DeviceMaster®
SocketServer Extension Guide

Introduction

This document describes how to add functionality to the example SocketServer application. It does not describe the internal workings of all of the parts of SocketServer, but it does describe how the **makefile** and the directory structure work and how to add functionality in a modular fashion.

Overview

SocketServer is organized in two levels: The main directory/makefile and a set of subdirectories, each with it's own **makefile**. The main directory contains some global initialization functions that performs the following functions:

- Read configuration data from I2C EPROM.
- Initialize the network stack.
- Call module initialization entry points.

All of the other functionality (e.g. web server, socket server, SNMP, telnet server) are provided by modules in subdirectories. This make it easy to add/remove functional modules from the build. It also make it easy to build multiple target images that include different combinations of modules. In the sample source tree provided, the modules are:

- **snmp**: starts the SNMP server included with eCos
- **SocketServer**: provides the socket-server functionality.
- **webserv**: starts the GoAhead web server and provides web support functions (entry points for JavaScript programming and forms handling functions). Also implements framework for telnet server.
- **webpages**: provides the ROM-image of the web pages.
- **admin**: Network interface for Comtrol administration tools.
- **myextension**: example extension that echoes data on Port 0.

In order to reduce the modifications needed to the GoAhead distribution, it is built as a library and the resulting library is linked in. It does not follow the conventions that the other modules do.

Module Conventions

Linkage

Each module is contained in a subdirectory. That subdirectory contains a makefile that is invoked from the main makefile. Doing a **make** directly in a subdirectory may not work since some **makefile** variables (**CFLAGS**, **CC**, etc.) are passed from the main **makefile**. The module makefile compiles the appropriate source files and links them together into a single object file named **app.o**. This file is then linked in by the main **makefile**.

Entry Points

Each module may provide two initialization entry points. One is called before the eCos scheduler is started from the **cyg_user_start**() function. The other is called after the eCos scheduler is started and network stack initialization has completed. These two entry points are placed into special linker sections using the macros defined in **dm.h**. Example usage of these macros is shown below:

```
static void myInit0(void)
{
// runs before scheduler has started, so not all eCos system
// calls are available.
[...]
}

DeviceMaster_init0(myInit0,1000);

static void myInit1(void)
{
// system is up and running
[...]
}

DeviceMaster_init1(myInit1,1000);
```

A module may provider either one or both of these entry points.

Both of these initialization functions must return to the caller. The initialization routines for all modules are called one at a time in a sequence determined by the lexicographical order of the macro's second parameter. This is by convention a 4-digit decimal number: 0000 is first, and 9999 is last.

You may create and initialize eCos threads in either of the initialization functions.

Note that the functions do not need to be globally visible (and generally should not be). It is a good idea to limit the lexical scope of everything in your modules as much as possible. If you must make symbols globally visible, prefix them with a unique module name to avoid name-space conflicts with the global symbols in other modules.

It should be possible to isolate a module's **namespace** from the rest of the program. For example, a module might consist of multiple source files that need to share global data, but that global data must not be visible to other modules. This can be done using a more elaborate linker command in the module makefile where **app.o** is created. It is possible to tell the linker to eliminate all completely resolved symbols from the resulting file except for a specific list of global symbols that are to remain visible. If you need to do this, see the GNU Using ld manual for more information.

System Resources

Serial Ports

The DeviceMaster serial ports may be accessed through facilities declared in the file **dm.h**:

```
typedef struct
{
  cyg_sem_t  ownership;
  cyg_io_handle_t handle;
}tDevMastPort;

tDevMastPort devMastPort[];
```

The global array **devMastPort** contains I/O handles and ownership semaphores for each of the serial ports. You should always perform a wait operation on the semaphore to acquire ownership of the port before accessing a serial port. When you wish to release ownership, do a post operation on the semaphore.

```
#define DevMastRsMode_232  0x00
#define DevMastRsMode_232I 0x18
#define DevMastRsMode_422  0x08
#define DevMastRsMode_485  0x10

extern void devMastSetPortMode(int port,int mode);
```

The function **devMastSetPortMode**() is used to set the electrical interface mode on a port. This is a write-only operation and there is no way to query the current interface mode.

