RTEMS Network Supplement

Edition 4.6.0, for RTEMS 4.6.0

30 August 2003

On-Line Applications Research Corporation

On-Line Applications Research Corporation
TEXinfo 2002-11-25.11

The RTEMS Project is hosted at http://www.rtems.com. Any inquiries concerning RTEMS, its related support components, its documentation, or any custom services for RTEMS should be directed to the contacts listed on that site. A current list of RTEMS Support Providers is at http://www.rtems.com/support.html.

Table of Contents

Preface

This document describes the RTEMS specific parts of the FreeBSD TCP/IP stack. Much of this documentation was written by Eric Norum (`eric@skatter.usask.ca`) of the Saskatchewan Accelerator Laboratory who also ported the FreeBSD TCP/IP stack to RTEMS.

The following is a list of resources which should be useful in trying to understand Ethernet:

- *Charles Spurgeon's Ethernet Web Site*

  "This site provides extensive information about Ethernet (IEEE 802.3) local area network (LAN) technology. Including the original 10 Megabit per second (Mbps) system, the 100 Mbps Fast Ethernet system (802.3u), and the Gigabit Ethernet system (802.3z)." The URL is: (http://www.ethermanage.com/ethernet/ethernet.html)
- *TCP/IP Illustrated, Volume 1 : The Protocols* by by W. Richard Stevens (ISBN: 0201633469)

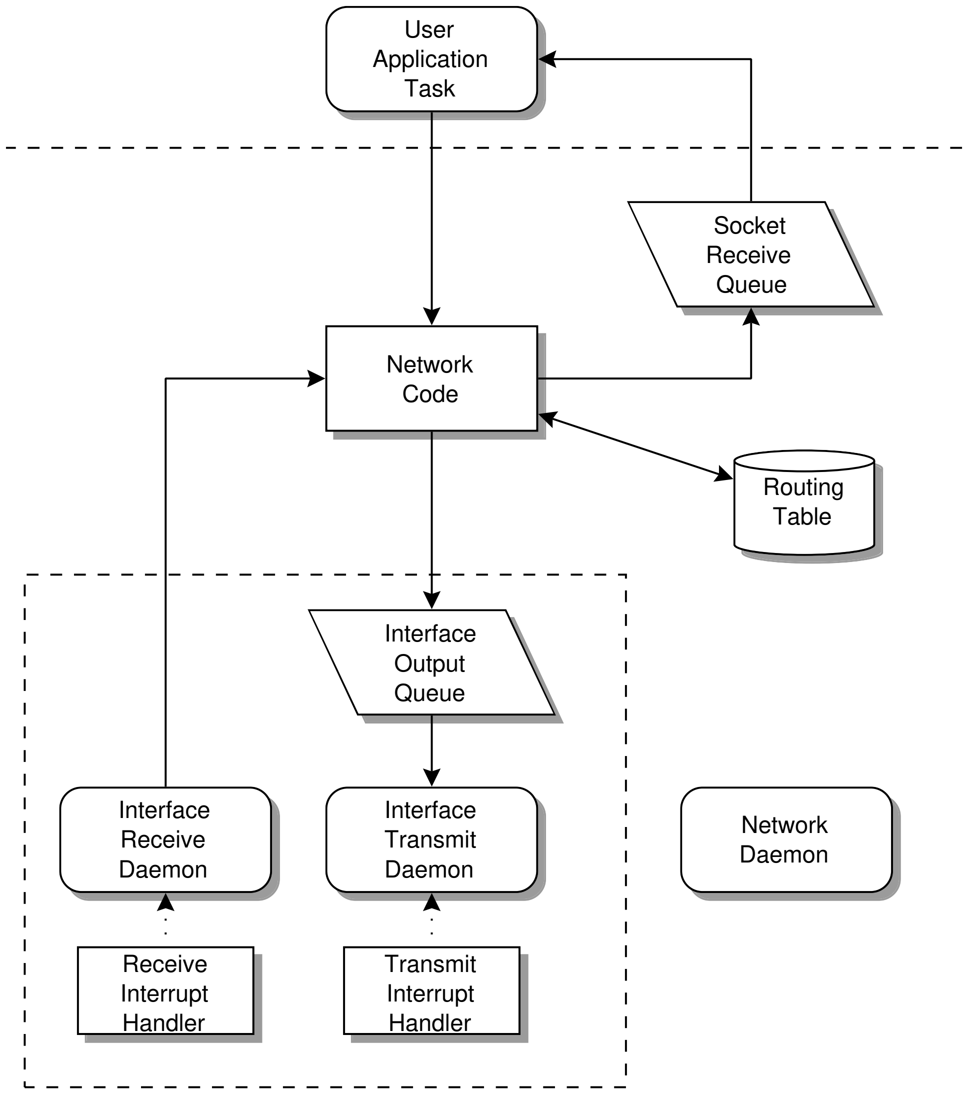
  This book provides detailed introduction to TCP/IP and includes diagnostic programs which are publicly available.
- *TCP/IP Illustrated, Volume 2 : The Implementation* by W. Richard Stevens and Gary Wright (ISBN: 020163354X)

  This book focuses on implementation issues regarding TCP/IP. The treat for RTEMS users is that the implementation covered is the BSD stack with most of the source code described in detail.
- *UNIX Network Programming, Volume 1 : 2nd Edition* by W. Richard Stevens (ISBN: 0-13-490012-X)

  This book describes how to write basic TCP/IP applications, again with primary focus on the BSD stack.

1 Network Task Structure and Data Flow

A schematic diagram of the tasks and message **mbuf** queues in a simple RTEMS networking application is shown in the following figure:



The transmit task for each network interface is normally blocked waiting for a packet to arrive in the transmit queue. Once a packet arrives, the transmit task may block waiting for an event from the transmit interrupt handler. The transmit interrupt handler sends an RTEMS event to the transmit task to indicate that transmit hardware resources have become available.

The receive task for each network interface is normally blocked waiting for an event from the receive interrupt handler. When this event is received the receive task reads the packet and forwards it to the network stack for subsequent processing by the network task.

The network task processes incoming packets and takes care of timed operations such as handling TCP timeouts and aging and removing routing table entries.

The 'Network code' contains routines which may run in the context of the user application tasks, the interface receive task or the network task. A network semaphore ensures that the data structures manipulated by the network code remain consistent.

2 Networking Driver

2.1 Introduction

This chapter is intended to provide an introduction to the procedure for writing RTEMS network device drivers. The example code is taken from the 'Generic 68360' network device driver. The source code for this driver is located in the `c/src/lib/libbsp/m68k/gen68360/network` directory in the RTEMS source code distribution. Having a copy of this driver at hand when reading the following notes will help significantly.

2.2 Learn about the network device

Before starting to write the network driver become completely familiar with the programmer's view of the device. The following points list some of the details of the device that must be understood before a driver can be written.

- Does the device use DMA to transfer packets to and from memory or does the processor have to copy packets to and from memory on the device?
- If the device uses DMA, is it capable of forming a single outgoing packet from multiple fragments scattered in separate memory buffers?
- If the device uses DMA, is it capable of chaining multiple outgoing packets, or does each outgoing packet require intervention by the driver?
- Does the device automatically pad short frames to the minimum 64 bytes or does the driver have to supply the padding?
- Does the device automatically retry a transmission on detection of a collision?
- If the device uses DMA, is it capable of buffering multiple packets to memory, or does the receiver have to be restarted after the arrival of each packet?
- How are packets that are too short, too long, or received with CRC errors handled? Does the device automatically continue reception or does the driver have to intervene?
- How is the device Ethernet address set? How is the device programmed to accept or reject broadcast and multicast packets?
- What interrupts does the device generate? Does it generate an interrupt for each incoming packet, or only for packets received without error? Does it generate an interrupt for each packet transmitted, or only when the transmit queue is empty? What happens when a transmit error is detected?

