



RTEMS Networking User Documentation

Release 4.11.3-rc1

©Copyright 2016, RTEMS Project (built 12nd February 2018)

CONTENTS

I	Preface	3
II	Network Task Structure and Data Flow	7
III	Networking Driver	9
1	Introduction	11
2	Learn about the network device	13
3	Understand the network scheduling conventions	15
4	Network Driver Makefile	17
5	Write the Driver Attach Function	19
6	Write the Driver Start Function.	21
7	Write the Driver Initialization Function.	23
8	Write the Driver Transmit Task	25
9	Write the Driver Receive Task	27
10	Write the Driver Interrupt Handler	29
11	Write the Driver IOCTL Function	31
12	Write the Driver Statistic-Printing Function	33
IV	Using Networking in an RTEMS Application	35
13	Makefile changes	37
13.1	Including the required managers	38
13.2	Increasing the size of the heap	39
14	System Configuration	41

15 Initialization	43
15.1 Additional include files	44
15.2 Network Configuration	45
15.3 Network device configuration	48
15.4 Network initialization	49
16 Application Programming Interface	51
16.1 Network Statistics	52
16.2 Tapping Into an Interface	53
16.3 Socket Options	54
16.4 Adding an IP Alias	55
16.5 Adding a Default Route	56
16.6 Time Synchronization Using NTP	60
V Testing the Driver	61
17 Preliminary Setup	63
18 Debug Output	65
19 Monitor Commands	67
20 Driver basic operation	69
21 BOOTP/DHCP operation	71
22 Stress Tests	73
22.1 Giant packets	74
22.2 Resource Exhaustion	75
22.3 Cable Faults	76
22.4 Throughput	77
VI Network Servers	79
23 RTEMS FTP Daemon	81
23.1 Configuration Parameters	82
23.2 Initializing FTPD (Starting the daemon)	83
23.3 Using Hooks	84
VII DEC 21140 Driver	85
24 DEC 21240 Driver Introduction	87
25 Document Revision History	89
26 DEC21140 PCI Board Generalities	91
27 RTEMS Driver Software Architecture	93
27.1 Initialization phase	94
27.2 Memory Buffer	95
27.3 Receiver Thread	96

27.4 Transmitter Thread	97
28 Encountered Problems	99
29 Netboot DEC driver	101
30 List of Ethernet cards using the DEC chip	103
 VIII Command and Variable Index	 105

COPYRIGHT (c) 1988 - 2015.

On-Line Applications Research Corporation (OAR).

The authors have used their best efforts in preparing this material. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. No warranty of any kind, expressed or implied, with regard to the software or the material contained in this document is provided. No liability arising out of the application or use of any product described in this document is assumed. The authors reserve the right to revise this material and to make changes from time to time in the content hereof without obligation to notify anyone of such revision or changes.

The RTEMS Project is hosted at <http://www.rtems.org/>. Any inquiries concerning RTEMS, its related support components, or its documentation should be directed to the Community Project hosted at <http://www.rtems.org/>.

RTEMS Online Resources

Home	https://www.rtems.org/
Developers	https://devel.rtems.org/
Documentation	https://docs.rtems.org/
Bug Reporting	https://devel.rtems.org/query
Mailing Lists	https://lists.rtems.org/
Git Repositories	https://git.rtems.org/

Part I

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.

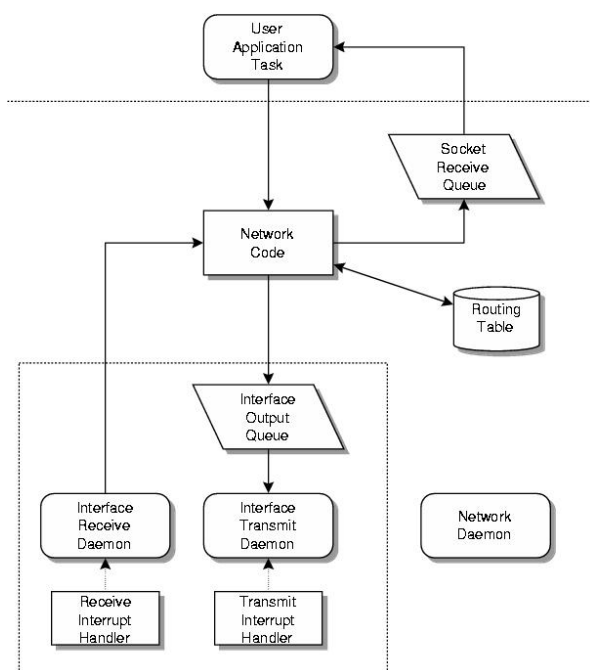
The following is a list of resources which should be useful in trying to understand Ethernet:

- *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 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.