Configuration Data

The **ident** structure defined by ident.h will contain various configuration data that were read from the I2C EPROM:

```
typedef struct
{
  unsigned long   modelId;
  unsigned short  programId;
  unsigned char   archId;
  unsigned long   boardRev;
  unsigned char   numPorts;
  unsigned char   versionMajor;
  unsigned char   versionMinor;
  unsigned long   ipAddr;
  unsigned long   netMask;
  unsigned long   gateway;
  unsigned char   macAddr[6];
  unsigned char   password[16];
  unsigned char   authMethod;
  unsigned char   telnetEnable;
  unsigned char   timeoutValue;
  unsigned char   telnetTimeout;
} tIdent;

extern tIdent ident;
```

These values are written to the I2C EPROM by the bootloader.

Adding a Module

Adding a module requires two steps:

1. Create a subdirectory containing source files and a **Makefile**. For your module's **Makefile**, it's generally easiest to copy a **Makefile** from another module and modify it as required.
2. Edit the main **Makefile** in two places.

   First, add the subdirectory to the **SUBDIRS** variable:

   SUBDIRS = snmp socketServer webserv webpages admin myextension

   Second, add the **app.o** object file to the **SOCKAPPS** variable:

   SOCKAPPS = snmp/app.o socketServer/app.o admin/app.o webserv/app.o \ webpages/app.o goahead/ECOS/ libwebs.a myextension/app.o

Example Module

The **myextension** subdirectory contains a simple module example. The module will acquire ownership of serial Port 0, configure the port for RS-232 operation at 9600 baud, and then echo data received on that port.

The **myextension** subdirectory contains the module's **Makefile** and a single source file, **task.c**.

The **Makefile** looks like this:

```
all: app.o

SRCS = task.c

%.o: %.c
$(XCC) -c -o $@ $(CFLAGS) $(INCLUDES) $(DEFINES) $<

app.o: task.o
$(XLD) -i -o $@ $^

clean:
rm -f  *.o *~ *.lst *.map \#*\# *.bin *.elf10 core *.bak .*~

depend:
gcc -M $(INCLUDES) $(DEFINES) $(SRCS) >.srcdeps

include .srcdeps
```

The values for the **XCC**, **XLD**, **CFLAGS**, **INCLUDES**, **DEFINES** variables are passed from the parent make, so you don not have to worry about defining them in module **makefile**. If you add source files, you should add them to the **SRCS** variable definition and corresponding object files to the dependency list for **app.o**.

The source file for the module itself looks like this:

```
#include <cyg/hal/hal_cache.h>
#include <cyg/hal/hal_tables.h>
#include <cyg/kernel/kapi.h>
#include <cyg/error/codes.h>
#include <cyg/io/io.h>
#include <cyg/io/serialio.h>
#include <cyg/io/file.h>
#include <net/etherio.h>
#include <stdlib.h>
#include <network.h>
#include <stdio.h>
#include <errno.h>

#include "../dm.h"
#include "../ident.h"

extern int select(int, fd_set *, fd_set *, fd_set *, struct timeval *tv);
extern int cyg_select_with_abort(int, fd_set *, fd_set *, fd_set *, struct timeval *tv);
extern void cyg_select_abort(void);
extern int close(int);

//==========================================================================
// A sample extenstion to DM SocketServer
//==========================================================================

// Notice that this file exposes no symbol names.  Everything is static.

// product ID info (we don't use it in this module, but you might want to
extern tIdent ident;

// state for our extension threads -- we're only going to have
// two, but you can have more if you like.

static cyg_thread_entry_t echoTask;
static unsigned char echoStack[4096];
static cyg_handle_t  echoHandle;
static cyg_thread    echoThread;

static cyg_thread_entry_t bgTask;
static unsigned char bgStack[4096];
static cyg_handle_t  bgHandle;
static cyg_thread    bgThread;

// global flag set by TCP/IP stack init routine, in case you
// want to know whether TCP/IP network is up or not

extern int TcpStackOk;

// declare the functions that run as tasks
```

