DeviceMaster® Ethernet Device Driver

This document describes the **dmrts** Ethernet device driver for eCos. The driver supports the Ethernet controller built in to the Samsung KS32C5000A and S3C4510 ARM micro-controllers.

The driver provides two interfaces. The first interface is to the eCos network stack (IP/ARP/ICMP/etc.). This interface is not visible to the application programmer. The normal BSD socket API is used at the application level: **socket()**, **ioctl()**, **send()**, **recv()**, **select()**, **close()**.

The second interface provided by the driver is a MAC layer interface that can be used to send and receive raw Ethernet frames. This interface corresponds to *raw* or *packet* sockets on Linux and to the packet-filter API under BSD.

The raw device interface implimented by the dmrts driver uses the standard eCos I/O device API. If you are unfamiliar with the eCos I/O device API, see the eCos Reference Manual, Part IV: I/O Package (Device Drivers).

dmrts Raw Frame API

The raw frame API consists of the five standard eCos system calls:

- **cyg_io_lookup()**
- **cyg_io_read()**
- **cyg_io_write()**
- **cyg_io_get_config()**
- **cyg_io_set_config()**

If you are unfamiliar with these calls please see the *User API* section of the eCos Reference Manual, Part IV: I/O Package (Device Drivers).

Ethernet Frame Buffers

Internally, the Samsung Ethernet hardware and this driver use a pool of frame buffers. Each buffer can contain exactly one Ethernet frame. If the user uses the standard **cyg_io_read()** and **cyg_io_write()** calls, the allocation and deallocation of frame buffers is handled automatically and the programmer does not have to be concerned with them. However, using the **cyg_io_read()** and **cyg_io_write()** calls will result in an extra memory-to-memory copy operation when the data is copied to/from the Ethernet frame buffers.

In performance-critical situations it is possible to eliminate this copy operation by using the frame buffers in the application code. To do this, the **cyg_io_read()** and **cyg_io_write()** calls are not used, and a set of calls to the **cyg_io_get_config()** are used to:

- Allocate a frame buffer
- Send a frame buffer
- Receive a frame buffer
- Free a frame buffer

These four operations are described in more detail below.

The Ethernet frame buffer structure is declared in **/net/etherio.h** as the **typedef tEthBuffer**:

```
//-------------------------------------------------------------
// Buffer structure passed back and forth between userspace and
// driver to avoid copying data.
//
// data is first so that we can typecast back/forth between BDMA
// data buffer pointers and pointers to this struct.  This means
// that it's important not to write off the end of the data area.

#define MAX_ETH_FRAME_SIZE 1520

typedef struct tEthBufferTag
```

```
{
  unsigned char data[MAX_ETH_FRAME_SIZE+8];
  unsigned length;
  unsigned userData;
  struct tEthBufferTag *next;
  struct tEthBufferTag *prev;
}tEthBuffer;
```

The **length** field is set by the driver for receive frames and must be set by the user for transmit frames.

The user may use the **userData**, **next**, and **prev** fields as desired. The contents of those fields is undefined when a buffer is passed to the user from the driver, and is ignored when a buffer is passed from the user to the driver.

The **data** field in receive frames contains two bytes of initial padding plus two bytes of checksum information. This is ostensibly done to improve efficiency of data handling -- the ethernet data payload will start at a word boundary. It's supposed to work without the padding. However, some versions of the Samsung parts seemed to have problems without the padding, so it's the default configuration for the driver:

data byte	Description
0-1	Padding (unused)
2-7	Dest Addr
8-13	Src Addr
14-15	Protocol/Length
16-N	Data

cyg_io_lookup()

The device name supported by the raw frame driver interface is **/dev/reth0**.

cyg_io_set_config()

The following keys are supported for the **cyg_io_set_config()** call:

CYG_IO_SET_CONFIG_ETHER_INFO

Sets the configuration of Ethernet driver. The configuration data is passed as a pointer to the following structure.

```
typedef struct
{
  cyg_uint8    mac_address[6];
  cyg_uint16   protocol;
  cyg_uint32   flags;
} cyg_ether_info_t;
```

The fields are defined below

mac_address The Ethernet address to be configured into the Ethernet controller. If the standard DeviceMaster boot loader is being used, it will have configured an Ethernet address and this address will be present in this field when configuration data is read using the **cyg_io_get_config()** call. To change the Ethernet address, change this field and pass the structure back using the **cyg_io_set_config()** call.

protocol If non-zero, determines the Ethernet protocol number which will be received by the raw frame I/O interface. Only frames received with this protocol number will be handled by the raw frame API. Other frame types will either be passed to the eCos networking stack or discarded. If set to **0xffff**, all received frames will be handled by the raw frame API (no frames will be passed to the eCos networking stack).