Part II

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.

Part III

Networking Driver

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.

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.

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.

`rtems_event_receive`.

- `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

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:

```
1  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.

```
1  -D_COMPILING_BSD_KERNEL_ -DKERNEL -DINET -  
    ↪ DNFS \  
2  -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.

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.

```
1 int rtems_bsdnet_parse_driver_name (
2     const struct rtems_bsdnet_ifconfig *
3     ↪ config,
4     char **namep
5 );
```

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.

WRITE THE DRIVER START FUNCTION.

This function is called each time the network stack wants to start the transmitter. This occurs 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.

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

```
1 rtems_id  
2 rtems_bsdnet_newproc (char *name,  
3     int stacksize,  
4     void(*entry)(void *),  
5     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.

WRITE THE DRIVER TRANSMIT TASK

This task is responsible 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.

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.

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.

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.

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.

Part IV

Using Networking in an RTEMS Application

MAKEFILE CHANGES

13.1 Including the required managers

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

```
1 MANAGERS = io event semaphore
```

13.2 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:

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

This sets aside 512 kbytes of memory for the heap.

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.

INITIALIZATION

15.1 Additional include files

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

```
1 #include <rtems/rtms_bsdnet.h>
```

15.2 Network Configuration

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

```

1 struct rtems_bsdnet_config {
2     /*
3      * This entry points to the head of the
4      * ifconfig chain.
5      */
6     struct rtems_bsdnet_ifconfig *ifconfig;
7     /*
8      * This entry should be rtems_bsdnet_do_
9      * bootp if BOOTP
10     * is being used to configure the network,
11     * and NULL
12     * if BOOTP is not being used.
13     */
14     void
15     (*bootp)(void);
16     /*
17      * The remaining items can be
18      * initialized to 0, in
19      * which case the default value will be
20      * used.
21      */
22     rtems_task_priority          network_
23     task_priority; /* 100          */
24     unsigned long                mbuf_
25     bytecount; /* 64 kbytes */
26     unsigned long                mbuf_
27     cluster_bytecount; /* 128 kbytes */
28     char                         *hostname;
29     /* BOOTP */
30     char                         *domainname;
31     /* BOOTP */
32     char                         *gateway;
33     /* BOOTP */
34     char                         *log_host;
35     /* BOOTP */
36     char                         *name_
37     server[3]; /* BOOTP */
38     char                         *ntp_
39     server[3]; /* BOOTP */
40     unsigned long                sb_
41     efficiency; /* 2 */
42     /* UDP TX: 9216 bytes */
43     unsigned long                udp_tx_buf_
44     size;
45     /* UDP RX: 40 * (1024 + sizeof(struct
46     sockaddr_in)) */
47     unsigned long                udp_rx_buf_
48     size;
49     /* TCP TX: 16 * 1024 bytes */
50     unsigned long                tcp_tx_buf_
51     size;

```

```

32     /* TCP TX: 16 * 1024 bytes */
33     unsigned long                tcp_rx_buf_
34     size;
35     /* Default Network Tasks CPU Affinity */
36     #ifdef RTEMS_SMP
37     const cpu_set_t              *network_
38     task_cpuset;
39     size_t                       network_
40     task_cpuset_size;
41     #endif
42 };

```

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 by default uses the BOOTP/DHCP client protocol to obtain network configuration information. It should be set to `NULL` if your application does not use BOOTP/DHCP. You can also use `rtems_bsdnet_do_bootp_rootfs` to have a set of standard files created with the information return by the BOOTP/DHCP protocol. The IP address is added to `/etc/hosts` with the host name and domain returned. If no host name or domain is returned `me.mydomain` is used. The BOOTP/DHCP server's address is also added to `/etc/hosts`. The domain name server listed in the BOOTP/DHCP information are added to `/etc/resolv.conf`. A "search" record is also added if a domain is returned. The files are created if they do not exist. The default `rtems_bsdnet_do_bootp` and `rtems_bsdnet_do_bootp_rootfs` handlers will loop for-ever waiting for a BOOTP/DHCP server to respond. If an error is detected such as not valid interface or

valid hardware address the target will re-boot allowing any hardware reset to correct itself. You can provide your own custom handler which allows you to perform an initialization that meets your specific system requirements. For example you could try BOOTP/DHCP then enter a configuration tool if no server is found allowing the user to switch to a static configuration.

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.

unsigned long sb_efficiency

This is the first of five configuration parameters related to the amount of memory each socket may consume for buffers. The TCP/IP stack reserves buffers (e.g. mbufs) for each open socket. The TCP/IP stack has different limits for the transmit and receive buffers associated with each TCP and UDP socket. By tuning these parameters, the application developer can make trade-offs between memory consumption and performance. The default parameters favor performance over memory consumption. See <http://www.rtems.org/ml/rtems-users/2004/february/msg00200.html> for more details but note that after the RTEMS 4.8 release series, the sb_efficiency default was changed from 8 to 2. The user should also be aware of the SO_SNDBUF and SO_RCVBUF IO control operations. These can be used to specify the send and receive buffer sizes for a specific socket. There is no standard IO control to change the sb_efficiency factor. The sb_efficiency parameter is a buffering factor used in the implementation of the TCP/IP stack. The default is 2 which indicates double buffering. When allocating memory for each socket, this number is multiplied by the buffer sizes for that socket.

unsigned long udp_tx_buf_size

This configuration parameter specifies the maximum amount of buffer memory which may be used for UDP sockets to transmit with. The default size is 9216 bytes which corresponds to the maximum datagram size.

unsigned long udp_rx_buf_size

This configuration parameter specifies the maximum amount of buffer memory which may be used for UDP sockets to receive into. The default size is the following length in bytes:

```
1 40 * (1024 + sizeof(struct sockaddr_in))
```

unsigned long tcp_tx_buf_size

This configuration parameter specifies the maximum amount of buffer memory which may be used for TCP sockets to transmit with. The default size is sixteen kilobytes.

unsigned long tcp_rx_buf_size

This configuration parameter specifies the maximum amount of buffer memory which may be used for TCP sockets to receive into. The default size is sixteen kilobytes.

const cpu_set_t *network_task_cpuset

This configuration parameter specifies the CPU affinity of the network task. If set to 0 the network task can be scheduled on any CPU. Only available in SMP configurations.

size_t network_task_cpuset_size

This configuration parameter specifies the size of the network_task_cpuset used. Only available in SMP configurations.

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.

15.3 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
1 netdriver_config = {
2     RTEMS_BSP_NETWORK_DRIVER_NAME,
3     RTEMS_BSP_NETWORK_DRIVER_ATTACH
4 };
5 struct rtems_bsdnet_config rtems_bsdnet_
6 config = {
7     &netdriver_config,
8     rtems_bsdnet_do_bootp,
9 };
```

15.4 Network initialization

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

```
1 rtems_bsdnet_initialize_network ();
```

This function is declared in `rtems/rtems_bsdnet.h`. It 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.

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.

16.1 Network Statistics

There are a number of functions to print statistics gathered by the network stack. These functions 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.

16.2 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:

```
1 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.

16.3 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:

```
1 void      (*sw_pfn) (struct socket *, caddr_t);
2 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).

16.4 Adding an IP Alias

The following code snippet adds an IP alias:

```
1 void addAlias(const char *pName, const char_  
  ↪ *pAddr, const char *pMask)  
2 {  
3     struct ifaliasreq  aliasreq;  
4     struct sockaddr_in *in;  
5  
6     /* initialize alias request */  
7     memset(&aliasreq, 0, sizeof(aliasreq));  
8     sprintf(aliasreq.ifra_name, pName);  
9  
10    /* initialize alias address */  
11    in = (struct sockaddr_in *)&aliasreq.  
  ↪ ifra_addr;  
12    in->sin_family = AF_INET;  
13    in->sin_len     = sizeof(aliasreq.ifra_  
  ↪ addr);  
14    in->sin_addr.s_addr = inet_addr(pAddr);  
15  
16    /* initialize alias mask */  
17    in = (struct sockaddr_in *)&aliasreq.  
  ↪ ifra_mask;  
18    in->sin_family = AF_INET;  
19    in->sin_len     = sizeof(aliasreq.ifra_  
  ↪ mask);  
20    in->sin_addr.s_addr = inet_addr(pMask);  
21  
22    /* call to setup the alias */  
23    rtems_bsdnet_ifconfig(pName, SIOCAIFADDR,  
  ↪ &aliasreq);  
24 }
```

Thanks to Mike Seirs <<mailto:mikes@poliac.com>> for this example code.