In addition, some controllers have specific questions regarding board specific configuration. For example, the SONIC Ethernet controller has a very configurable data bus interface. It can even be configured for sixteen and thirty-two bit data buses. This type of information should be obtained from the board vendor.

2.3 Understand the network scheduling conventions

When writing code for the driver transmit and receive tasks, take care to follow the network scheduling conventions. All tasks which are associated with networking share various data structures and resources. To ensure the consistency of these structures the tasks execute only when they hold the network semaphore (`rtems_bsdnet_semaphore`). The transmit and receive tasks must abide by this protocol. Be very careful to avoid 'deadly embraces' with the other network tasks. A number of routines are provided to make it easier for the network driver code to conform to the network task scheduling conventions.

- `void rtems_bsdnet_semaphore_release(void)`

  This function releases the network semaphore. The network driver tasks must call this function immediately before making any blocking RTEMS request.
- `void rtems_bsdnet_semaphore_obtain(void)`

  This function obtains the network semaphore. If a network driver task has released the network semaphore to allow other network-related tasks to run while the task blocks, then this function must be called to reobtain the semaphore immediately after the return from the blocking RTEMS request.
- `rtems_bsdnet_event_receive(rtems_event_set, rtems_option, rtems_interval, rtems_event_set *)` The network driver task should call this function when it wishes to wait for an event. This function releases the network semaphore, calls `rtems_event_receive` to wait for the specified event or events and reobtains the semaphore. The value returned is the value returned by the `rtems_event_receive`.

2.4 Network Driver Makefile

Network drivers are considered part of the BSD network package and as such are to be compiled with the appropriate flags. This can be accomplished by adding `-D__INSIDE_RTEMS_BSD_TCPIP_STACK__` to the `command line`. If the driver is inside the RTEMS source tree or is built using the RTEMS application Makefiles, then adding the following line accomplishes this:

```
DEFINES += -D__INSIDE_RTEMS_BSD_TCPIP_STACK__
```

This is equivalent to the following list of definitions. Early versions of the RTEMS BSD network stack required that all of these be defined.

```
-D_COMPILING_BSD_KERNEL_ -DKERNEL -DINET -DNFS \
  -DDIAGNOSTIC -DBOOTP_COMPAT
```

Defining these macros tells the network header files that the driver is to be compiled with extended visibility into the network stack. This is in sharp contrast to applications that simply use the network stack. Applications do not require this level of visibility and should stick to the portable application level API.

As a direct result of being logically internal to the network stack, network drivers use the BSD memory allocation routines This means, for example, that malloc takes three arguments. See the SONIC device driver (`c/src/lib/libchip/network/sonic.c`) for an example of this. Because of this, network drivers should not include `<stdlib.h>`. Doing so will result in conflicting definitions of `malloc()`.

Application level code including network servers such as the FTP daemon are **not** part of the BSD kernel network code and should not be compiled with the BSD network flags. They should include `<stdlib.h>` and not define the network stack visibility macros.

2.5 Write the Driver Attach Function

The driver attach function is responsible for configuring the driver and making the connection between the network stack and the driver.

Driver attach functions take a pointer to an `rtems_bsdnet_ifconfig` structure as their only argument. and set the driver parameters based on the values in this structure. If an entry in the configuration structure is zero the attach function chooses an appropriate default value for that parameter.

The driver should then set up several fields in the ifnet structure in the device-dependent data structure supplied and maintained by the driver:

`ifp->if_softc`
: Pointer to the device-dependent data. The first entry in the device-dependent data structure must be an `arpcom` structure.

`ifp->if_name`
: The name of the device. The network stack uses this string and the device number for device name lookups. The device name should be obtained from the `name` entry in the configuration structure.

`ifp->if_unit`
: The device number. The network stack uses this number and the device name for device name lookups. For example, if `ifp->if_name` is '`scc`' and `ifp->if_unit` is '`1`', the full device name would be '`scc1`'. The unit number should be obtained from the 'name' entry in the configuration structure.

`ifp->if_mtu`
: The maximum transmission unit for the device. For Ethernet devices this value should almost always be 1500.

`ifp->if_flags`
: The device flags. Ethernet devices should set the flags to `IFF_BROADCAST|IFF_SIMPLEX`, indicating that the device can broadcast packets to multiple destinations and does not receive and transmit at the same time.

`ifp->if_snd.ifq_maxlen`
: The maximum length of the queue of packets waiting to be sent to the driver. This is normally set to `ifqmaxlen`.

`ifp->if_init`
: The address of the driver initialization function.

`ifp->if_start`
: The address of the driver start function.

`ifp->if_ioctl`
: The address of the driver ioctl function.

`ifp->if_output` The address of the output function. Ethernet devices should set this to `ether_output`.

RTEMS provides a function to parse the driver name in the configuration structure into a device name and unit number.

```
int rtems_bsdnet_parse_driver_name (
  const struct rtems_bsdnet_ifconfig *config,
  char **namep
);
```

The function takes two arguments; a pointer to the configuration structure and a pointer to a pointer to a character. The function parses the configuration name entry, allocates memory for the driver name, places the driver name in this memory, sets the second argument to point to the name and returns the unit number. On error, a message is printed and -1 is returned.

Once the attach function has set up the above entries it must link the driver data structure onto the list of devices by calling `if_attach`. Ethernet devices should then call `ether_ifattach`. Both functions take a pointer to the device's `ifnet` structure as their only argument.

The attach function should return a non-zero value to indicate that the driver has been successfully configured and attached.

2.6 Write the Driver Start Function.

This function is called each time the network stack wants to start the transmitter. This occures whenever the network stack adds a packet to a device's send queue and the `IFF_OACTIVE` bit in the device's `if_flags` is not set.

For many devices this function need only set the `IFF_OACTIVE` bit in the `if_flags` and send an event to the transmit task indicating that a packet is in the driver transmit queue.

2.7 Write the Driver Initialization Function.

This function should initialize the device, attach to interrupt handler, and start the driver transmit and receive tasks. The function

```
rtems_id
rtems_bsdnet_newproc (char *name,
      int stacksize,
      void(*entry)(void *),
      void *arg);
```

should be used to start the driver tasks.

Note that the network stack may call the driver initialization function more than once. Make sure multiple versions of the receive and transmit tasks are not accidentally started.

2.8 Write the Driver Transmit Task

This task is reponsible for removing packets from the driver send queue and sending them to the device. The task should block waiting for an event from the driver start function indicating that packets are waiting to be transmitted. When the transmit task has drained the driver send queue the task should clear the `IFF_OACTIVE` bit in `if_flags` and block until another outgoing packet is queued.

2.9 Write the Driver Receive Task

This task should block until a packet arrives from the device. If the device is an Ethernet interface the function `ether_input` should be called to forward the packet to the network stack. The arguments to `ether_input` are a pointer to the interface data structure, a pointer to the ethernet header and a pointer to an mbuf containing the packet itself.

2.10 Write the Driver Interrupt Handler

A typical interrupt handler will do nothing more than the hardware manipulation required to acknowledge the interrupt and send an RTEMS event to wake up the driver receive or transmit task waiting for the event. Network interface interrupt handlers must not make any calls to other network routines.

2.11 Write the Driver IOCTL Function

This function handles ioctl requests directed at the device. The ioctl commands which must be handled are:

`SIOCGIFADDR`

`SIOCSIFADDR`
: If the device is an Ethernet interface these commands should be passed on to `ether_ioctl`.

`SIOCSIFFLAGS`
: This command should be used to start or stop the device, depending on the state of the interface `IFF_UP` and `IFF_RUNNING` bits in `if_flags`:

  `IFF_RUNNING`
  : Stop the device.

  `IFF_UP`
  : Start the device.

  `IFF_UP|IFF_RUNNING`
  : Stop then start the device.

  `0`
  : Do nothing.