The default value of this field is 0: this means that no frames will be received by the raw frame interface unless this field is changed.

Frames may be transmitted at any time with any protocol value regardless of the value of this field.

flags Configuration flag bits:

CYG_ETHER_FLAGS_DISABLE_AUTONEG
Disable physical layer autonegotiation if manual onfiguration is supported by hardware platform. Not supported by DeviceMaster RTS.

CYG_ETHER_FLAGS_DISABLE_100M
Disable 100Mbit operation -- forces 10Mbit operation if manual configuration is supported by hardware platform. Not supported by DeviceMaster RTS.

CYG_ETHER_FLAGS_DISABLE_FULLDUPLEX
Disable full-duplex operation (forces half-duplex operation) if manual configuration is supported by hardware platform. Not supported by DeviceMaster RTS.

CYG_ETHER_FLAGS_RD_NONBLOCK
Enable non-blocking read mode for the **cyg_io_read()** call and for the **CYG_IO_GET_CONFIG_ETHER_RECV_BUFFER** operation.

CYG_ETHER_FLAGS_WR_NONBLOCK
Enable non-blocking write mode for the **cyg_io_write()** call and for the **CYG_IO_GET_CONFIG_ETHER_SEND_BUFFER** operation.

CYG_ETHER_FLAGS_PROMISCUOUS
Place Ethernet controller in promiscuous receive mode. Not currently supported by this driver.

cyg_io_read()

Transfers a received frame into the user buffer. The padding bytes mentioned in the Ethernet frame description are discarded before the data is copied into the user buffer.

Reads may be blocking or non-blocking.

blocking The default state for the driver is blocking reads. When a call is made to **cyg_io_read()**, the call will not return until a frame has been transferred to the user's buffer. The call will wait indefinitely for a frame to be received.

Each **cyg_io_read()** call will dequeue one received Ethernet frame. If the user data buffer is not large enough to receive the entire frame, any extra data bytes are discarded.

non-blocking The call to **cyg_io_read()** will return immediately. If a received frame is available, it will be transferred to the user's buffer and the returned length will reflect the number of bytes actually transferred. If no receive frame is available, no bytes will be transferred and the call will return -**EAGAIN**.

cyg_io_write()

Writes may be blocking or non-blocking.

blocking The call to **cyg_io_write()** will return as soon as the requested data has been transferred to a frame buffer and placed in the transmit queue. If there is insufficient room in the buffer, the call will block indefinitely until room becomes available.

Each call to **cyg_io_write()** will queue a single Ethernet frame for transmission. If the data length is invalid (less than 14 or larger than 1540), an error status is returned and no frame is queued.

non-blocking The call to **cyg_io_write()** will return immediately. If there was no room in the trasmit queue (or if no frame buffers were available), the length will return 0, and the status will return -**EAGAIN**.

cyg_io_get_config()

The following keys are supported for the **cyg_io_get_config()** call:

CYG_IO_GET_CONFIG_ETHER_INFO

Gets the current driver configuration -- see the set_config() section for a description of the configuration data structure.

CYG_IO_GET_CONFIG_ETHER_LINK_STATUS

** Not supported by DeviceMaster RTS **

If physical link status is supported by the hardware platform, return the link status as an integer composed of a bitwise "or" of the following flags:

CYG_ETHER_LINK_STATUS_UP Set if the link is up.

CYG_ETHER_LINK_STATUS_FULL_DUPLEX Set if the link is operating in full-duplex mode.

CYG_ETHER_LINK_STATUS_100M Set if the link is operating in 100Mbit mode.

The four keys below all expect the "buf" parameter to be of the type ***tEthBuffer**, and the **len** parameter is ignored.

CYG_IO_GET_CONFIG_ETHER_ALLOC_BUFFER
Allocates a frame buffer and writes a pointer to the address passed in the **cyg_io_get_config()**. Frames allocated with this call must be deallocated by passing them back to the driver with either the **FREE_BUFFER** key OR the **SEND_BUFFER** key but not both.