16.5 Adding a Default Route

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

```

1 void mon_ifconfig(int argc, char *argv[],
2   ↪ unsigned32 command_arg, bool verbose)
3 {
4     struct sockaddr_in  ipaddr;
5     struct sockaddr_in  dstaddr;
6     struct sockaddr_in  netmask;
7     struct sockaddr_in  broadcast;
8     char                 *iface;
9     int                  f_ip      = 0;
10    int                   f_ptp     = 0;
11    int                   f_netmask = 0;
12    int                   f_up      = 0;
13    int                   f_down    = 0;
14    int                   f_bcast   = 0;
15    int                   cur_idx;
16    int                   rc;
17    int                   flags;
18
19    bzero((void*) &ipaddr, sizeof(ipaddr));
20    bzero((void*) &dstaddr, sizeof(dstaddr));
21    bzero((void*) &netmask, sizeof(netmask));
22    ↪ bzero((void*) &broadcast,
23    ↪ sizeof(broadcast));
24    ipaddr.sin_len = sizeof(ipaddr);
25    ipaddr.sin_family = AF_INET;
26    dstaddr.sin_len = sizeof(dstaddr);
27    dstaddr.sin_family = AF_INET;
28    netmask.sin_len = sizeof(netmask);
29    netmask.sin_family = AF_INET;
30    broadcast.sin_len = sizeof(broadcast);
31    broadcast.sin_family = AF_INET;
32    cur_idx = 0;
33
34    if (argc <= 1) {
35        /* display all interfaces */
36        iface = NULL;
37        cur_idx += 1;
38    } else {
39        iface = argv[1];
40        if (isdigit(*argv[2])) {
41            if (inet_pton(AF_INET, argv[2],
42            ↪ &ipaddr.sin_addr) < 0) {
43                printf("bad ip address: %s\n",
44                ↪ argv[2]);
45                return;
46            }
47            f_ip = 1;
48            cur_idx += 3;
49        } else {
50            cur_idx += 2;
51        }
52    }
53
54    while(argc > cur_idx) {
55        if (strcmp(argv[cur_idx], "up") ==
56        ↪ 0) {
57            f_up = 1;
58        } else if (strcmp(argv[cur_idx], "down
59        ↪") == 0) {
60            f_down = 1;
61        } else if (strcmp(argv[cur_idx], "netmask"
62        ↪") == 0) {
63            if ((cur_idx + 1) >= argc) {
64                printf("No netmask address\n");
65                return;
66            }
67            if (inet_pton(AF_INET, argv[cur_
68            ↪ idx+1], &netmask.sin_addr) < 0) {
69                printf("bad netmask: %s\n",
70                ↪ argv[cur_idx]);
71                return;
72            }
73            f_netmask = 1;
74            cur_idx += 1;
75        } else if (strcmp(argv[cur_idx], "broadcast"
76        ↪") == 0) {
77            if ((cur_idx + 1) >= argc) {
78                printf("No broadcast address\n");
79                return;
80            }
81            if (inet_pton(AF_INET, argv[cur_
82            ↪ idx+1], &broadcast.sin_addr) < 0) {
83                printf("bad broadcast: %s\n",
84                ↪ argv[cur_idx]);
85                return;
86            }
87            f_bcast = 1;
88            cur_idx += 1;
89        } else if (strcmp(argv[cur_idx], "pointopoint"
90        ↪") == 0) {
91            if ((cur_idx + 1) >= argc) {
92                printf("No pointopoint
93                ↪ address\n");
94                return;
95            }
96        }
97    }
98
99    if ((f_down != 0) && (f_ip != 0)) {
100        f_up = 1;
101    }
102
103    while(argc > cur_idx) {
104        if (strcmp(argv[cur_idx], "up") ==
105        ↪ 0) {
106            f_up = 1;
107        } else if (strcmp(argv[cur_idx], "down
108        ↪") == 0) {
109            f_down = 1;
110        } else if (strcmp(argv[cur_idx], "netmask"
111        ↪") == 0) {
112            if ((cur_idx + 1) >= argc) {
113                printf("No netmask address\n");
114                return;
115            }
116            if (inet_pton(AF_INET, argv[cur_
117            ↪ idx+1], &netmask.sin_addr) < 0) {
118                printf("bad netmask: %s\n",
119                ↪ argv[cur_idx]);
120                return;
121            }
122            f_netmask = 1;
123            cur_idx += 1;
124        } else if (strcmp(argv[cur_idx], "broadcast"
125        ↪") == 0) {
126            if ((cur_idx + 1) >= argc) {
127                printf("No broadcast address\n");
128                return;
129            }
130            if (inet_pton(AF_INET, argv[cur_
131            ↪ idx+1], &broadcast.sin_addr) < 0) {
132                printf("bad broadcast: %s\n",
133                ↪ argv[cur_idx]);
134                return;
135            }
136            f_bcast = 1;
137            cur_idx += 1;