2.12 Write the Driver Statistic-Printing Function

This function should print the values of any statistic/diagnostic counters the network driver may use. The driver ioctl function should call the statistic-printing function when the ioctl command is `SIO_RTEMS_SHOW_STATS`.

3 Using Networking in an RTEMS Application

3.1 Makefile changes

3.1.1 Including the required managers

The FreeBSD networking code requires several RTEMS managers in the application:

```
MANAGERS = io event semaphore
```

3.1.2 Increasing the size of the heap

The networking tasks allocate a lot of memory. For most applications the heap should be at least 256 kbytes. The amount of memory set aside for the heap can be adjusted by setting the `CFLAGS_LD` definition as shown below:

```
CFLAGS_LD += -Wl,--defsym -Wl,HeapSize=0x80000
```

This sets aside 512 kbytes of memory for the heap.

3.2 System Configuration

The networking tasks allocate some RTEMS objects. These must be accounted for in the application configuration table. The following lists the requirements.

TASKS
One network task plus a receive and transmit task for each device.

SEMAPHORES
One network semaphore plus one syslog mutex semaphore if the application uses openlog/syslog.

EVENTS
The network stack uses `RTEMS_EVENT_24` and `RTEMS_EVENT_25`. This has no effect on the application configuration, but application tasks which call the network functions should not use these events for other purposes.

3.3 Initialization

3.3.1 Additional include files

The source file which declares the network configuration structures and calls the network initialization function must include

```
#include <rtems/rtems_bsdnet.h>
```

3.3.2 Network Configuration

The network configuration is specified by declaring and initializing the `rtems_bsdnet_config` structure.

```
struct rtems_bsdnet_config {
  /*
   * This entry points to the head of the ifconfig chain.
   */
  struct rtems_bsdnet_ifconfig *ifconfig;

  /*
   * This entry should be rtems_bsdnet_do_bootp if BOOTP
   * is being used to configure the network, and NULL
   * if BOOTP is not being used.
   */
  void                    (*bootp)(void);

  /*
   * The remaining items can be initialized to 0, in
   * which case the default value will be used.
   */
  rtems_task_priority  network_task_priority;  /* 100        */
  unsigned long        mbuf_bytecount;         /* 64 kbytes  */
  unsigned long        mbuf_cluster_bytecount; /* 128 kbytes */
  char                *hostname;               /* BOOTP      */
  char                *domainname;             /* BOOTP      */
  char                *gateway;                /* BOOTP      */
  char                *log_host;               /* BOOTP      */
  char                *name_server[3];         /* BOOTP      */
  char                *ntp_server[3];          /* BOOTP      */
};
```

The structure entries are described in the following table. If your application uses BOOTP/DHCP to obtain network configuration information and if you are happy with the default values described below, you need to provide only the first two entries in this structure.

`struct rtems_bsdnet_ifconfig *ifconfig`
: A pointer to the first configuration structure of the first network device. This structure is described in the following section. You must provide a value for this entry since there is no default value for it.

`void (*bootp)(void)`
: This entry should be set to `rtems_bsdnet_do_bootp` if your application will use BOOTP/DHCP to obtain network configuration in-

formation. It should be set to `NULL` if your application does not use BOOTP/DHCP.

`int network_task_priority`
: The priority at which the network task and network device receive and transmit tasks will run. If a value of 0 is specified the tasks will run at priority 100.

`unsigned long mbuf_bytecount`
: The number of bytes to allocate from the heap for use as mbufs. If a value of 0 is specified, 64 kbytes will be allocated.

`unsigned long mbuf_cluster_bytecount`
: The number of bytes to allocate from the heap for use as mbuf clusters. If a value of 0 is specified, 128 kbytes will be allocated.

`char *hostname`
: The host name of the system. If this, or any of the following, entries are `NULL` the value may be obtained from a BOOTP/DHCP server.

`char *domainname`
: The name of the Internet domain to which the system belongs.

`char *gateway`
: The Internet host number of the network gateway machine, specified in 'dotted decimal' (`129.128.4.1`) form.

`char *log_host`
: The Internet host number of the machine to which `syslog` messages will be sent.

`char *name_server[3]`
: The Internet host numbers of up to three machines to be used as Internet Domain Name Servers.

`char *ntp_server[3]`
: The Internet host numbers of up to three machines to be used as Network Time Protocol (NTP) Servers.

In addition, the following fields in the `rtems_bsdnet_ifconfig` are of interest.

int port
: The I/O port number (ex: 0x240) on which the external Ethernet can be accessed.

int irno
: The interrupt number of the external Ethernet controller.

int bpar
: The address of the shared memory on the external Ethernet controller.

3.3.3 Network device configuration

Network devices are specified and configured by declaring and initializing a `struct rtems_bsdnet_ifconfig` structure for each network device.

The structure entries are described in the following table. An application which uses a single network interface, gets network configuration information from a BOOTP/DHCP server, and uses the default values for all driver parameters needs to initialize only the first two entries in the structure.

`char *name` The full name of the network device. This name consists of the driver name and the unit number (e.g. `"scc1"`). The `bsp.h` include file usually defines RTEMS_BSP_NETWORK_DRIVER_NAME as the name of the primary (or only) network driver.

`int (*attach)(struct rtems_bsdnet_ifconfig *conf)`
The address of the driver `attach` function. The network initialization function calls this function to configure the driver and attach it to the network stack. The `bsp.h` include file usually defines RTEMS_BSP_NETWORK_DRIVER_ATTACH as the name of the attach function of the primary (or only) network driver.

`struct rtems_bsdnet_ifconfig *next`
A pointer to the network device configuration structure for the next network interface, or `NULL` if this is the configuration structure of the last network interface.

`char *ip_address` The Internet address of the device, specified in 'dotted decimal' (`129.128.4.2`) form, or `NULL` if the device configuration information is being obtained from a BOOTP/DHCP server.

`char *ip_netmask` The Internet inetwork mask of the device, specified in 'dotted decimal' (`255.255.255.0`) form, or `NULL` if the device configuration information is being obtained from a BOOTP/DHCP server.

`void *hardware_address`
The hardware address of the device, or `NULL` if the driver is to obtain the hardware address in some other way (usually by reading it from the device or from the bootstrap ROM).

`int ignore_broadcast`
Zero if the device is to accept broadcast packets, non-zero if the device is to ignore broadcast packets.

`int mtu` The maximum transmission unit of the device, or zero if the driver is to choose a default value (typically 1500 for Ethernet devices).

`int rbuf_count` The number of receive buffers to use, or zero if the driver is to choose a default value

`int xbuf_count` The number of transmit buffers to use, or zero if the driver is to choose a default value Keep in mind that some network devices may use 4 or more transmit descriptors for a single transmit buffer.

A complete network configuration specification can be as simple as the one shown in the following example. This configuration uses a single network interface, gets network configuration information from a BOOTP/DHCP server, and uses the default values for all driver parameters.

```
static struct rtems_bsdnet_ifconfig netdriver_config = {
RTEMS_BSP_NETWORK_DRIVER_NAME,
RTEMS_BSP_NETWORK_DRIVER_ATTACH
};
```

```
struct rtems_bsdnet_config rtems_bsdnet_config = {
&netdriver_config,
rtems_bsdnet_do_bootp,
};
```

3.3.4 Network initialization

The networking tasks must be started before any network I/O operations can be performed. This is done by calling:

```
rtems_bsdnet_initialize_network ();
```

This function is declared in `rtems/rtems_bsdnet.h`. t returns 0 on success and -1 on failure with an error code in `errno`. It is not possible to undo the effects of a partial initialization, though, so the function can be called only once irregardless of the return code. Consequently, if the condition for the failure can be corrected, the system must be reset to permit another network initialization attempt.

