data addressed from the parameters are read-only. Changing the data for the second instance of the program could upset the first task if it has not read the parameters yet.

If "ATTACH-ing" a SAS/C program with the normal main() entry point, you must provide an OS-format parameter list (i.e., register 1 points to a word which addresses a field which starts with a two-byte length prefix and the high-order bit is on in the word addressed by R1). When the caller of ATTACH does not bother to create a parameter list (perhaps no parameters are needed), then generally the ATTACHed task will abend with an S0C4.

stdout & stderr
Care must be taken when using stdout & stderr. If 2 or more subtasks try to write to stdout or stderr then the output may appear to be out of order. If stdout/stderr are defined to the terminal or SYSOUT, ordinarily no output would be lost. However, if stdout/stderr are defined to a disk dataset, abends are very likely as well as losing random bits of data in the output.

It is best to re-direct stdout to alternate destinations for each subtask. Stdout can be re-directed by using the ">" command line parameter or the SAS/C external variable _stdonm. Stderr can be re-directed by using the SAS/C external variable _stdenm.

C runtime environment
For each subtask that uses SAS/C, a new C runtime environment must be created. The CRAB was not designed to be re-entrant, so sharing the CRAB between tasks will certainly result in abends and other strange behavior.

One reason why the CRAB is not re-entrant is the fact that the library keeps the address of the current PRV in the CRAB. The PRV is different for each task, so this area in the CRAB would also change. Another reason is the fact that the current stack pointer and end-of-stack pointer are both stored in the CRAB. These pointers will also be different for each task.

If there is a question about the subtasks using different environments, you can look at the value of R12 in a dump or interactively using TSO/TEST. R12 should be different for the various subtasks.

shared memory between subtasks
Memory allocated by one subtask may be shared with other subtasks. However, memory malloc'd under one task cannot be freed by another (even if subpools are shared). This type of operation will corrupt the

HEAP for both tasks. Also, if a subtask has a pointer to some memory owned by another subtask which has been deleted from the system, then the pointer will be invalid and abends will result upon use that pointer.

If an application needs to share memory between subtasks, we recommend that a "memory manager" which will live until the application ends be implemented. The memory manager will receive malloc/free requests from other subtasks, allocate/free the memory, and then inform the requestor of the address of the allocated memory for malloc requests, and a return code for free requests. The communication between the subtasks can be accomplished with POST & WAIT logic. The important issue is that the subtask which allocates the shared memory must be guaranteed to live longer or as long as any task that needs to use shared memory. This can be accomplished by including the "memory manager" in the main task, or by putting it in a task which will not be deleted until the application is complete.

BSD Sockets

Sockets are an API for network communications across a TCP/IP network. The University of California at Berkeley designed and implemented the Socket API. Subsequently, many vendors, including IBM and SAS, have ported the BSD socket code to the S/370 platform.

A socket is simply an endpoint for communication. It typically takes 2 endpoints to communicate. Communication can proceed in a client/server fashion or peer-to-peer. Also, a socket application can communicate with similar socket applications in a connection-less mode (UDP, datagrams) or in a connection oriented mode (TCP, stream).

Depending upon the application, the flow of a Socket program may vary somewhat. However, following is the basic flow for UDP and TCP socket applications.

UDP Program Flow

```
 SERVER                             CLIENT
 ------                             ------
 socket()
   |
   V
 bind()
   |
   V
 recvfrom()                         socket()
   |                                   |
   V
  *blocks I/O until data            bind()
    received from client               |
   |                                   V
   |      *data (request) +---------sendto()
   |                      |            |
   |  <-------------------+            |
   |                                   |
   V                                   |
*process request                       |
   |                                   |
   V       *data (reply)               |
 sendto() ----------------+            |
   |                      |            V
   |                      +-------> recvfrom()
   |                                   |
   V
 close()                            close()
```

TCP Program Flow

```
 SERVER                             CLIENT
 ------                             ------
 socket()
   |
   V
 bind()
   |
   V
 listen()
   |
   V
 accept()
   |
   V                                socket()
  *blocks I/O until                    |
    client connects                    V
   |                                bind()
   |                                   |
   |    *connection est.               V
   |<------------------------------ connect()
   |                                   |
   |                                   V
   |      *data (request) +-------- write()
   V                      |            |
 read() <-----------------+            |
   |                                   |
   V                                   |
*process request                       |
   |                                   |
   V      *data (reply)                |
 write() -----------------+            |
   |                      |            V
   |                      +-------> read()
 *further processing                   |
   |                                   |
   V                                   V
 close()                            close()
```

A Multitasking TCP/IP server in SAS/C on MVS

This is a connection based example using TCP. The server is started on MVS and then clients connect via telnet or perhaps another socket program. Once connected, the client can then send the server the name of a utility to run on MVS. The server then does a givesocket() and then ATTACHes the requested program and readies itself for another request.