138        } else if (strcmp(argv[cur_idx], "pointopoint"
139        ↪") == 0) {
140            if ((cur_idx + 1) >= argc) {
141                printf("No pointopoint
142                ↪ address\n");
143                return;
144            }
145        }
146    }
147
148    if ((f_down != 0) && (f_ip != 0)) {
149        f_up = 1;
150    }
151
152    while(argc > cur_idx) {
153        if (strcmp(argv[cur_idx], "up") ==
154        ↪ 0) {
155            f_up = 1;
156        } else if (strcmp(argv[cur_idx], "down
157        ↪") == 0) {
158            f_down = 1;
159        } else if (strcmp(argv[cur_idx], "netmask"
160        ↪") == 0) {
161            if ((cur_idx + 1) >= argc) {
162                printf("No netmask address\n");
163                return;
164            }
165            if (inet_pton(AF_INET, argv[cur_
166            ↪ idx+1], &netmask.sin_addr) < 0) {
167                printf("bad netmask: %s\n",
168                ↪ argv[cur_idx]);
169                return;
170            }
171            f_netmask = 1;
172            cur_idx += 1;
173        } else if (strcmp(argv[cur_idx], "broadcast"
174        ↪") == 0) {
175            if ((cur_idx + 1) >= argc) {
176                printf("No broadcast address\n");
177                return;
178            }
179            if (inet_pton(AF_INET, argv[cur_
180            ↪ idx+1], &broadcast.sin_addr) < 0) {
181                printf("bad broadcast: %s\n",
182                ↪ argv[cur_idx]);
183                return;
184            }
185            f_bcast = 1;
186            cur_idx += 1;
187        } else if (strcmp(argv[cur_idx], "pointopoint"
188        ↪") == 0) {
189            if ((cur_idx + 1) >= argc) {
190                printf("No pointopoint
191                ↪ address\n");
192                return;
193            }
194        }
195    }
196
197    if ((f_down != 0) && (f_ip != 0)) {
198        f_up = 1;
199    }
200
201    while(argc > cur_idx) {
202        if (strcmp(argv[cur_idx], "up") ==
203        ↪ 0) {
204            f_up = 1;
205        } else if (strcmp(argv[cur_idx], "down
206        ↪") == 0) {
207            f_down = 1;
208        } else if (strcmp(argv[cur_idx], "netmask"
209        ↪") == 0) {
210            if ((cur_idx + 1) >= argc) {
211                printf("No netmask address\n");
212                return;
213            }
214            if (inet_pton(AF_INET, argv[cur_
215            ↪ idx+1], &netmask.sin_addr) < 0) {
216                printf("bad netmask: %s\n",
217                ↪ argv[cur_idx]);
218                return;
219            }
220            f_netmask = 1;
221            cur_idx += 1;
222        } else if (strcmp(argv[cur_idx], "broadcast"
223        ↪") == 0) {
224            if ((cur_idx + 1) >= argc) {
225                printf("No broadcast address\n");
226                return;
227            }
228            if (inet_pton(AF_INET, argv[cur_
229            ↪ idx+1], &broadcast.sin_addr) < 0) {
230                printf("bad broadcast: %s\n",
231                ↪ argv[cur_idx]);
232                return;
233            }
234            f_bcast = 1;
235            cur_idx += 1;
236        } else if (strcmp(argv[cur_idx], "pointopoint"
237        ↪") == 0) {
238            if ((cur_idx + 1) >= argc) {
239                printf("No pointopoint
240                ↪ address\n");
241                return;
242            }
243        }
244    }
245
246    if ((f_down != 0) && (f_ip != 0)) {
247        f_up = 1;
248    }
249
250    while(argc > cur_idx) {
251        if (strcmp(argv[cur_idx], "up") ==
252        ↪ 0) {
253            f_up = 1;
254        } else if (strcmp(argv[cur_idx], "down
255        ↪") == 0) {
256            f_down = 1;
257        } else if (strcmp(argv[cur_idx], "netmask"
258        ↪") == 0) {
259            if ((cur_idx + 1) >= argc) {
260                printf("No netmask address\n");
261                return;
262            }
263            if (inet_pton(AF_INET, argv[cur_
264            ↪ idx+1], &netmask.sin_addr) < 0) {
265                printf("bad netmask: %s\n",
266                ↪ argv[cur_idx]);
267                return;
268            }
269            f_netmask = 1;
270            cur_idx += 1;
271        } else if (strcmp(argv[cur_idx], "broadcast"
272        ↪") == 0) {
273            if ((cur_idx + 1) >= argc) {
274                printf("No broadcast address\n");
275                return;
276            }
277            if (inet_pton(AF_INET, argv[cur_