CYG_IO_GET_CONFIG_ETHER_FREE_BUFFER
Frees a frame buffer that was obtained by the user via the **ALLOC_BUFFER** key or via the **RECV_BUFFER** key. All frames obtained by the user by either method must be deallocated via either the **FREE_BUFFER** or the **SEND_BUFFER** operations.

CYG_IO_GET_CONFIG_ETHER_SEND_BUFFER
Places the passed frame buffer in the transmit queue. The user must not modify the frame buffer or perform a **FREE_BUFFER** operation on a frame buffer that has been placed in the transmit queue using this operation.

The frame buffer will be automatically deallocated after it has been transmitted.

A **SEND_BUFFER** operation may be blocking or non-blocking.

If the driver is configured for blocking writes, the call will block indefinitely until there is room in the transmit queue and the frame has been queued.

If the driver is configured for non-blocking writes and no room is available in the transmit queue, the call will immediately return -**EAGAIN**. If that happens, the frame buffer is still owned by the user and must either be sent again with the **SEND_BUFFER** operation or deallocated using the **FREE_BUFFER** key.

CYG_IO_GET_CONFIG_ETHER_RECV_BUFFER
Returns a received frame buffer to the user by writing the address of the recieved frame to the address passed in the **cyg_io_get_config()** call. Frames obtained in this manner must be deallocated by passing them back to the driver with either the **FREE_BUFFER** key OR the **SEND_BUFFER** key but not both.

Receive frames contain two bytes of initial padding. This is done to improve efficiency of data handling -- the ethernet data payload will start at a word boundary:

Byte	Description
0-1	Padding (unused)
2-7	Dest Addr
8-13	Src Addr
14-15	Protocol/Length
16-xx	Data

A **RECV_BUFFER** operation may be blocking or non-blocking.

If the driver is configured for blocking reads, the call will block indefinitely until a received frame is available.

If the driver is configured for non-blocking reads and no receive frames are available, the call will immediately return -**EAGAIN** and no frame will be passed to the user.

Example Program

The sample program below echos data received with Ethernet protocol **0x5432**. It demonstrates the usage of both the regular **cyg_io_read()** and **cyg_io_write()** calls as well as the use of the Ethernet frame buffer method that avoids an additional memory-to-memory copy operation in the driver.

```
 1  #include <cyg/kernel/kapi.h>
 2  #include <cyg/error/codes.h>
 3  #include <cyg/io/io.h>
 4  #include <net/etherio.h>
 5  #include <stdio.h>
 6  #include <stdlib.h>
 7  #include <network.h>
 8
```

The following is used to insure that output from different calls to **diag_printf_m()** don't get interlaced in the diag UART output data stream. The **diag_printf()** function is a very low level, busy-wait, non-buffered, non-atomic output routine:

```
 9
10  // Stuff to impliment atomic diagnostic message output
11
12  // printf routine that prints messages to KS32C5000 UART
13  extern void diag_printf(const char *fmt, ...);
14
15  // atomic diag_printf operation -- only use in running tasks,
16  // not in initialization code, DSR or ISR code.
17
18  #define UseDiagPrintfMutex 1
19
20  #if UseDiagPrintfMutex
21  static cyg_mutex_t dpMutex;
22  #define diag_printf_m(fmt, args...) \
23    do { \
24         cyg_mutex_lock(&dpMutex); \
25         diag_printf(fmt, ##args); \
26         cyg_mutex_unlock(&dpMutex); \
27       } while (0)
```

```
28  #define diag_init()  cyg_mutex_init(&dpMutex)
29  #define diag_lock()  cyg_mutex_lock(&dpMutex)
30  #define diag_unlock() cyg_mutex_unlock(&dpMutex)
31  #else
32  #define diag_printf_m(fmt, args...) diag_printf(fmt, ##args)
33  #define diag_init()  /* noop */
34  #define diag_lock()  /* noop */
35  #define diag_unlock() /* noop */
36  #endif
37
38
39  typedef unsigned char tStack[4096];
40
```

We're going to have two threads: one that echos data on the Ethernet interface, and another that prints status messages once per second. Each thread needs a thread object, a thread handle, and a stack:

```
41  cyg_thread   rawEchoThread, backgroundThread;
42  tStack       rawEchoStack, backgroundStack;
43  cyg_handle_t rawEchoHandle, backgroundHandle;
44  cyg_thread_entry_t rawEchoTask, backgroundTask;
45
46  // Here is where user execution starts
47
48  void cyg_user_start(void)
49  {
50    diag_printf("Entering cyg_user_start() function\n");
51
52    diag_init();
53
```