Program Flow

```
   SERVER                              CLIENT
   ------                              ------
   socket()
     |
     V
   bind()
     |
     V
   listen()
     |
     V
+> select() *loop until a
|    |          socket is ready
|    |
|   *blocks I/O until client
|     connects or timeout
+----|
     V
   accept()
     |
     V
     |
     |
     |  *connection est.
     |<------------------------------- TELNET SDCMVS 3002
     |                                    |
     |                                    V
     |  *data (request)      +-------- TIMESRV
     V                       |
   read() <------------------+
     |
     V
givesocket()     +----------------------------------------+
     |           |           SUBTASK- TIMESRV              |
     V           |                                         |
  ATTACH() ==>   |      takesocket()                       |
     |           |         |                               |
     V           |         V                               |
   close()       |       write()                           |
                 |         |                               |
                 |         V                               |
                 |      close()                            |
                 |                                         |
                 +----------------------------------------+
```

Appendix A. listener.c

```
/*-------------------------------------------------------------------+
 |                                                                   |
 |                       S A S / C   S A M P L E                     |
 |                                                                   |
 |          NAME: LISTENER                                           |
 |       PURPOSE: Demonstrate socket function calls and ATTACH for MVS|
 |                via a listener server type application             |
 |  INSTALLATION:                                                    |
 |       COMPILE: LC370 LISTENER EXTNAME                             |
 |          LINK: CLK370 LISTENER                                    |
 |  INPUT/OUTPUT: TCP/IP sockets                                     |
 |         USAGE: see SAS Technical Report C-111 (SAS/C Socket Library|
 |                for TCP/IP)                                        |
 |                TELNET may be used to communicate with this server:|
 |                TELNET <listener_host_name> portid                 |
 |                At the TELNET input prompt enter the load module   |
 |                name of the program to be ATTACH'ed e.g. timesrv   |
 |                                                                   |
 |  SYSTEM NOTES: The program ATTACH'ed by the server must make use  |
 |                of the supplied getsocket() function contained in  |
 |                the GETSOCK source file.  This function will allow |
 |                the attached pgm to either use a socket given by the|
 |                LISTENER or it will obtain, bind, listen, and accept|
 |                a new socket.                                      |
 |                                                                   |
 +-------------------------------------------------------------------*/

#define LISTENPORT 0    /* Let TCP/IP Decide the port #             */
#define QUITSTRING "LSNRQUIT"
#define MAXSOCKETS 50   /* max sockets opened at once               */

/*------------------------------------------------------------------+
* Includes for socket calls.                                        *
+------------------------------------------------------------------*/
#include <sys/types.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <fcntl.h>
/*------------------------------------------------------------------+
* Include for ATTACH.                                               *
+------------------------------------------------------------------*/
#include <ostask.h>
/*------------------------------------------------------------------+
* Miscellaneous includes.                                           *
+------------------------------------------------------------------*/
#include <lclib.h>
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <lcstring.h>

#define TRUE 1
#define FALSE 0
```

```
#define MAXREQLEN 256   /* max length of request, in chars          */

/*-------------------------------------------------------------------+
* Macros for converting EBCDIC <-> ASCII.                            *
+-------------------------------------------------------------------*/
#define convFromAscii(str) {cp=(str); while(*cp = ntohcs(*cp)) cp++;}
#define convToAscii(str)   {cp=(str); while(*cp = htoncs(*cp)) cp++;}

int setupSocket(void);
int acceptRequest(void);
int closeSockets(int);
void Quit(void);

struct Argument {
   short len;
   char parms[1000];
};

struct socklist {       /* linked list of file desriptors of sockets*/
   int sock;
   int  domain;         /*  so that it can issue takesocket()       */
   char name[8];        /*  so that it can issue takesocket()       */
   char subtaskname[8];
   char pgmname[8];
   void *subtcb;        /* Handle for ATTACHed program              */
   void *plist[1];      /* pointer to argument structure for ATTACH */
   struct socklist *next;
} *servlist;

char *cp;               /* char pointer for EBCDIC<->ASCII macros   */
int listensock,         /* socket on which we listen for requests   */
    numsockets,         /* number of sockets opened by listener     */
    highsocket,         /* highest socket number                    */
    holdrequests;       /* flag: listener cannot handle any more    */
                        /*  requests right now (all sockets used)   */
fd_set pending,         /* list of sockets waiting to be taken      */
       taken;           /* list of sockets which have been taken    */
struct clientid myid,   /* clientid of listener for argument struct */
                giveid; /* generic clientid used for givesocket()   */

void main()
{
   int ns;              /* number of sockets with action            */
   fd_set incoming;     /* bound socket description for select()    */
   struct timeval timeout;

   /*----------------------------------------------------------------+
   * Set up a socket, initialize list of pending servers to none,    *
   * and set timeout for select() call.                              *
   +----------------------------------------------------------------*/
   if (setupSocket() == -1)