278            ↪ idx+1], &broadcast.sin_addr) < 0) {
279                printf("bad broadcast: %s\n",
280                ↪ argv[cur_idx]);
281                return;
282            }
283            f_bcast = 1;
284            cur_idx += 1;
285        } else if (strcmp(argv[cur_idx], "pointopoint"
286        ↪") == 0) {
287            if ((cur_idx + 1) >= argc) {
288                printf("No pointopoint
289                ↪ address\n");
290                return;
291            }
292        }
293    }
294
295    if ((f_down != 0) && (f_ip != 0)) {
296        f_up = 1;
297    }
298
299    while(argc > cur_idx) {
300        if (strcmp(argv[cur_idx], "up") ==
301        ↪ 0) {
302            f_up = 1;
303        } else if (strcmp(argv[cur_idx], "down
304        ↪") == 0) {
305            f_down = 1;
306        } else if (strcmp(argv[cur_idx], "netmask"
307        ↪") == 0) {
308            if ((cur_idx + 1) >= argc) {
309                printf("No netmask address\n");
310                return;
311            }
312            if (inet_pton(AF_INET, argv[cur_
313            ↪ idx+1], &netmask.sin_addr) < 0) {
314                printf("bad netmask: %s\n",
315                ↪ argv[cur_idx]);
316                return;
317            }
318            f_netmask = 1;
319            cur_idx += 1;
320        } else if (strcmp(argv[cur_idx], "broadcast"
321        ↪") == 0) {
322            if ((cur_idx + 1) >= argc) {
323                printf("No broadcast address\n");
324                return;
325            }
326            if (inet_pton(AF_INET, argv[cur_
327            ↪ idx+1], &broadcast.sin_addr) < 0) {
328                printf("bad broadcast: %s\n",
329                ↪ argv[cur_idx]);
330                return;
331            }
332            f_bcast = 1;
333            cur_idx += 1;
334        } else if (strcmp(argv[cur_idx], "pointopoint"
335        ↪") == 0) {
336            if ((cur_idx + 1) >= argc) {
337                printf("No pointopoint
338                ↪ address\n");
339                return;
340            }
341        }
342    }
343
344    if ((f_down != 0) && (f_ip != 0)) {
345        f_up = 1;
346    }
347
348    while(argc > cur_idx) {
349        if (strcmp(argv[cur_idx], "up") ==
350        ↪ 0) {
351            f_up = 1;
352        } else if (strcmp(argv[cur_idx], "down
353        ↪") == 0) {
354            f_down = 1;
355        } else if (strcmp(argv[cur_idx], "netmask"
356        ↪") == 0) {
357            if ((cur_idx + 1) >= argc) {
358                printf("No netmask address\n");
359                return;
360            }
361            if (inet_pton(AF_INET, argv[cur_
362            ↪ idx+1], &netmask.sin_addr) < 0) {
363                printf("bad netmask: %s\n",
364                ↪ argv[cur_idx]);
365                return;
366            }
367            f_netmask = 1;
368            cur_idx += 1;
369        } else if (strcmp(argv[cur_idx], "broadcast"
370        ↪") == 0) {
371            if ((cur_idx + 1) >= argc) {
372                printf("No broadcast address\n");
373                return;
374            }
375            if (inet_pton(AF_INET, argv[cur_
376            ↪ idx+1], &broadcast.sin_addr) < 0) {
377                printf("bad broadcast: %s\n",
378                ↪ argv[cur_idx]);
379                return;
380            }
381            f_bcast = 1;
382            cur_idx += 1;
383        } else if (strcmp(argv[cur_idx], "pointopoint"
384        ↪") == 0) {
385            if ((cur_idx + 1) >= argc) {
386                printf("No pointopoint
387                ↪ address\n");
388                return;
389            }
390        }
391    }
392
393    if ((f_down != 0) && (f_ip != 0)) {
394        f_up = 1;
395    }
396
397    while(argc > cur_idx) {
398        if (strcmp(argv[cur_idx], "up") ==
399        ↪ 0) {
400            f_up = 1;
401        } else if (strcmp(argv[cur_idx], "down
402        ↪") == 0) {
403            f_down = 1;
404        } else if (strcmp(argv[cur_idx], "netmask"
405        ↪") == 0) {
406            if ((cur_idx + 1) >= argc) {
407                printf("No netmask address\n");
408                return;
409            }
410            if (inet_pton(AF_INET, argv[cur_
411            ↪ idx+1], &netmask.sin_addr) < 0) {
412                printf("bad netmask: %s\n",
413                ↪ argv[cur_idx]);
414                return;
415            }
416            f_netmask = 1;
417            cur_idx += 1;
418        } else if (strcmp(argv[cur_idx], "broadcast"
419        ↪") == 0) {