Note that the **diag_printf**() function is a non-blocking, busy-wait serial output routine. If it is called simultaneously from multiple threads, the output data stream may be interleaved. This is may seem inconvenient, but it means that it can be called from anywhere. Attempting to make it an atomic operation would restrict it's use to threads that run after the scheduler has started.

```
// diag printf routine that prints messages to internal UART
extern void diag_printf(const char *fmt, ...);
```

Since the following routine is called before the scheduler is running, you can not use all of the available kernel API calls. In general, calls to create or initialize kernel objects are okay. Calls that might block probably are not.

```
// initialization routine called before scheduler is running

static void extensionInit0(void)
{
 diag_printf("ExtensionInit0()\n");
  cyg_thread_create(8, echoTask, (cyg_addrword_t)0, "Extension - Echo",
                    (void *)echoStack,
                    sizeof echoStack,
                    &echoHandle,
                    &echoThread);
  cyg_thread_resume(echoHandle);

  cyg_thread_create(8, bgTask, (cyg_addrword_t)0, "Extension - Bg",
                    (void *)bgStack,
                    sizeof bgStack,
                    &bgHandle,
                    &bgThread);
  cyg_thread_resume(bgHandle);
}
```

The following routine is called after the scheduler is running and the network has been initialized. You can use any of the eCos kernel API calls -- however, remember that there may be other modules whose **init1** routines are waiting to be called. **Init** routines are called one at a time, and if you block in this routine it may prevent or delay other modules' initialization.

```
// initialization routine called after scheduler is running
// and network initialization has completed (either with
// success or failure)

static void extensionInit1(void)
{
  diag_printf("extensionInit1()\n");
  diag_printf("TCP stack OK = %d\n",TcpStackOk);
}

// Put pointers to init routines into init tables using linker
// trick. They will get called by main.c.  Notice that the
// functions do _not_ need to be globally visible.

DeviceMaster_init0(extensionInit0,9999);
DeviceMaster_init1(extensionInit1,9999);

// someplace to go when we decide to give up
static void done(void)
{
  diag_printf("extension done!\n");
  while (1)
    cyg_thread_delay(100);
}

// count of bytes echoed

static unsigned totalBytes;
```

Here is the function that runs as it's own thread. You can block all you want in this routine and it will not stop anybody else from running.

```
// this is a simple thread that echos data on a serial port

static void echoTask(cyg_addrword_t data)
{
  unsigned char buf[512];
  cyg_uint32 len;
  Cyg_ErrNo status;
  cyg_io_handle_t ioHandle;
  cyg_serial_info_t serConfig;
```

```
diag_printf("echoTask Beginning execution\n");

// acquire ownership of port

cyg_semaphore_wait(&devMastPort[0].ownership);

// The socket server code may be currently blocked in
// select(), and to be polite, we should wake it up so that
// it can shut down listening sockets if it wants to.

cyg_select_abort();


diag_printf("echoTask acquired port 0\n");

// configure port

ioHandle = devMastPort[0].handle;

len = sizeof serConfig;
status = cyg_io_get_config(ioHandle, CYG_IO_GET_CONFIG_SERIAL_INFO, &serConfig, &len);

if (status != ENOERR)
  {
    diag_printf("ERROR, cyg_io_get_config returned %d: %s\n",status,strerror(status));
    done();
  }

serConfig.flags &= ~CYG_SERIAL_FLAGS_RTSCTS;
serConfig.flags &= ~CYG_SERIAL_FLAGS_XONXOFF_RXFLOW;
serConfig.flags &= ~CYG_SERIAL_FLAGS_XONXOFF_TXFLOW;

serConfig.flags = CYG_SERIAL_FLAGS_RD_SEMIBLOCK;

serConfig.baud = 9600;
serConfig.stop = CYGNUM_SERIAL_STOP_1;
serConfig.parity = CYGNUM_SERIAL_PARITY_NONE;
serConfig.word_length = CYGNUM_SERIAL_WORD_LENGTH_8;

len = sizeof serConfig;
status = cyg_io_set_config(ioHandle, CYG_IO_SET_CONFIG_SERIAL_INFO, &serConfig, &len);

if (status != ENOERR)
  {
    diag_printf("ERROR, cyg_io_get_config returned %d: %s\n",status,strerror(status));
    done();
  }

// set port interface mode to RS-232

devMastSetPortMode(0,DevMastRsMode_232);
```