Create the threads with cyg_thread_create() and mark them as runnable using the cyg_thread_resume():

```
54    cyg_thread_create(5, rawEchoTask, (cyg_addrword_t) -1,
55      "raw echo thread", rawEchoStack, sizeof rawEchoStack,
56                      &rawEchoHandle,&rawEchoThread);
57    cyg_thread_resume(rawEchoHandle);
58
59    cyg_thread_create(6, backgroundTask, (cyg_addrword_t) -1,
60      "background thread", backgroundStack, sizeof backgroundStack,
61                      &backgroundHandle,&backgroundThread);
62    cyg_thread_resume(backgroundHandle);
63
64    // returning from this function starts scheduler
65  }
66
67  void done(void)
68  {
69    for (;;)
70      ;
71  }
72
```

```
 73  /* count of bytes echoed */
 74
 75  unsigned totalBytes;
 76
 77  #define UseCygIoReadWrite 1
 78
 79  #define protoNumber 0x5432
 80
 81  // this is a simple thread that echos data on a raw Ethernet
 82  // connection using the cyg_io_read() and cyg_io_write() calls
 83
 84  void rawEchoTask(cyg_addrword_t data)
 85  {
 86    Cyg_ErrNo status;
 87    cyg_io_handle_t ethHandle;
 88    cyg_ether_info_t ethConfig;
 89    cyg_uint32 len;
 90    unsigned short netorderProto = ntohs(protoNumber);
 91
 92    diag_printf_m("Beginning execution\n");
 93
```

Lookup the raw Ethernet device. It's name is hard-wired into the driver -- and only a single device instance is supported:

```
 94    status = cyg_io_lookup("/dev/reth0", &ethHandle);
 95
 96    if (status != ENOERR)
 97      {
 98       diag_printf_m("ERROR, cyg_io_lookup returned %d: %s\n",status,strerror(status));
 99        done();
100      }
101
```

Get the current configuration. The bootloader will have already set the MAC address in the controller chip, and the driver will have read that address from the chip when it was initialized.

```
102    len = sizeof ethConfig;
103    status = cyg_io_get_config(ethHandle, CYG_IO_GET_CONFIG_ETHER_INFO,
                                             &ethConfig, &len);
104
105    if (status != ENOERR)
106      {
107        diag_printf_m("ERROR, cyg_io_get_config returned %d:
                                %s\n",status,strerror(status));
108        done();
109      }
```

```
110
111     diag_printf_m("MAC addr %02x:%02x:%02x:%02x:%02x:%02x\n",
112                   ethConfig.mac_address[0],
113                   ethConfig.mac_address[1],
114                   ethConfig.mac_address[2],
115                   ethConfig.mac_address[3],
116                   ethConfig.mac_address[4],
117                   ethConfig.mac_address[5]);
118
```

Set the Ethernet protocol number for which we want to see frames, then write the configuration info back:

```
119     ethConfig.protocol = netorderProto;
120
121     len = sizeof ethConfig;
122     status = cyg_io_set_config(ethHandle, CYG_IO_SET_CONFIG_ETHER_INFO,
                                               &ethConfig, &len);
123
124     if (status != ENOERR)
125       {
126         diag_printf_m("ERROR, cyg_io_set_config returned %d:
                                  %s\n",status,strerror(status));
127         done();
128       }
129
130   #if UseCygIoReadWrite
131
```