420            if ((cur_idx + 1) >= argc) {
421                printf("No broadcast address\n");
422                return;
423            }
424            if (inet_pton(AF_INET, argv[cur_
425            ↪ idx+1], &broadcast.sin_addr) < 0) {
426                printf("bad broadcast: %s\n",
427                ↪ argv[cur_idx]);
428                return;
429            }
430            f_bcast = 1;
431            cur_idx += 1;
432        } else if (strcmp(argv[cur_idx], "pointopoint"
433        ↪") == 0) {
434            if ((cur_idx + 1) >= argc) {
435                printf("No pointopoint
436                ↪ address\n");
437                return;
438            }
439        }
440    }
441
442    if ((f_down != 0) && (f_ip != 0)) {
443        f_up = 1;
444    }
445
446    while(argc > cur_idx) {
447        if (strcmp(argv[cur_idx], "up") ==
448        ↪ 0) {
449            f_up = 1;
450        } else if (strcmp(argv[cur_idx], "down
451        ↪") == 0) {
452            f_down = 1;
453        } else if (strcmp(argv[cur_idx], "netmask"
454        ↪") == 0) {
455            if ((cur_idx + 1) >= argc) {
456                printf("No netmask address\n");
457                return;
458            }
459            if (inet_pton(AF_INET, argv[cur_
460            ↪ idx+1], &netmask.sin_addr) < 0) {
461                printf("bad netmask: %s\n",
462                ↪ argv[cur_idx]);
463                return;
464            }
465            f_netmask = 1;
466            cur_idx += 1;
467        } else if (strcmp(argv[cur_idx], "broadcast"
468        ↪") == 0) {
469            if ((cur_idx + 1) >= argc) {
470                printf("No broadcast address\n");
471                return;
472            }
473            if (inet_pton(AF_INET, argv[cur_
474            ↪ idx+1], &broadcast.sin_addr) < 0) {
475                printf("bad broadcast: %s\n",
476                ↪ argv[cur_idx]);
477                return;
478            }
479            f_bcast = 1;
480            cur_idx += 1;
481        } else if (strcmp(argv[cur_idx], "pointopoint"
482        ↪") == 0) {
483            if ((cur_idx + 1) >= argc) {
484                printf("No pointopoint
485                ↪ address\n");
486                return;
487            }
488        }
489    }
490
491    if ((f_down != 0) && (f_ip != 0)) {
492        f_up = 1;
493    }
494
495    while(argc > cur_idx) {
496        if (strcmp(argv[cur_idx], "up") ==
497        ↪ 0) {
498            f_up = 1;
499        } else if (strcmp(argv[cur_idx], "down
500        ↪") == 0) {
501            f_down = 1;
502        } else if (strcmp(argv[cur_idx], "netmask"
503        ↪") == 0) {
504            if ((cur_idx + 1) >= argc) {
505                printf("No netmask address\n");
506                return;
507            }
508            if (inet_pton(AF_INET, argv[cur_
509            ↪ idx+1], &netmask.sin_addr) < 0) {
510                printf("bad netmask: %s\n",
511                ↪ argv[cur_idx]);
512                return;
513            }
514            f_netmask = 1;
515            cur_idx += 1;
516        } else if (strcmp(argv[cur_idx], "broadcast"
517        ↪") == 0) {
518            if ((cur_idx + 1) >= argc) {
519                printf("No broadcast address\n");
520                return;
521            }
522            if (inet_pton(AF_INET, argv[cur_
523            ↪ idx+1], &broadcast.sin_addr) < 0) {
524                printf("bad broadcast: %s\n",
525                ↪ argv[cur_idx]);
526                return;
527            }
528            f_bcast = 1;
529            cur_idx += 1;
530        } else if (strcmp(argv[cur_idx], "pointopoint"
531        ↪") == 0) {
532            if ((cur_idx + 1) >= argc) {
533                printf("No pointopoint
534                ↪ address\n");
535                return;
536            }
537        }
538    }
539
540    if ((f_down != 0) && (f_ip != 0)) {
541        f_up = 1;
542    }
543
544    while(argc > cur_idx) {
545        if (strcmp(argv[cur_idx], "up") ==
546        ↪ 0) {
547            f_up = 1;
548        } else if (strcmp(argv[cur_idx], "down
549        ↪") == 0) {
550            f_down = 1;
551        } else if (strcmp(argv[cur_idx], "netmask"
552        ↪") == 0) {
553            if ((cur_idx + 1) >= argc) {
554                printf("No netmask address\n");
555                return;
556            }
557            if (inet_pton(AF_INET, argv[cur_
558            ↪ idx+1], &netmask.sin_addr) < 0) {
559                printf("bad netmask: %s\n",