      exit(EXIT_FAILURE);
   FD_ZERO(&pending);
   holdrequests = FALSE;
   timeout.tv_sec = 60;
   timeout.tv_usec = 0;
   while (TRUE)
   {
      do {
         /*-------------------------------------------------------------+
```

```
         * Check for action on sockets given to ATTACHed programs,   *
         * as well as incoming socket (unless all sockets are used). *
         +-----------------------------------------------------------*/
         FD_ZERO(&incoming);
         if ((!holdrequests) && (numsockets < MAXSOCKETS))
            FD_SET(listensock, &incoming);
         taken = pending;
         ns = select(highsocket+1, &incoming, NULL, &taken, &timeout);
      } while (ns == 0);
      if (ns == -1)
      {
         perror("select() call failed");
         Quit();
      }
      /*-------------------------------------------------------------+
      * If there was action on listensock, then handle the incoming  *
      * request.                                                     *
      +-------------------------------------------------------------*/
      if (FD_ISSET(listensock, &incoming))
      {
         ns--;
         acceptRequest();
      }
      /*-------------------------------------------------------------+
      * If there was action on any other sockets, then release the   *
      * sockets of those programs which have taken them.             *
      +-------------------------------------------------------------*/
      if (ns > 0)
         closeSockets(ns);
   }
}


/*-------------------------------------------------------------------+
* Open a socket and set it up to accept requests. Return -1 if error.*
+-------------------------------------------------------------------*/
int setupSocket()
{
   struct sockaddr_in sa_serv;  /* socket address for server        */
   int sa_len = sizeof(sa_serv);
   char hostname[72];

   /*----------------------------------------------------------------+
   * Get a socket of the proper type.                                *
   +----------------------------------------------------------------*/
   listensock = socket(AF_INET, SOCK_STREAM, 0);
   if (listensock == -1)
   {
      perror("socket() call failed in setupSocket()");
      return -1;
   }
   numsockets = 1;

   /*----------------------------------------------------------------+
   * Get information about listener to pass to ATTACHed servers.     *
   * Clientid used for givesocket() should be the same but with a    *
   * subtaskname of all blanks.                                      *
   +----------------------------------------------------------------*/
   if (getclientid(AF_INET, &myid) == -1)
   {
```

```
      perror("getclientid() call failed in setupSocket()");
      close(listensock);
      return -1;
   }
   giveid = myid;
   memset(giveid.subtaskname, ' ', 8);

   /*---------------------------------------------------------------+
   * Prepare a socket address for the server.  We will request the   *
   * port number LISTENPORT, unless LISTENPORT is 0, in which case   *
   * we are assigned the next available port number.                *
   +---------------------------------------------------------------*/
   memset(&sa_serv, '\0', sizeof(sa_serv));
   sa_serv.sin_family = AF_INET;
   sa_serv.sin_addr.s_addr = htonl(INADDR_ANY);
   sa_serv.sin_port = htons(LISTENPORT);

   /*---------------------------------------------------------------+
   * Bind our socket to the desired address so clients may reach us. *
   +---------------------------------------------------------------*/
   if (bind(listensock, &sa_serv, sizeof(sa_serv)) == -1)
   {
      perror("bind() call failed in setupSocket()");
      close(listensock);
      return -1;
   }

   /*---------------------------------------------------------------+
   * Print listener information.                                     *
   +---------------------------------------------------------------*/
   if (getsockname(listensock, &sa_serv, &sa_len) == -1)
   {
      perror("getsockname() failed in setupSocket()");
      close(listensock);
      return -1;
   }

   if (gethostname(hostname, sizeof(hostname)))
   {
      perror("gethostname() failed in setupSocket()");
      close(listensock);
      return -1;
   }
   printf("Listener is on Host:  %.64s   Port#:  %d.\n",
           hostname,  (int) ntohs(sa_serv.sin_port));
   printf("Request server name '" QUITSTRING "' to quit.\n");

   /*---------------------------------------------------------------+
   * Set up a queue for incoming requests.  SOMAXCONN is maximum     *
   * size of queue.                                                  *
   +---------------------------------------------------------------*/
   listen(listensock, SOMAXCONN);
   highsocket = listensock;
   return 0;
}


/*-------------------------------------------------------------------+
* A request has been detected--accept it, givesocket(), and ATTACH   *
* program.  Return -1 if error.                                      *
```

```
+------------------------------------------------------------------*/
int acceptRequest()
{
   int numchars,               /* number of chars read from socket */
       rc,                     /* rc from MVS ATTACH function      */
       pos,                    /* index variable                   */
       reqsock,                /* request socket                   */
       i;                      /* index variable                   */
   char req[MAXREQLEN+1],      /* buffer for reading in request    */
        errstring[MAXREQLEN+80];/* error message sent to client     */
   struct socklist *new;       /* new node in list of open sockets */
   struct sockaddr_in sa_client;/* socket address of client         */
   int sa_len = sizeof(sa_client);
   char Env_buff[256];
   struct Argument *args;
   char *valid = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" \
                 "abcdefghijklmnopqrstuvwxyz" \
                 "0123456789@$#";

   /*---------------------------------------------------------------+
   * Allocate space for a new addtion to the socklist               *
   +---------------------------------------------------------------*/
   new  = (struct socklist *)malloc(sizeof(struct socklist));
   if (new == NULL)
      {
      perror("malloc() for new failed in acceptRequest()");
      return -1;
      }
   memset(new, 0, sizeof(struct socklist));

   args = (struct Argument *)malloc(sizeof(struct Argument));
   if (args == NULL)
      {