3.4 Application Programming Interface

The RTEMS network package provides almost a complete set of BSD network services. The network functions work like their BSD counterparts with the following exceptions:

- A given socket can be read or written by only one task at a time.
- The `select` function only works for file descriptors associated with sockets.
- You must call `openlog` before calling any of the `syslog` functions.
- **Some of the network functions are not thread-safe.** For example the following functions return a pointer to a static buffer which remains valid only until the next call:

  `gethostbyaddr`

  `gethostbyname`

  `inet_ntoa` (`inet_ntop` is thread-safe, though).
- The RTEMS network package gathers statistics.
- Addition of a mechanism to "tap onto" an interface and monitor every packet received and transmitted.
- Addition of `SO_SNDWAKEUP` and `SO_RCVWAKEUP` socket options.

Some of the new features are discussed in more detail in the following sections.

3.4.1 Network Statistics

There are a number of functions to print statistics gathered by the network stack. These function are declared in `rtems/rtems_bsdnet.h`.

`rtems_bsdnet_show_if_stats`
: Display statistics gathered by network interfaces.

`rtems_bsdnet_show_ip_stats`
: Display IP packet statistics.

`rtems_bsdnet_show_icmp_stats`
: Display ICMP packet statistics.

`rtems_bsdnet_show_tcp_stats`
: Display TCP packet statistics.

`rtems_bsdnet_show_udp_stats`
: Display UDP packet statistics.

`rtems_bsdnet_show_mbuf_stats`
: Display mbuf statistics.

`rtems_bsdnet_show_inet_routes`
: Display the routing table.

3.4.2 Tapping Into an Interface

RTEMS add two new ioctls to the BSD networking code: SIOCSIFTAP and SIOCGIFTAP. These may be used to set and get a *tap function*. The tap function will be called for every Ethernet packet received by the interface.

These are called like other interface ioctls, such as SIOCSIFADDR. When setting the tap function with SIOCSIFTAP, set the ifr_tap field of the ifreq struct to the tap function. When retrieving the tap function with SIOCGIFTAP, the current tap function will be returned in the ifr_tap field. To stop tapping packets, call SIOCSIFTAP with a ifr_tap field of 0.

The tap function is called like this:

```
int tap (struct ifnet *, struct ether_header *, struct mbuf *)
```

The tap function should return 1 if the packet was fully handled, in which case the caller will simply discard the mbuf. The tap function should return 0 if the packet should be passed up to the higher networking layers.

The tap function is called with the network semaphore locked. It must not make any calls on the application levels of the networking level itself. It is safe to call other non-networking RTEMS functions.

3.4.3 Socket Options

RTEMS adds two new `SOL_SOCKET` level options for `setsockopt` and `getsockopt`: `SO_SNDWAKEUP` and `SO_RCVWAKEUP`. For both, the option value should point to a sockwakeup structure. The sockwakeup structure has the following fields:

```
void    (*sw_pfn) (struct socket *, caddr_t);
caddr_t sw_arg;
```

These options are used to set a callback function to be called when, for example, there is data available from the socket (`SO_RCVWAKEUP`) and when there is space available to accept data written to the socket (`SO_SNDWAKEUP`).

If `setsockopt` is called with the `SO_RCVWAKEUP` option, and the `sw_pfn` field is not zero, then when there is data available to be read from the socket, the function pointed to by the `sw_pfn` field will be called. A pointer to the socket structure will be passed as the first argument to the function. The `sw_arg` field set by the `SO_RCVWAKEUP` call will be passed as the second argument to the function.

If `setsockopt` is called with the `SO_SNDWAKEUP` function, and the `sw_pfn` field is not zero, then when there is space available to accept data written to the socket, the function pointed to by the `sw_pfn` field will be called. The arguments passed to the function will be as with `SO_SNDWAKEUP`.

When the function is called, the network semaphore will be locked and the callback function runs in the context of the networking task. The function must be careful not to call any networking functions. It is OK to call an RTEMS function; for example, it is OK to send an RTEMS event.

The purpose of these callback functions is to permit a more efficient alternative to the select call when dealing with a large number of sockets.

The callbacks are called by the same criteria that the select function uses for indicating "ready" sockets. In Stevens *Unix Network Programming* on page 153-154 in the section "Under what Conditions Is a Descriptor Ready?" you will find the definitive list of conditions for readable and writable that also determine when the functions are called.

When the number of received bytes equals or exceeds the socket receive buffer "low water mark" (default 1 byte) you get a readable callback. If there are 100 bytes in the receive buffer and you only read 1, you will not immediately get another callback. However, you will get another callback after you read the remaining 99 bytes and at least 1 more byte arrives. Using a non-blocking socket you should probably read until it produces error EWOULDBLOCK and then allow the readable callback to tell you when more data has arrived. (Condition 1.a.)

For sending, when the socket is connected and the free space becomes at or above the "low water mark" for the send buffer (default 4096 bytes) you will receive a writable callback. You don't get continuous callbacks if you don't write anything. Using a non-blocking write socket, you can then call write until it returns a value less than the amount of data requested to be sent or it produces error EWOULDBLOCK (indicating buffer full and no longer writable). When this happens you can try the write again, but it is often better to go do other things and let the writable callback tell you when space is available to send again. You only get a writable callback when the free space transitions to above the "low water mark" and not every time you write to a non-full send buffer. (Condition 2.a.)

The remaining conditions enumerated by Stevens handle the fact that sockets become readable and/or writable when connects, disconnects and errors occur, not just when data is received or sent. For example, when a server "listening" socket becomes readable it indicates that a client has connected and accept can be called without blocking, not that network data was received (Condition 1.c).

3.4.4 Adding an IP Alias

The following code snippet adds an IP alias:

```
void addAlias(const char *pName, const char *pAddr, const char *pMask)
{
  struct ifaliasreq      aliasreq;
  struct sockaddr_in    *in;

  /* initialize alias request */
  memset(&aliasreq, 0, sizeof(aliasreq));
  sprintf(aliasreq.ifra_name, pName);

  /* initialize alias address */
  in = (struct sockaddr_in *)&aliasreq.ifra_addr;
  in->sin_family = AF_INET;
  in->sin_len    = sizeof(aliasreq.ifra_addr);
  in->sin_addr.s_addr = inet_addr(pAddr);

  /* initialize alias mask */
  in = (struct sockaddr_in *)&aliasreq.ifra_mask;
  in->sin_family = AF_INET;
  in->sin_len    = sizeof(aliasreq.ifra_mask);
  in->sin_addr.s_addr = inet_addr(pMask);

  /* call to setup the alias */
  rtems_bsdnet_ifconfig(pName, SIOCAIFADDR, &aliasreq);
}
```

Thanks to Mike Seirs (`mailto:mikes@poliac.com`) for this example code.