Notice that the following loop never exits. If something goes wrong and it decides to go belly-up it just calls **done**(), which prints a message and then goes into a delay loop.

```
    // echo data

    for (;;)
      {
        len = sizeof buf;
        status = cyg_io_read(ioHandle, buf, &len);

        // diag_printf("%d[%d]\n",portNum,len);

        // semi-blocking read shouldn't return with len of 0,
        // but let's check just the same

        if (len == 0)
          continue;

        totalBytes += len;

        if (status != ENOERR && status != -EAGAIN)
          {
            diag_printf("ERROR, cyg_io_read returned %d: %s\n",status,strerror(status));
            done();
          }

        status = cyg_io_write(ioHandle, buf, &len);

        if (status != ENOERR)
          {
            diag_printf("ERROR, cyg_io_write returned %d: %s\n",status,strerror(status));
            done();
          }
      }
  }
```

A second thread that does nothing but print a message once per second that says how may bytes we have echoed.

```
  static void bgTask(cyg_addrword_t data)
  {
    cyg_thread_delay(300);  // wait a while...
    diag_printf("Extension background task running\n");
    while (1)
      {
        cyg_thread_delay(100);  // 1 second delay
        diag_printf("Bytes transfered: %d\n",totalBytes);
      }
  }
```

Modifying the SocketServer Data Stream

This section is for users wish to use the SocketServer application more-or-less as-is but wish to add code to process the serial data stream as it goes by.

Each port has a two threads associated with it. One thread handles data being transferred from the network to the serial port, and other handles data flowing in the other direction. Both of these routines are in the file **socketServer/server.c**.

These functions are instantiated as tasks once for each serial port on the system. Each instance receives a parameter pointing to a status structure that contains pertinent instance data: **tServerState** structure. Each pair of the instances of the data transfer functions has a private copy that pertains to their serial port.

The transmit and receive functions for port N share an instance structure, but that structure is unique to Port N (and thus to that pair of transmit and receive function instances).

Some fields of interest in that structure are:

state

State of this port. If it's **ServerBusy**, then we're supposed to be transferring any data we get data.

serverNumber

Which port number we are attached to: 0-N socket. The TCP socket associated with the serial port.

idleTimer

An **idle-timout** counter that is incremented once per second by somebody else.

rxCount

Count of data bytes transferred from the network socket.

txCount

Count of data bytes transferred from the serial port

Receive Data

The function that handles data being transferred from the network to the serial port is called **TcpRxTask**():

```
/*
 * Tcp receive task: transfer data from socket to serial port
 */

static void TcpRxTask(cyg_addrword_t data)
{
  int n;
  char rxDataBuf[1024];
  char *p;
  Cyg_ErrNo status;
  tServerState *s = (tServerState*)data;
  static int rxByes;
```

The pointer "**s**" above, is a pointer to the instance structure we describe previously.

```
    while (1)
      {
        while (s->state == ServerBusy)
          {
            // transfer data from TCP to serial port.
            n = read(s->socket, rxDataBuf, sizeof rxDataBuf);
            rxBytes t=n;
            rw_diag_printf_m("TcpRx%d: rd %d\n",s->serverNumber, n);

            if (s->state != ServerBusy)
              break;

            if (n==0)
              {
                // Connection is gone
                diag_printf_m("TcpRx%d: connection closed\n",s->serverNumber);
                s->state = ServerClosing;
              rxBytes = 0
              }
            else if (n<0)
              {
                // read() error
                diag_printf_m("TcpRx%d: read(%d)==%d  error: %d\n",s->serverNumber,
                                          s->socket,n,errno);
                s->state = ServerClosing;
              }
            else
              {
                // write data to serial port
                s->idleTimer = 0;
                s->rxCount += n;
```