      perror("malloc() for args failed in acceptRequest()");
      return -1;
      }
   memset(args, 0, sizeof(struct Argument));

   /*---------------------------------------------------------------+
   * Divert incoming request to its own socket.  If accept() call    *
   * fails, then all sockets are being used, so hold off requests.   *
   +---------------------------------------------------------------*/
   reqsock = accept(listensock, &sa_client, &sa_len);
   if (reqsock == -1)
   {
      perror("accept() failed in acceptRequest()");
      holdrequests = TRUE;
      return -1;
   }

   /*---------------------------------------------------------------+
   * Read request from socket and convert from ASCII to EBCDIC.      *
   +---------------------------------------------------------------*/
   numchars = read(reqsock, req, MAXREQLEN);
   if (numchars < 0)
     {
      perror("socket read() failed in acceptRequest()");
      close(reqsock);
      return -1;
     }
```

```
req[numchars] = '\0';
convFromAscii(req);

/*-----------------------------------------------------------------+
* Check request for validity (a pretty lax check, admittedly).    *
+-----------------------------------------------------------------*/
if (req[0] == '\0')
{
   strcpy(errstring, "MVS listener: empty request received.\n");
   convToAscii(errstring);
   write(reqsock, errstring, strlen(errstring));
   close(reqsock);
   return -1;
}

/*-----------------------------------------------------------------+
| Set up the Env vars string for subtaskname, name, sock, etc.    |
+-----------------------------------------------------------------*/
new->domain = AF_INET;
memcpy(new->name, myid.name, 8);
memcpy(new->subtaskname, myid.subtaskname, 8);
sprintf(Env_buff, " =SOCK=%d =DOMAIN=%d =NAME=%s =SUBTASKNAME=%s",
        reqsock, new->domain, new->name, new->subtaskname);

/*-----------------------------------------------------------------+
* Separate the command from the parmameters                       *
+-----------------------------------------------------------------*/
pos = strspn(req, valid);
req[pos] = '\0';
strcpy(new->pgmname, req);
strcpy(args->parms, (req+pos+1));

/*-----------------------------------------------------------------+
* Clean up the Program name.                                      *
+-----------------------------------------------------------------*/
for (i = 0; i < sizeof(new->pgmname); i++)
   {
    if  (new->pgmname[i] <  ' ')
         new->pgmname[i]  = ' ';
    else new->pgmname[i] |= ' ';
   }

/*-----------------------------------------------------------------+
* Check for QUIT request.                                         *
+-----------------------------------------------------------------*/
if (strcmp(new->pgmname, QUITSTRING) == 0)
{
   close(reqsock);
   Quit();
}

/*-----------------------------------------------------------------+
* Combine the Program parms and the environment vars and assign   *
* to the plist pointer in new.                                    *
+-----------------------------------------------------------------*/
strcat(args->parms, Env_buff);
args->len = strlen(args->parms);
for (i = 0; i < args->len; i++)
   {
    if  (args->parms[i] <  ' ')
```

```
         args->parms[i]  = ' ';
      }

   new->plist [0] = (void *) (0x80000000 | (unsigned) args);

   /*----------------------------------------------------------------+
   * Give the socket and then attach the new subtask.                 *
   +----------------------------------------------------------------*/
   givesocket(reqsock, &giveid);
   rc = ATTACH(&(new->subtcb), _Aep, new->pgmname, _Aparam,
               new->plist, _Aend);

   /*----------------------------------------------------------------+
   * Check return code from ATTACH.                                   *
   +----------------------------------------------------------------*/
   if (rc != 0)
   {
      fprintf(stderr, "ATTACH for server '%s' failed with rc = %d.\n",
              new->pgmname, rc);
      close(reqsock);
      return -1;
   }

   /*----------------------------------------------------------------+
   * Add this server to lists of pending programs.                    *
   +----------------------------------------------------------------*/
   new->sock = reqsock;
   new->next = servlist;
   servlist = new;
   FD_SET(reqsock, &pending);
   if (reqsock > highsocket)
      highsocket = reqsock;
   numsockets++;
   return 0;
}


/*-------------------------------------------------------------------+
* Close sockets taken by ATTACHed servers.  Return -1 if error.      *
+-------------------------------------------------------------------*/
int closeSockets(nt)
int nt;         /* number of sockets taken by ATTACHed servers      */
{
   struct socklist *servptr,    /* list of file descriptors of      */
                 *next, *prev;  /*  sockets given to servers        */
   int nr = nt;                 /* number of sockets left to close  */

   prev = servptr = servlist;   /* point to first socket in list    */
   while ((nr > 0) && (servptr != NULL))
   {
      /*-------------------------------------------------------------+
      * If this socket was taken by an ATTACHed server, then close   *
      * the socket, remove it from our lists, and signal that another*
      * socket is free for use.                                      *
      +-------------------------------------------------------------*/
      if (FD_ISSET(servptr->sock, &taken))
      {
         close(servptr->sock);
         nr--;
         numsockets--;
```

```
         holdrequests = FALSE;
         FD_CLR(servptr->sock, &pending);
         next = servptr->next;
         if (servptr == servlist)
            servlist = next;
         else
            prev->next = next;
         free((char *) servptr->plist[0]);
         free((char *) servptr);
         servptr = next;
      }
      else
      {
         prev = servptr;
         servptr = servptr->next;
      }
   }
   return 0;
}


/*-------------------------------------------------------------------+
* Close all sockets and quit.                                        *
+-------------------------------------------------------------------*/
void Quit()
{
   struct socklist *servptr, *next;

   next = servlist;
   while (next != NULL)
   {
      servptr = next;
      next = servptr->next;
      close(servptr->sock);
      free((char *) servptr->plist[0]);
      free((char *) servptr);
   }
   close(listensock);
   exit(EXIT_SUCCESS);
}
```