This **for()** loop uses the normal **cyg_io_read()** and **cyg_io_write()** calls. This is more in keeping with other eCos driver usage, but it requires that the driver copy the data between the user buffers and the driver's internal frame buffers.

```
132     for (;;)
133       {
134         static unsigned char rxBuf[2048];
135         static unsigned char txBuf[2048];
136
137         len = sizeof rxBuf;
138         status = cyg_io_read(ethHandle, rxBuf, &len);
139
140         // don't forget data starts with 2 padding bytes
141
142         diag_printf_m("rx [%d]\n",len-2);
143
144         // blocking read shouldn't return with len of 0, but
145         // let's check just the same
146
147         if (len == 0)
148           continue;
149
150         len -= 2;
151
152         totalBytes += len;
```

```
153
154         if (status != ENOERR)
155           {
156            diag_printf("ERROR, cyg_io_read returned %d: %s\n",status,strerror(status));
157             done();
158           }
159
160         // swap ethernet addresses around, and copy data to tx buffer
161
162         memcpy(txBuf+0,  rxBuf+8, 6);                // dst addr
163         memcpy(txBuf+6,  ethConfig.mac_address, 6);  // src addr (ours)
164         memcpy(txBuf+12, &netorderProto, 2);          // proto
165         memcpy(txBuf+14, rxBuf+16, len-14);          // data
166
167
168         status = cyg_io_write(ethHandle, txBuf, &len);
169
170         if (status != ENOERR)
171           {
172            diag_printf("ERROR, cyg_io_write returned %d: %s\n",status,strerror(status));
173             done();
174           }
175
176         diag_printf_m("tx [%d]\n",len);
177       }
178
179   #else
180
181   #define allocPacket(h,b)
                          cyg_io_get_config(h,CYG_IO_GET_CONFIG_ETHER_ALLOC_BUFFER,b,NULL)
182   #define getRxPacket(h,b)
                      cyg_io_get_config(h,CYG_IO_GET_CONFIG_ETHER_RECV_BUFFER,b,NULL)
183   #define freePacket(h,b)
                      cyg_io_get_config(h,CYG_IO_GET_CONFIG_ETHER_FREE_BUFFER,b,NULL)
184   #define sendPacket(h,b)
                      cyg_io_get_config(h,CYG_IO_GET_CONFIG_ETHER_SEND_BUFFER,b,NULL)
185
```

This **for()** loop uses the driver's Ethernet frame buffers instead of having the data copied to/from user buffer space.

Notice that when passing a frame *__to__* the driver (to send or free the frame) you pass a pointer to the frame. When receiving a frame *__from__* the driver (receive or allocate) you pass a the address of the pointer where you want the frame address stored.

```
186     for (;;)
187       {
188         tEthBuffer *frame;
189         int i;
190
191         status = getRxPacket(ethHandle,&frame);
192
193         if (status != ENOERR)
```

```
194           {
195             diag_printf("ERROR, get packet returned %d:
                                %s\n",status,strerror(status));
196             done();
197           }
198
199         if (frame->length < 16)
200           continue;
201
202         frame->length -= 2;  // subtract 2 for padding,
                                   real data starts at frame->data+2
203
204         diag_printf_m("rx [%d]\n",frame->length);
205
206         totalBytes += frame->length;
207
208         memcpy(frame->data+0, frame->data+8, 6);           // dst addr
209         memcpy(frame->data+6, ethConfig.mac_address, 6);   // src addr (ours)
210         memcpy(frame->data+12, &netorderProto, 2);         // proto
211         // don't want to depend on overlapping memcpy() for moving data
212         for (i=14; i<frame->length; ++i)
213           frame->data[i] = frame->data[i+2];
214
215         status = sendPacket(ethHandle,frame);
216
217         if (status != ENOERR)
218           {
219             diag_printf("ERROR, sendPacket returned %d:
                                %s\n",status,strerror(status));
220             done();
221           }
222
```

Note that we re-used the **rx** frame as the **tx** frame. That way we avoid having to free the **rx** frame and **alloc** a **tx** frame.

```
223         diag_printf_m("tx [%d]\n",frame->length);
224       }
225   #endif
226   }
227
228
229   // idle thread loop count provided by eCos
230   extern volatile unsigned idle_thread_loops;
231
232   void backgroundTask(cyg_addrword_t data)
233   {
234     unsigned lastIdleThreadLoops, thisIdleThreadLoops;
235
236     lastIdleThreadLoops = idle_thread_loops;
237
```

```
238     while (1)
239       {
240         cyg_thread_delay(100);  // 1 second
241         thisIdleThreadLoops = idle_thread_loops;
242
243         diag_lock();
244         diag_printf("Bytes transfered: %d ",totalBytes);
245         diag_printf(" -- Idle Loops/Sec: %d\n",
                              thisIdleThreadLoops - lastIdleThreadLoops);
246         diag_unlock();
247
248         lastIdleThreadLoops = thisIdleThreadLoops;
249       }
250   }
```


Second Edition, June 17, 2004

Document Number: 2000240 Rev. B