At this point, we have received data from the network, but have yet to send it out the serial port. There are "**n**" bytes of data in **rxDataBuf**. If you want to do something to the data, here is the place to do it.

```
                p = rxDataBuf;
                while (n>0)
                  {
                    cyg_uint32 len = n;
                    status = cyg_io_write(s->serialHandle,p,&len);
                    rw_diag_printf_m("TcpRx%d: wr %d\n", s->serverNumber, len);
                    if (status == -EINTR)
                        break; //somebody aborted the read/write
                    if (status != ENOERR)
                      {
                        diag_printf_m("TcpRx%d: error writing to serial port: %d: %s\n",
                                                    s->serverNumber, status, strerror(status));
                        cyg_thread_delay(20);
                      }
                    n -= len;
                    p += len;
                  }
              }
          }
        diag_printf_m("TcpRx%d: waiting on semaphore\n", s->serverNumber);
        cyg_semaphore_wait(&s->socketSemaphore);
        diag_printf_m("TcpRx%d: awake socket=%d\n", s->serverNumber, s->socket);
      }
}
```

Transmit Data

The second function which transfers data in the other direction (serial port -> TCP) is **TcpTxTask**(). It is significantly more complicated than the above function because it also contains the logic to start up and shut down the connection between a serial port and a TCP socket.

```
/*
 * Tcp transmit task: transfers data from serial port to Tcp connection.
 */

static void TcpTxTask(cyg_addrword_t data)
{
  tServerState *s = (tServerState*) data;
  unsigned char txDataBuf[1024];
  unsigned txBufCount = 0;
  unsigned char *p;
  cyg_uint32 len;
  Cyg_ErrNo status;
  int semVal;
  int n;
  unsigned loops=0;
  static int txBytes;

  // stagger thread start-up just for fun

  cyg_thread_delay(11*s->serverNumber);

  diag_printf_m("TcpTx%d started\n",s->serverNumber);

  txDataBuf = malloc(TxBufSize);

  if (!txDataBuf);
  {
       diag_printf(“ACK! malloc failed: %s %d\n”,_FILE_,_LINE_);
       return;
  }

  while(1)
     {
      // delay for a bit -- so that we don’t swamp the net with tiny packets.

      ++loops;
      cyg_thread_delay(5);
```

Reducing the delay value above will reduce data latency but increase network and processor overhead by sending smaller, more frequent packets out onto the network.