Appendix B.   getsock.c

```
/*-------------------------------------------------------------------+
*                                                                    *
*                       S A S / C   S A M P L E                      *
*                                                                    *
*          NAME: GETSOCK                                             *
*       PURPOSE: Implement the getsocket() function which will either *
*                use a socket given by the LISTENER or create a new   *
*                socket for use.                                     *
* INSTALLATION:                                                      *
*       COMPILE: see LISTENER jcl                                    *
*          LINK: see LISTENER jcl                                    *
* INPUT/OUTPUT: TCP/IP sockets                                       *
*         USAGE: see SAS Technical Report C-111 (SAS/C Socket Library *
*                for TCP/IP)                                         *
* SYSTEM NOTES:                                                      *
+-------------------------------------------------------------------*/

#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define TIMEPORT 0

/*----------------------------------------------------------------
 * Get the Socket either from a calling Main task of via a
 * Socket,bind, sequence.
 */
int GetSocket(int *cs)
   {

   /* struct for clientid information                          */
   struct clientid clientid;
   char *buffer;

   /* Socket addresses for server and client. */
   struct sockaddr_in sa_serv;
   struct sockaddr_in sa_clnt;
   int sa_len;

   int s;  /* Working socket                                        */
   int i;
   int rc=0;

   /*
    * If The Environment variable SOCK is not Null, we were attached
    * by the LISTENER and need to call takesocket to communicate.
    * Otherwise, we need to open a socket and accept a request on that.
    */
   buffer = getenv("SOCK");
   if (buffer != NULL)
      {
      /*
       * Build a clientid struct for takesocket
       */
      memset(&clientid, 0, sizeof(clientid)); /* zero out client id */
```

```
   s = atoi(buffer);
   buffer = getenv("DOMAIN");
   clientid.domain = atoi(buffer);
   buffer = getenv("NAME");
   memcpy(clientid.name, buffer, 8);

   /*Pad out the program name.                                      */
   for (i = 0; i < sizeof(clientid.name); i++)
      {
       if  (clientid.name[i] < ' ')
            clientid.name[i] = ' ';
       else clientid.name[i] |= ' ';
      }

   buffer = getenv("SUBTASKNAME");
   memcpy(clientid.subtaskname, buffer, 8);

   /*Pad out the Subtask name.                                      */
   for (i = 0; i < sizeof(clientid.subtaskname); i++)
      {
      if  (clientid.subtaskname[i] < ' ')
           clientid.subtaskname[i] = ' ';
      }

   printf("info from ENV vars:"
          "Domain = <%d>, Socket# <%d> Name = <%.8s>, Task = <%.8s>\n",
          clientid.domain, s, clientid.name, clientid.subtaskname);

   /* Take the passed client socket; takesocket will return a local   */
   /*  socket number enabling us to write to the client               */

   *cs = takesocket(&clientid, s);

   /* Check takesocket rc */
   if (*cs == -1)
      {
      perror("takesock() call failed");
      rc = -1;
      }
   }
else
   {
   /*
    * Since this instance is standalone (not invoked by LISTENER,
    *   get a stream socket to communicate.
    */
   s = socket(AF_INET, SOCK_STREAM, 0);
   if (s == -1)
      {
      perror("timesvr - socket() failed");
      rc = -1;
      }

   /* Prepare a socket address for the server.            */
   memset(&sa_serv,'\0',sizeof(sa_serv));
   sa_serv.sin_family = AF_INET;
   sa_serv.sin_addr.s_addr = INADDR_ANY;
   sa_serv.sin_port = TIMEPORT;

   /* Bind our socket to the desired address. Now clients*/
```

```
      /*   specifying this address will reach this server.  */
      if (bind(s, &sa_serv, sizeof(sa_serv)) == -1)
         {
         perror("timesrv - bind() failed");
         rc = -1;
         }

      /* Find out what port was choosen and report it.      */
      sa_len = sizeof(sa_serv);
      if (getsockname(s, &sa_serv, &sa_len) == -1)
         {
         perror("timesrv - getsockname() failed");
         rc = -1;
         }
      printf("TIMESRV server port is: %d\n",
                     (int) ntohs(sa_serv.sin_port));

      /* Set up a queue for incoming connection requests.   */
      listen(s, SOMAXCONN);

      /* Accept a new request. Ask for client's address  */
      /*   so that we can print it if there is an error. */
      sa_len = sizeof(sa_clnt);
      *cs = accept(s, &sa_clnt, &sa_len);
      if (*cs==-1)
         {
         perror("timesrv - accept() failed");
         rc = -1;
         }
      }

   return(rc);
}
```