3.4.5 Adding a Default Route

The function provided in this section is functionally equivalent to the command `route add default gw yyy.yyy.yyy.yyy`:

```
void mon_ifconfig(int argc, char *argv[],  unsigned32 command_arg,
                  boolean verbose)
{
    struct sockaddr_in  ipaddr;
    struct sockaddr_in  dstaddr;
    struct sockaddr_in  netmask;
    struct sockaddr_in  broadcast;
    char               *iface;
    int                 f_ip        = 0;
    int                 f_ptp       = 0;
    int                 f_netmask   = 0;
```

```
int                 f_up        = 0;
int                 f_down      = 0;
int                 f_bcast     = 0;
int                 cur_idx;
int                 rc;
int                 flags;

bzero((void*) &ipaddr, sizeof(ipaddr));
bzero((void*) &dstaddr, sizeof(dstaddr));
bzero((void*) &netmask, sizeof(netmask));
bzero((void*) &broadcast, sizeof(broadcast));

ipaddr.sin_len = sizeof(ipaddr);
ipaddr.sin_family = AF_INET;

dstaddr.sin_len = sizeof(dstaddr);
dstaddr.sin_family = AF_INET;

netmask.sin_len = sizeof(netmask);
netmask.sin_family = AF_INET;

broadcast.sin_len = sizeof(broadcast);
broadcast.sin_family = AF_INET;

cur_idx = 0;
if (argc <= 1) {
    /* display all interfaces */
    iface = NULL;
    cur_idx += 1;
} else {
    iface = argv[1];
    if (isdigit(*argv[2])) {
        if (inet_pton(AF_INET, argv[2], &ipaddr.sin_addr) < 0) {
            printf("bad ip address: %s\n", argv[2]);
            return;
        }
        f_ip = 1;
        cur_idx += 3;
    } else {
        cur_idx += 2;
    }
}

if ((f_down !=0) && (f_ip != 0)) {
    f_up = 1;
}
```

```
while(argc > cur_idx) {
    if (strcmp(argv[cur_idx], "up") == 0) {
        f_up = 1;
        if (f_down != 0) {
            printf("Can't make interface up and down\n");
        }
    } else if(strcmp(argv[cur_idx], "down") == 0) {
        f_down = 1;
        if (f_up != 0) {
            printf("Can't make interface up and down\n");
        }
    } else if(strcmp(argv[cur_idx], "netmask") == 0) {
        if ((cur_idx + 1) >= argc) {
            printf("No netmask address\n");
            return;
        }
        if (inet_pton(AF_INET, argv[cur_idx+1], &netmask.sin_addr) < 0) {
            printf("bad netmask: %s\n", argv[cur_idx]);
            return;
        }
        f_netmask = 1;
        cur_idx += 1;
    } else if(strcmp(argv[cur_idx], "broadcast") == 0) {
        if ((cur_idx + 1) >= argc) {
            printf("No broadcast address\n");
            return;
        }
        if (inet_pton(AF_INET, argv[cur_idx+1], &broadcast.sin_addr) < 0) {
            printf("bad broadcast: %s\n", argv[cur_idx]);
            return;
        }
        f_bcast = 1;
        cur_idx += 1;
    } else if(strcmp(argv[cur_idx], "pointopoint") == 0) {
        if ((cur_idx + 1) >= argc) {
            printf("No pointopoint address\n");
            return;
        }
        if (inet_pton(AF_INET, argv[cur_idx+1], &dstaddr.sin_addr) < 0) {
            printf("bad pointopoint: %s\n", argv[cur_idx]);
            return;
        }

        f_ptp = 1;
        cur_idx += 1;
    } else {
        printf("Bad parameter: %s\n", argv[cur_idx]);
```

```
            return;
        }

        cur_idx += 1;
    }

    printf("ifconfig ");
    if (iface != NULL) {
        printf("%s ", iface);
        if (f_ip != 0) {
            char str[256];
            inet_ntop(AF_INET, &ipaddr.sin_addr, str, 256);
            printf("%s ", str);
        }

        if (f_netmask != 0) {
            char str[256];
            inet_ntop(AF_INET, &netmask.sin_addr, str, 256);
            printf("netmask %s ", str);
        }

        if (f_bcast != 0) {
            char str[256];
            inet_ntop(AF_INET, &broadcast.sin_addr, str, 256);
            printf("broadcast %s ", str);
        }

        if (f_ptp != 0) {
            char str[256];
            inet_ntop(AF_INET, &dstaddr.sin_addr, str, 256);
            printf("pointopoint %s ", str);
        }

        if (f_up != 0) {
            printf("up\n");
        } else if (f_down != 0) {
            printf("down\n");
        } else {
            printf("\n");
        }
    }

    if ((iface == NULL) || ((f_ip == 0) && (f_down == 0) && (f_up == 0))) {
        rtems_bsdnet_show_if_stats();
        return;
    }
```

```
flags = 0;
if (f_netmask) {
    rc = rtems_bsdnet_ifconfig(iface, SIOCSIFNETMASK, &netmask);
    if (rc < 0) {
        printf("Could not set netmask: %s\n", strerror(errno));
        return;
    }
}

if (f_bcast) {
    rc = rtems_bsdnet_ifconfig(iface, SIOCSIFBRDADDR, &broadcast);
    if (rc < 0) {
        printf("Could not set broadcast: %s\n", strerror(errno));
        return;
    }
}

if (f_ptp) {
    rc = rtems_bsdnet_ifconfig(iface, SIOCSIFDSTADDR, &dstaddr);
    if (rc < 0) {
        printf("Could not set destination address: %s\n", strerror(errno));
        return;
    }
    flags |= IFF_POINTOPOINT;
}

/* This must come _after_ setting the netmask, broadcast addresses */
if (f_ip) {
    rc = rtems_bsdnet_ifconfig(iface, SIOCSIFADDR, &ipaddr);
    if (rc < 0) {
        printf("Could not set IP address: %s\n", strerror(errno));
        return;
    }
}

if (f_up != 0) {
    flags |= IFF_UP;
}

if (f_down != 0) {
    printf("Warning: taking interfaces down is not supported\n");
}

rc = rtems_bsdnet_ifconfig(iface, SIOCSIFFLAGS, &flags);
if (rc < 0) {
    printf("Could not set interface flags: %s\n", strerror(errno));
    return;
```

```
    }
}


void mon_route(int argc, char *argv[],  unsigned32 command_arg,
               boolean verbose)
{
    int                cmd;
    struct sockaddr_in dst;
    struct sockaddr_in gw;
    struct sockaddr_in netmask;
    int                f_host;
    int                f_gw       = 0;
    int                cur_idx;
    int                flags;
    int                rc;

    memset(&dst, 0, sizeof(dst));
    memset(&gw, 0, sizeof(gw));
    memset(&netmask, 0, sizeof(netmask));

    dst.sin_len = sizeof(dst);
    dst.sin_family = AF_INET;
    dst.sin_addr.s_addr = inet_addr("0.0.0.0");

    gw.sin_len = sizeof(gw);
    gw.sin_family = AF_INET;
    gw.sin_addr.s_addr = inet_addr("0.0.0.0");

    netmask.sin_len = sizeof(netmask);
    netmask.sin_family = AF_INET;
    netmask.sin_addr.s_addr = inet_addr("255.255.255.0");

    if (argc < 2) {
        rtems_bsdnet_show_inet_routes();
        return;
    }

    if (strcmp(argv[1], "add") == 0) {
        cmd = RTM_ADD;
    } else if (strcmp(argv[1], "del") == 0) {
        cmd = RTM_DELETE;
    } else {
        printf("invalid command: %s\n", argv[1]);
        printf("\tit should be 'add' or 'del'\n");
        return;
```

```
    }

    if (argc < 3) {
        printf("not enough arguments\n");
        return;
    }

    if (strcmp(argv[2], "-host") == 0) {
        f_host = 1;
    } else if (strcmp(argv[2], "-net") == 0) {
        f_host = 0;
    } else {
        printf("Invalid type: %s\n", argv[1]);
        printf("\tit should be '-host' or '-net'\n");
        return;
    }

    if (argc < 4) {
        printf("not enough arguments\n");
        return;
    }

    inet_pton(AF_INET, argv[3], &dst.sin_addr);

    cur_idx = 4;
    while(cur_idx < argc) {
        if (strcmp(argv[cur_idx], "gw") == 0) {
            if ((cur_idx +1) >= argc) {
                printf("no gateway address\n");
                return;
            }
            f_gw = 1;
            inet_pton(AF_INET, argv[cur_idx + 1], &gw.sin_addr);
            cur_idx += 1;
        } else if(strcmp(argv[cur_idx], "netmask") == 0) {
            if ((cur_idx +1) >= argc) {
                printf("no netmask address\n");
                return;
            }
            f_gw = 1;
            inet_pton(AF_INET, argv[cur_idx + 1], &netmask.sin_addr);
            cur_idx += 1;
        } else {
            printf("Unknown argument\n");
            return;
        }
        cur_idx += 1;
```

```
    }

    flags = RTF_STATIC;
    if (f_gw != 0) {
        flags |= RTF_GATEWAY;
    }
    if (f_host != 0) {
        flags |= RTF_HOST;
    }

    rc = rtems_bsdnet_rtrequest(cmd, &dst, &gw, &netmask, flags, NULL);
    if (rc < 0) {
        printf("Error adding route\n");
    }
}
```

Thanks to Jay Monkman (`mailto:jtm@smoothmsmoothie.com`) for this example code.