Most of the stuff in the **if**() statement below is deciding whether or not we need to initiate a TCP connection.

```
          if (s->config.enabled && s->state == ServerIdle)
            {
              /* check to see if we should initiate a connection */
              int initiate = 0;

              cyg_semaphore_peek(s->ownership, &semVal);

              if (semVal)
                {
                  if (s->configUpdated)
                    initSerialPort(s->serverNumber);

                  if (s->config.connectAlways)
                    initiate = 1;

                  if (s->config.connectOnData)
                    {
                      cyg_serial_buf_info_t serbufinfo;
                      len = sizeof serbufinfo;
                      status = cyg_io_get_config(s->serialHandle,
                                                 CYG_IO_GET_CONFIG_SERIAL_BUFFER_INFO,
                                                 &serbufinfo, &len);
                   //diag_printf("TcpTx%d rx_count=%d\n",s->serverNumber, serbufinfo.rx_count);
                      if (status != ENOERR)
                        diag_printf("TcpTx%d serial get buffer info error %d\n",
                                                      s->serverNumber, status);
                      else if (serbufinfo.rx_count)
                        initiate = 1;
                    }

                  if (s->config.connectOnCD || s->config.connectOnDSR)
                    {
                      int msr;
                      len = sizeof msr;
                   status = cyg_io_get_config(s->serialHandle,CYG_IO_GET_CONFIG_SERIAL_MODEM,
                                              &msr, &len);
                      if ((loops%50) == 0)
                   diag_printf("TcpTx%d serial get modem status %04x\n",s->serverNumber, msr);
                      if (status != ENOERR)
                        diag_printf("TcpTx%d serial get modem status error %d\n",
                                                       s->serverNumber, status);
                      else if (((msr & CYG_SERIAL_MODEM_CD) && s->config.connectOnCD) ||
                               ((msr & CYG_SERIAL_MODEM_DSR) && s->config.connectOnDSR))
                        initiate = 1;
                    }
                }
Here's where we initiate a connection if we have decided we want to.
              if (initiate)
                {
                  int sock;
                  struct sockaddr_in dest;

                  memset(&dest, 0, sizeof(dest));
                  dest.sin_family = AF_INET;
                  dest.sin_port = htons(s->config.connectToPort);
                  dest.sin_addr.s_addr = htonl(s->config.connectToIP);

                  sock = socket(PF_INET, SOCK_STREAM, 0);

                  if (sock < 0)
                    {
                      diag_printf("TcpTx%d: socket() failed: %d\n",s->serverNumber,errno);
                      cyg_thread_delay(100);
                      continue;
                    }
```

```
                if (connect(sock,(struct sockaddr *)&dest,sizeof dest) < 0)
                  {
                    diag_printf("TcpTx%d: connect() failed: %d\n",s->serverNumber,errno);
                    close(sock);
                    cyg_thread_delay(100);
                    continue;
                  }

                diag_printf("TcpTx%d: initiating\n",s->serverNumber);

                cyg_mutex_lock(&s->serverMutex);
                if (s->state == ServerIdle && cyg_semaphore_trywait(s->ownership))
                  connectionUp(s,sock, s->config.connectToIP,s->config.connectToPort,0,0);
                else
                  {
                    diag_printf("TcpTx%d: acquire failure -- closing
                                          %d\n",s->serverNumber,sock);
                    close(sock);
                  }
                cyg_mutex_unlock(&s->serverMutex);
              }
          }
```

Now we're going to decide if we need to shut down an existing connection.

```
          if (s->state == ServerBusy && s->config.disconnOnIdle &&
              (s->idleTimer > s->config.idleTimeout))
            {
              diag_printf("TcpTx%d: idle timeout\n",s->serverNumber);
              s->state = ServerClosing;
            }

         if (s->state == ServerBusy && (s->config.disconnOnNoCD || s->config.disconnOnNoDSR))
            {
              int msr;
              len = sizeof msr;
              status = cyg_io_get_config(s->serialHandle,CYG_IO_GET_CONFIG_SERIAL_MODEM,
                                                         &msr, &len);
    #if 0
              if ((loops%50) == 0)
                diag_printf("TcpTx%d serial get modem status %04x\n",s->serverNumber, msr);
    #endif
              if (status != ENOERR)
                diag_printf("TcpTx%d serial get modem status error %d\n",
                                        s->serverNumber, status);
              else if (((~msr & CYG_SERIAL_MODEM_CD) && s->config.disconnOnNoCD) ||
                       ((~msr & CYG_SERIAL_MODEM_DSR) && s->config.disconnOnNoDSR))
                s->state = ServerClosing;
            }

          if (s->state==ServerBusy && SocketIsClosed(s->socket))
            {
              diag_printf("TcpTx%d: socket closed by remote host\n",s->serverNumber);
              s->state = ServerClosing;
            }
          if (s->state == ServerBusy && !s->config.enabled)
            s->state = ServerClosing;
```