Appendix C. timesrv.c

```
/*-----------------------------------------------------------------+
*                                                                  *
*                     S A S / C   S A M P L E                      *
*                                                                  *
*         NAME: TIMESRV                                            *
*      PURPOSE: Demonstrate socket function calls and ATTACH for MVS.*
* INSTALLATION:                                                    *
*      COMPILE: see LISTENER jcl                                   *
*         LINK: see LISTENER jcl                                   *
* INPUT/OUTPUT: TCP/IP sockets                                     *
*        USAGE: see SAS Technical Report C-111 (SAS/C Socket Library *
*               for TCP/IP)                                        *
* SYSTEM NOTES:                                                    *
+-----------------------------------------------------------------*/

#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <options.h>

#if defined(DEBUG)
int _options = _DEBUG;
#endif

extern int GetSocket(int *cs);

main(int argc, void **argv)
{
   int cs=0;
   int i;

   /* Variables used to obtain the time.                    */
   char *p;
   int len;
   time_t t;

   /* Loop count */
   int n_times;

   /* Buffer for outgoing time string.                      */
   char outbuf[128];

   /* If there were any arguments passed, echo them  */
   if (argc >1)
      {
      for (i=0; i<argc; i++)
         printf("argv[%d] = %s\n", i, argv[i]);
      }

   /*
    * Get a socket.  If TIMESRV was attached by the LISTENER, then
    * GetSocket() call takesocket().  Otherwise, GetSocket() will
    * open a new socket, bind, listen and accept before returning.
    */
```

```
   if (GetSocket(&cs) == -1)
      exit(EXIT_FAILURE);

   /* Now, get the time and write it out to the client */
   n_times = 2;
   while (n_times--)
   {
      /* Send the time to the client. clients           */
      /*  expect the string to be in ASCII.             */
      time(&t);               /* Machine readable time. */
      p = ctime(&t);          /* Human readable time.   */
      /* Convert to ASCII if necessary. */
      for (len=0; p[len] && len<sizeof(outbuf); len++)
         outbuf[len] = htoncs(p[len]);
      outbuf[len+1] = htoncs('\n');   /* send a new line*/

      if (write(cs, outbuf, len)==-1)
      {
         perror("write() failed");
         return EXIT_FAILURE;
      }
   }
   close(cs);
   return EXIT_SUCCESS;  /* Avoid compilation warnings. */
}
```