3.4.6 Time Synchronization Using NTP

```
int rtems_bsdnet_synchronize_ntp (int interval, rtems_task_priority priority);
```

If the interval argument is 0 the routine synchronizes the RTEMS time-of-day clock with the first NTP server in the rtems_bsdnet_ntpserve array and returns. The priority argument is ignored.

If the interval argument is greater than 0, the routine also starts an RTEMS task at the specified priority and polls the NTP server every 'interval' seconds. NOTE: This mode of operation has not yet been implemented.

On successful synchronization of the RTEMS time-of-day clock the routine returns 0. If an error occurs a message is printed and the routine returns -1 with an error code in errno. There is no timeout – if there is no response from an NTP server the routine will wait forever.

4 Testing the Driver

4.1 Preliminary Setup

The network used to test the driver should include at least:

- The hardware on which the driver is to run. It makes testing much easier if you can run a debugger to control the operation of the target machine.
- An Ethernet network analyzer or a workstation with an 'Ethernet snoop' program such as `ethersnoop` or `tcpdump`.
- A workstation.

During early debug, you should consider putting the target, workstation, and snooper on a small network by themselves. This offers a few advantages:

- There is less traffic to look at on the snooper and for the target to process while bringing the driver up.
- Any serious errors will impact only your small network not a building or campus network. You want to avoid causing any unnecessary problems.
- Test traffic is easier to repeatably generate.
- Performance measurements are not impacted by other systems on the network.

4.2 Debug Output

There are a number of sources of debug output that can be enabled to aid in tracing the behavior of the network stack. The following is a list of them:

- mbuf activity

  There are commented out calls to `printf` in the file `sys/mbuf.h` in the network stack code. Uncommenting these lines results in output when mbuf's are allocated and freed. This is very useful for finding memory leaks.
- TX and RX queuing

  There are commented out calls to `printf` in the file `net/if.h` in the network stack code. Uncommenting these lines results in output when packets are placed on or removed from one of the transmit or receive packet queues. These queues can be viewed as the boundary line between a device driver and the network stack. If the network stack is enqueuing packets to be transmitted that the device driver is not dequeuing, then that is indicative of a problem in the transmit side of the device driver. Conversely, if the device driver is enqueueing packets as it receives them (via a call to `ether_input`) and they are not being dequeued by the network stack, then there is a problem. This situation would likely indicate that the network server task is not running.

- TCP state transitions

  In the unlikely event that one would actually want to see TCP state transitions, the `TCPDEBUG` macro can be defined in the file `opt_tcpdebug.h`. This results in the routine `tcp_trace()` being called by the network stack and the state transitions logged into the `tcp_debug` data structure. If the variable `tcpconsdebug` in the file `netinet/tcp_debug.c` is set to 1, then the state transitions will also be printed to the console.

4.3 Driver basic operation

The network demonstration program `netdemo` may be used for these tests.

- Edit `networkconfig.h` to reflect the values for your network.
- Start with `RTEMS_USE_BOOTP` not defined.
- Edit `networkconfig.h` to configure the driver with an explicit Ethernet and Internet address and with reception of broadcast packets disabled:

  Verify that the program continues to run once the driver has been attached.
- Issue a '`u`' command to send UDP packets to the 'discard' port. Verify that the packets appear on the network.
- Issue a '`s`' command to print the network and driver statistics.
- On a workstation, add a static route to the target system.
- On that same workstation try to 'ping' the target system. Verify that the ICMP echo request and reply packets appear on the net.
- Remove the static route to the target system. Modify `networkconfig.h` to attach the driver with reception of broadcast packets enabled. Try to 'ping' the target system again. Verify that ARP request/reply and ICMP echo request/reply packets appear on the net.
- Issue a '`t`' command to send TCP packets to the 'discard' port. Verify that the packets appear on the network.
- Issue a '`s`' command to print the network and driver statistics.
- Verify that you can telnet to ports 24742 and 24743 on the target system from one or more workstations on your network.

4.4 BOOTP/DHCP operation

Set up a BOOTP/DHCP server on the network. Set define `RTEMS USE_BOOT` in `networkconfig.h`. Run the `netdemo` test program. Verify that the target system configures itself from the BOOTP/DHCP server and that all the above tests succeed.

4.5 Stress Tests

Once the driver passes the tests described in the previous section it should be subjected to conditions which exercise it more thoroughly and which test its error handling routines.

4.5.1 Giant packets

- Recompile the driver with `MAXIMUM_FRAME_SIZE` set to a smaller value, say 514.
- 'Ping' the driver from another workstation and verify that frames larger than 514 bytes are correctly rejected.
- Recompile the driver with `MAXIMUM_FRAME_SIZE` restored to 1518.

4.5.2 Resource Exhaustion

- Edit `networkconfig.h` so that the driver is configured with just two receive and transmit descriptors.
- Compile and run the `netdemo` program.
- Verify that the program operates properly and that you can still telnet to both the ports.
- Display the driver statistics (Console '`s`' command or telnet 'control-G' character) and verify that:
    1. The number of transmit interrupts is non-zero. This indicates that all transmit descriptors have been in use at some time.
    2. The number of missed packets is non-zero. This indicates that all receive descriptors have been in use at some time.

4.5.3 Cable Faults

- Run the `netdemo` program.
- Issue a '`u`' console command to make the target machine transmit a bunch of UDP packets.
- While the packets are being transmitted, disconnect and reconnect the network cable.
- Display the network statistics and verify that the driver has detected the loss of carrier.
- Verify that you can still telnet to both ports on the target machine.

4.5.4 Throughput

Run the `ttcp` network benchmark program. Transfer large amounts of data (100's of megabytes) to and from the target system.

The procedure for testing throughput from a host to an RTEMS target is as follows:

1. Download and start the ttcp program on the Target.
2. In response to the `ttcp` prompt, enter `-s -r`. The meaning of these flags is described in the `ttcp.1` manual page found in the `ttcp_orig` subdirectory.
3. On the host run `ttcp -s -t <<insert the hostname or IP address of the Target here>>`

The procedure for testing throughput from an RTEMS target to a Host is as follows:

1. On the host run `ttcp -s -r`.
2. Download and start the ttcp program on the Target.
3. In response to the `ttcp` prompt, enter `-s -t <<insert the hostname or IP address of the Target here>>`. You need to type the IP address of the host unless your Target is talking to your Domain Name Server.

To change the number of buffers, the buffer size, etc. you just add the extra flags to the `-t` machine as specified in the `ttcp.1` manual page found in the `ttcp_orig` subdirectory.

5 Network Servers

5.1 RTEMS FTP Daemon

The RTEMS FTPD is a complete file transfer protocol (FTP) daemon which can store, retrieve, and manipulate files on the local filesystem. In addition, the RTEMS FTPD provides "hooks" which are actions performed on received data. Hooks are useful in situations where a destination file is not necessarily appropriate or in cases when a formal device driver has not yet been implemented.

This server was implemented and documented by Jake Janovetz (janovetz@tempest.ece.uiuc.edu).