And here is where we actually shut down the connection.

```
if (s->state==ServerClosing)
  {
    cyg_mutex_lock(&s->serverMutex);

    diag_printf("TcpTx%d: closing %d\n",s->serverNumber,s->socket);
    close(s->socket);
    s->socket = -1;
    s->locIP = 0;
    s->locPort = 0;
    s->remIP = 0;
    s->remPort = 0;
    s->idleTimer = 0;
    s->state = ServerIdle;
    cyg_io_get_config(s->serialHandle, CYG_IO_GET_CONFIG_SERIAL_INPUT_FLUSH, NULL, 0);
    cyg_io_get_config(s->serialHandle, CYG_IO_GET_CONFIG_SERIAL_OUTPUT_FLUSH, NULL, 0);
    if (s->config.dtrControl == DTRon)
      portDtr(s->serialHandle,1);
    else
      portDtr(s->serialHandle,0);
    cyg_mutex_unlock(&s->serverMutex);
    cyg_semaphore_post(s->ownership);
    cyg_select_abort(); // wake up the accept task
  }

// read data from serial port and shove into TCP
// connection until there is none left

while (s->state==ServerBusy && s->config.enabled)
  {
    if (s->configUpdated)
      initSerialPort(s->serverNumber);

    len = TxBufSize - txBufCount;
    status = cyg_io_read(s->serialHandle, txDataBuf, &len);

    if (status != ENOERR &&
        status != -EAGAIN &&
        status != -EEOL &&
        status != -ETIMDOUT &&
        status != -EINTERCHARTIMEOUT)
    {
        diag_printf_m("TcpTx%d: cyg_io_read of %08x returned %d:%s\n",
                      s->serverNumber, s->serialHandle, status, strerror(-status));
        cyg_thread_delay(10);
    }

    if (status ==-EAGAIN && len == 0
        break;

    if (len)
        rw_diag_printf_m("TcpTx5d: rd %d (%d)\n", s->serverNumber, len, status);

    txBufCount += len;
    s->txCount += len;
```

At this point we've read some data from the serial port but have not written it to the TCP socket. There are **len** bytes of data in **txDataBuf**. If you want to manipulate the data stream, this is the place to do it.

```
    s->idleTimer = 0;

       // If we're waiting for IC Timeout and timed
    // out waiting for one, then leave data in buffer.
    if (status == -ETIMEDOUT)
    break;

    p = txDataBuf;

    while (txBufCount)
      {
        n = write(s->socket,p,len);
```

```
                if (n<0)
                  {
                    diag_printf_m("TcpTx%d: write() returned %d: %s\n",
                                                  s->serverNumber,n,strerror(n));
                    if (s->state == ServerBusy)
                      s->state = ServerClosing;
                    break;
                  }
                  txBytes += n;
                rw_diag_printf_m("TcpTx%d: wr %d total %d\n", s->serverNumber, n, txBytes);
                p += n;
                txBufCount -= n;
              }
          }
      }
  }
```

GoAhead Web Server

The SocketServer program uses the GoAhead web server, version 2.1.3, with a few bug fixes. These bug fixes have been submitted to GoAhead, but have not yet been incorporated into the GoAhead distribution. The goahead directory contains the fixed GoAhead distribution (except for the web pages). The **makefile** in the ECOS directory has been modified to build only the webserver library, **libwebs.a**, and not build the demo eCos application.

The webserver itself is run by a thread in **webserv/webserv.c**. This file contains much of the same functionality that is present in the demonstration eCos **app** that is provided with the GoAhead distribution.

The custom JavaScript/ASP interface and the forms handling functions are in the file **webserv/websupp.c**.

For documentation on the JavaScript/ASP interface, see the GoAhead documentation at http://webserver.goahead.com/webserver/webserver.htm.

The source files for the web pages themselves are in the **webpages/root** directory. These pages are combined into a single data structure contained in the file **webrom.c** by the **webcomp** program. The **webrom.c** file is then compiled and linked in using the normal module **Makefile** methods.

Trademark Notices

Second Edition, June 30, 2004

Document Number: 2000222 Rev. B