Appendix D. listener.jcl

```
//LISTENER JOB job card info...
/*JOBPARM FETCH
//*****************************************************************
//*  COMPILE THE LISTENER
//*
//CLISTEN  EXEC LC370C,PARM.C='EXTNAME ENXREF RENT SN(LISTN)'
//C.SYSLIN   DD DSN=yourid.SASC.OBJ(LISTENER),DISP=SHR
//C.SYSDBLIB DD DSN=yourid.SASC.DBGLIB,DISP=SHR
//C.SYSIN    DD DSN=yourid.SASC.C(LISTENER),DISP=SHR
//*****************************************************************
//*  CLINK THE LISTENER
//*
//LLISTEN  EXEC LC370LR
//LKED.SYSIN   DD DSN=yourid.SASC.OBJ(LISTENER),DISP=SHR
//LKED.SYSLMOD DD DSN=yourid.SASC.LOAD(LISTENER),DISP=SHR
//*****************************************************************
//*  COMPILE THE GETSOCKET FUNCTION
//*
//CGETSOCK EXEC LC370C,PARM.C='EXTNAME ENXREF RENT SN(GETSK)'
//C.SYSLIN   DD DSN=yourid.SASC.OBJ(GETSOCK),DISP=SHR
//C.SYSDBLIB DD DSN=yourid.SASC.DBGLIB,DISP=SHR
//C.SYSIN    DD DSN=yourid.SASC.C(GETSOCK),DISP=SHR
//****************************************************************
//*  COMPILE THE TIMESRV PGM
//*
//CTIMESV  EXEC LC370C,PARM.C='EXTNAME ENXREF RENT SN(TIMES)'
//C.SYSLIN   DD DSN=yourid.SASC.OBJ(TIMESRV),DISP=SHR
//C.SYSDBLIB DD DSN=yourid.SASC.DBGLIB,DISP=SHR
//C.SYSIN    DD DSN=yourid.SASC.C(TIMESRV),DISP=SHR
//*****************************************************************
//*  CLINK THE TIMESRV PGM WITH THE GETSOCKET FUNCTION
//*
//LTIMESV  EXEC LC370LR
//LKED.SYSLMOD DD DSN=yourid.SASC.LOAD(TIMESRV),DISP=SHR
//LKED.SYSIN  DD *
 INCLUDE MYOBJ(TIMESRV)
 INCLUDE MYOBJ(GETSOCK)
 NAME TIMESRV(R)
/*
//LKED.MYOBJ  DD DSN=yourid.SASC.OBJ,DISP=SHR
//
```