5.1.1 Configuration Parameters

The configuration structure for FTPD is as follows:

```
struct rtems_ftpd_configuration
{
   rtems_task_priority     priority;           /* FTPD task priority  */
   unsigned long           max_hook_filesize;  /* Maximum buffersize  */
                                               /*    for hooks        */
   int                     port;               /* Well-known port     */
   struct rtems_ftpd_hook  *hooks;             /* List of hooks       */
};
```

The FTPD task priority is specified with `priority`. Because hooks are not saved as files, the received data is placed in an allocated buffer. `max_hook_filesize` specifies the maximum size of this buffer. Finally, `hooks` is a pointer to the configured hooks structure.

5.1.2 Initializing FTPD (Starting the daemon)

Starting FTPD is done with a call to `rtems_initialize_ftpd()`. The configuration structure must be provided in the application source code. Example hooks structure and configuration structure folllow.

```
struct rtems_ftpd_hook ftp_hooks[] =
   {
      {"untar", Untar_FromMemory},
      {NULL, NULL}
   };

struct rtems_ftpd_configuration rtems_ftpd_configuration =
   {
      40,                     /* FTPD task priority */
      512*1024,               /* Maximum hook 'file' size */
```

```
    0,                         /* Use default port */
    ftp_hooks                  /* Local ftp hooks */
  };
```

Specifying 0 for the well-known port causes FTPD to use the UNIX standard FTPD port (21).

5.1.3 Using Hooks

In the example above, one hook was installed. The hook causes FTPD to call the function `Untar_FromMemory` when the user sends data to the file `untar`. The prototype for the `untar` hook (and hooks, in general) is:

```
int Untar_FromMemory(unsigned char *tar_buf, unsigned long size);
```

An example FTP transcript which exercises this hook is:

```
220 RTEMS FTP server (Version 1.0-JWJ) ready.
Name (dcomm0:janovetz): John Galt
230 User logged in.
Remote system type is RTEMS.
ftp> bin
200 Type set to I.
ftp> dir
200 PORT command successful.
150 ASCII data connection for LIST.
drwxrwx--x      0     0         268  dev
drwxrwx--x      0     0           0  TFTP
226 Transfer complete.
ftp> put html.tar untar
local: html.tar remote: untar
200 PORT command successful.
150 BINARY data connection.
210 File transferred successfully.
471040 bytes sent in 0.48 secs (9.6e+02 Kbytes/sec)
ftp> dir
200 PORT command successful.
150 ASCII data connection for LIST.
drwxrwx--x      0     0         268  dev
drwxrwx--x      0     0           0  TFTP
drwxrwx--x      0     0        3484  public_html
226 Transfer complete.
ftp> quit
221 Goodbye.
```

6 DEC 21140 Driver

6.1 DEC 21240 Driver Introduction

One aim of our project is to port RTEMS on a standard PowerPC platform. To achieve it, we have chosen a Motorola MCP750 board. This board includes an Ethernet controller based on a DEC21140 chip. Because RTEMS has a TCP/IP stack, we will have to develop the DEC21140 related ethernet driver for the PowerPC port of RTEMS. As this controller is able to support 100Mbps network and as there is a lot of PCI card using this DEC chip, we have decided to first implement this driver on an Intel PC386 target to provide a solution for using RTEMS on PC with the 100Mbps network and then to port this code on PowerPC in a second phase.

The aim of this document is to give some PCI board generalities and to explain the software architecture of the RTEMS driver. Finally, we will see what will be done for ChorusOs and Netboot environment .

6.2 Document Revision History

Current release:

- Current applicable release is 1.0.

Existing releases:

- 1.0 : Released the 10/02/98. First version of this document.
- 0.1 : First draft of this document

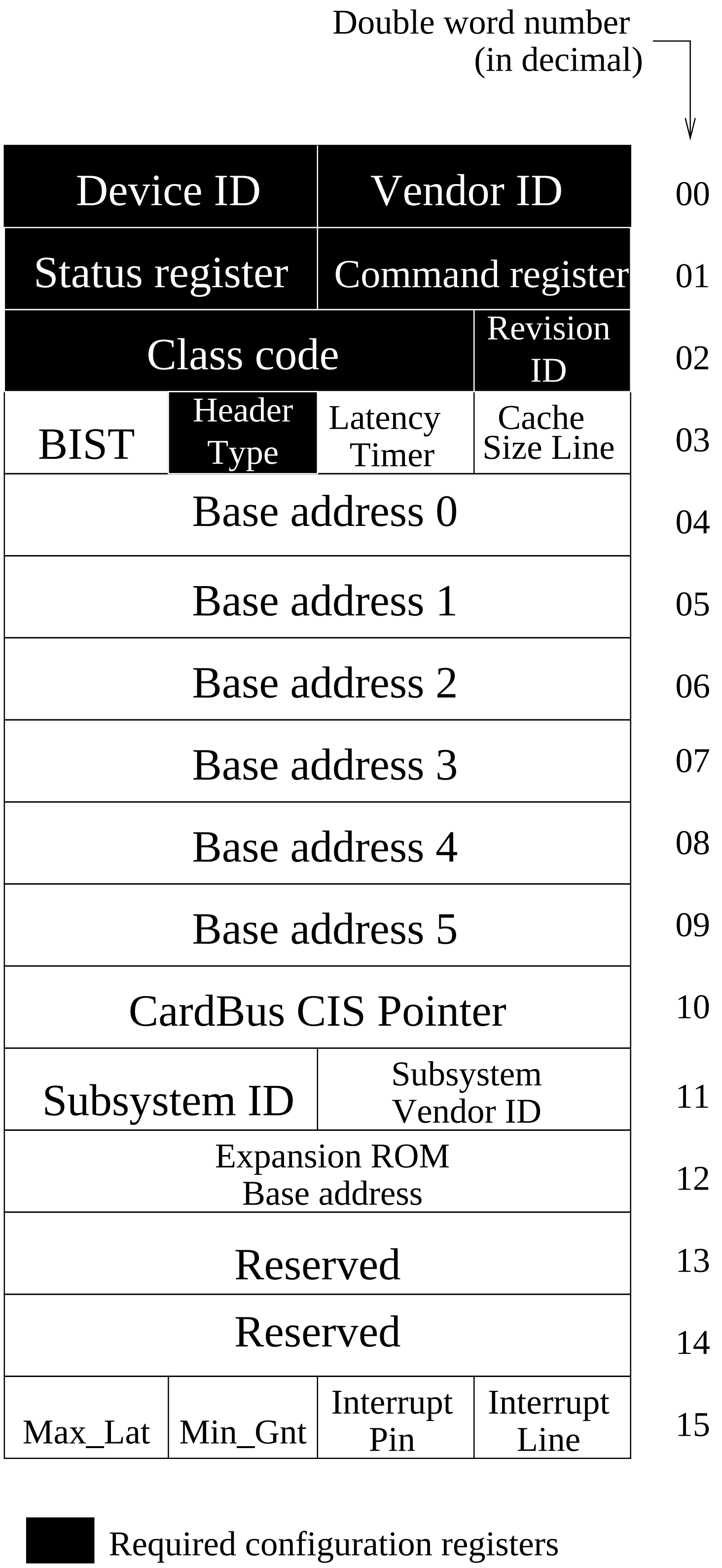
Planned releases:

- None planned today.

6.3 DEC21140 PCI Board Generalities

This chapter describes rapidely the PCI interface of this Ethernet controller. The board we have chosen for our PC386 implementation is a D-Link DFE-500TX. This is a dual-speed 10/100Mbps Ethernet PCI adapter with a DEC21140AF chip. Like other PCI devices, this board has a PCI device's header containing some required configuration registers, as shown in the PCI Register Figure. By reading or writing these registers, a driver can obtain information about the type of the board, the interrupt it uses, the mapping of the chip specific registers, ...

On Intel target, the chip specific registers can be accessed via 2 methods : I/O port access or PCI address mapped access. We have chosen to implement the PCI address access to obtain compatible source code to the port the driver on a PowerPC target.



On RTEMS, a PCI API exists. We have used it to configure the board. After initializing this PCI module via the `pcib_init()` function, we try to detect the DEC21140 based ethernet board. This board is characterized by its Vendor ID (0x1011) and its Device ID (0x0009). We give these arguments to the `pcib_find_by_deviceid` function which returns , if the device is present, a pointer to the configuration header space (see PCI Registers Fgure). Once this operation performed, the driver is able to extract the information it needs to configure the board internal registers, like the interrupt line, the base address,... The board internal registers will not be detailled here. You can find them in **DIGITAL Semiconductor 21140A PCI Fast Ethernet LAN Controller - Hardware Reference Manual**.

6.4 RTEMS Driver Software Architecture

In this chapter will see the initialization phase, how the controller uses the host memory and the 2 threads launched at the initialization time.

6.4.1 Initialization phase

The DEC21140 Ethernet driver keeps the same software architecture than the other RTEMS ethernet drivers. The only API the programmer can use is the `rtems_dec21140_driver_attach (struct rtems_bsdnet_ifconfig *config)` function which detects the board and initializes the associated data structure (with registers base address, entry points to low-level initialization function,...), if the board is found.

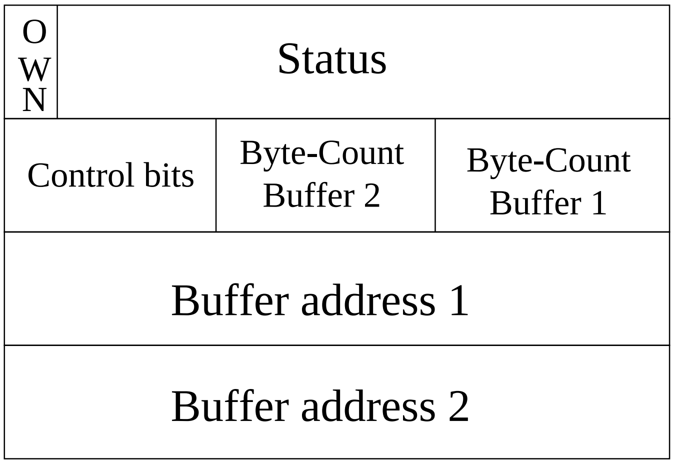
Once the attach function executed, the driver initializes the DEC chip. Then the driver connects an interrupt handler to the interrupt line driven by the Ethernet controller (the only interrupt which will be treated is the receive interrupt) and launches 2 threads : a receiver thread and a transmitter thread. Then the driver waits for incoming frame to give to the protocol stack or outcoming frame to send on the physical link.

6.4.2 Memory Buffer

This DEC chip uses the host memory to store the incoming Ethernet frames and the descriptor of these frames. We have chosen to use 7 receive buffers and 1 transmit buffer to optimize memory allocation due to cache and paging problem that will be explained in the section **Encountered Problems**.

To reference these buffers to the DEC chip we use a buffer descriptors ring. The descriptor structure is defined in the Buffer Descriptor Figure. Each descriptor can reference one or two memory buffers. We choose to use only one buffer of 1520 bytes per descriptor.

The difference between a receive and a transmit buffer descriptor is located in the status and control bits fields. We do not give details here, please refer to the [DEC21140 Hardware Manual].



6.4.3 Receiver Thread

This thread is event driven. Each time a DEC PCI board interrupt occurs, the handler checks if this is a receive interrupt and send an event "reception" to the receiver thread which looks into the entire buffer descriptors ring the ones that contain a valid incoming frame (bit OWN=0 means descriptor belongs to host processor). Each valid incoming ethernet frame is sent to the protocol stack and the buffer descriptor is given back to the DEC board (the host processor reset bit OWN, which means descriptor belongs to 21140).

6.4.4 Transmitter Thread

This thread is also event driven. Each time an Ethernet frame is put in the transmit queue, an event is sent to the transmit thread, which empty the queue by sending each outcoming frame. Because we use only one transmit buffer, we are sure that the frame is well-sent before sending the next.

6.5 Encountered Problems

On Intel PC386 target, we were faced with a problem of memory cache management. Because the DEC chip uses the host memory to store the incoming frame and because the DEC21140 configuration registers are mapped into the PCI address space, we must ensure that the data read (or written) by the host processor are the ones written (or read) by the DEC21140 device in the host memory and not old data stored in the cache memory. Therefore, we had to provide a way to manage the cache. This module is described in the document **RTEMS Cache Management For Intel**. On Intel, the memory region cache management is available only if the paging unit is enabled. We have used this paging mechanism, with 4Kb page. All the buffers allocated to store the incoming or outcoming frames, buffer descriptor and also the PCI address space of the DEC board are located in a memory space with cache disable.

Concerning the buffers and their descriptors, we have tried to optimize the memory space in term of allocated page. One buffer has 1520 bytes, one descriptor has 16 bytes. We have 7 receive buffers and 1 transmit buffer, and for each, 1 descriptor : (7+1)*(1520+16) = 12288 bytes = 12Kb = 3 entire pages. This allows not to lose too much memory or not to disable cache memory for a page which contains other data than buffer, which could decrease performance.

6.6 ChorusOs DEC Driver

Because ChorusOs is used in several Canon CRF projects, we must provide such a driver on this OS to ensure compatibility between the RTEMS and ChorusOs developments. On ChorusOs, a DEC driver source code already exists but only for a PowerPC target. We plan to port this code (which uses ChorusOs API) on Intel target. This will allow us to have homogeneous developments. Moreover, the port of the development performed with ChorusOs environment to RTEMS environment will be easier for the developers.

6.7 Netboot DEC driver

We use Netboot tool to load our development from a server to the target via an ethernet network. Currently, this tool does not support the DEC board. We plan to port the DEC driver for the Netboot tool.

But concerning the port of the DEC driver into Netboot, we are faced with a problem : in RTEMS environment, the DEC driver is interrupt or event driven, in Netboot environment, it must be used in polling mode. It means that we will have to re-write some mechanisms of this driver.

6.8 List of Ethernet cards using the DEC chip

Many Ethernet adapter cards use the Tulip chip. Here is a non exhaustive list of adapters which support this driver :

- Accton EtherDuo PCI.
- Accton EN1207 All three media types supported.
- Adaptec ANA6911/TX 21140-AC.
- Cogent EM110 21140-A with DP83840 N-Way MII transceiver.
- Cogent EM400 EM100 with 4 21140 100mbps-only ports + PCI Bridge.
- Danpex EN-9400P3.
- D-Link DFE500-Tx 21140-A with DP83840 transceiver.

- Kingston EtherX KNE100TX 21140AE.
- Netgear FX310 TX 10/100 21140AE.
- SMC EtherPower10/100 With DEC21140 and 68836 SYM transceiver.
- SMC EtherPower10/100 With DEC21140-AC and DP83840 MII transceiver. Note: The EtherPower II uses the EPIC chip, which requires a different driver.
- Surecom EP-320X DEC 21140.
- Thomas Conrad TC5048.
- Znyx ZX345 21140-A, usually with the DP83840 N-Way MII transciever. Some ZX345 cards made in 1996 have an ICS 1890 transciver instead.
- ZNYX ZX348 Two 21140-A chips using ICS 1890 transcievers and either a 21052 or 21152 bridge. Early versions used National 83840 transcievers, but later versions are depopulated ZX346 boards.
- ZNYX ZX351 21140 chip with a Broadcom 100BaseT4 transciever.

Our DEC driver has not been tested with all these cards, only with the D-Link DFE500-TX.

[DEC21140 Hardware Manual] DIGITAL, **DIGITAL Semiconductor 21140A PCI Fast Ethernet LAN Controller - Hardware Reference Manual**.

[99.TA.0021.M.ER]Emmanuel Raguet, **RTEMS Cache Management For Intel**.

Command and Variable Index

There are currently no Command and Variable Index entries.

Concept Index

There are currently no Concept Index entries.