560                ↪ argv[cur_idx]);
561                return;
562            }
563            f_netmask = 1;
564            cur_idx += 1;
565        } else if (strcmp(argv[cur_idx], "broadcast"
566        ↪") == 0) {
567            if ((cur_idx + 1) >= argc) {
568                printf("No broadcast address\n");
569                return;
570            }
571            if (inet_pton(AF_INET, argv[cur_
572            ↪ idx+1], &broadcast.sin_addr) < 0) {
573                printf("bad broadcast: %s\n",
574                ↪ argv[cur_idx]);
575                return;
576            }
577            f_bcast = 1;
578            cur_idx += 1;
579        } else if (strcmp(argv[cur_idx], "pointopoint"
580        ↪") == 0) {
581            if ((cur_idx + 1) >= argc) {
582                printf("No pointopoint
583                ↪ address\n");
584                return;
585            }
586        }
587    }
588
589    if ((f_down != 0) && (f_ip != 0)) {
590        f_up = 1;
591    }
592
593    while(argc > cur_idx) {
594        if (strcmp(argv[cur_idx], "up") ==
595        ↪ 0) {
596            f_up = 1;
597        } else if (strcmp(argv[cur_idx], "down
598        ↪") == 0) {
599            f_down = 1;
600        } else if (strcmp(argv[cur_idx], "netmask"
601        ↪") == 0) {
602            if ((cur_idx + 1) >= argc) {
603                printf("No netmask address\n");
604                return;
605            }
606            if (inet_pton(AF_INET, argv[cur_
607            ↪ idx+1], &netmask.sin_addr) < 0) {
608                printf("bad netmask: %s\n",
609                ↪ argv[cur_idx]);
610                return;
611            }
612            f_netmask = 1;
613            cur_idx += 1;
614        } else if (strcmp(argv[cur_idx], "broadcast"
615        ↪") == 0) {
616            if ((cur_idx + 1) >= argc) {
617                printf("No broadcast address\n");
618                return;
619            }
620            if (inet_pton(AF_INET, argv[cur_
621            ↪ idx+1], &broadcast.sin_addr) < 0) {
622                printf("bad broadcast: %s\n",
623                ↪ argv[cur_idx]);
624                return;
625            }
626            f_bcast = 1;
627            cur_idx += 1;
628        } else if (strcmp(argv[cur_idx], "pointopoint"
629        ↪") == 0) {
630            if ((cur_idx + 1) >= argc) {
631                printf("No pointopoint
632                ↪ address\n");
633                return;
634            }
635        }
636    }
637
638    if ((f_down != 0) && (f_ip != 0)) {
639        f_up = 1;
640    }
641
642    while(argc > cur_idx) {
643        if (strcmp(argv[cur_idx], "up") ==
644        ↪ 0) {
645            f_up = 1;
646        } else if (strcmp(argv[cur_idx], "down
647        ↪") == 0) {
648            f_down = 1;
649        } else if (strcmp(argv[cur_idx], "netmask"
650        ↪") == 0) {
651            if ((cur_idx + 1) >= argc) {
652                printf("No netmask address\n");
653                return;
654            }
655            if (inet_pton(AF_INET, argv[cur_
656            ↪ idx+1], &netmask.sin_addr) < 0) {
657                printf("bad netmask: %s\n",
658                ↪ argv[cur_idx]);
659                return;
660            }
661            f_netmask = 1;
662            cur_idx += 1;
663        } else if (strcmp(argv[cur_idx], "broadcast"
664        ↪") == 0) {
665            if ((cur_idx + 1) >= argc) {
666                printf("No broadcast address\n");
667                return;
668            }
669            if (inet_pton(AF_INET, argv[cur_
670            ↪ idx+1], &broadcast.sin_addr) < 0) {
671                printf("bad broadcast: %s\n",
672                ↪ argv[cur_idx]);
673                return;
674            }
675            f_bcast = 1;
676            cur_idx += 1;
677        } else if (strcmp(argv[cur_idx], "pointopoint"
678        ↪") == 0) {
679            if ((cur_idx + 1) >= argc) {
680                printf("No pointopoint
681                ↪ address\n");
682                return;
683            }
684        }
685    }
686
687    if ((f_down != 0) && (f_ip != 0)) {
688        f_up = 1;
689    }
690
691    while(argc > cur_idx) {
692        if (strcmp(argv[cur_idx], "up") ==
693        ↪ 0) {
694            f_up = 1;
695        } else if (strcmp(argv[cur_idx], "down
696        ↪") == 0) {
697            f_down = 1;
698        } else if (strcmp(argv[cur_idx], "netmask"
699        ↪") == 0) {
700            if ((cur_idx + 1) >= argc) {
701                printf("No netmask address\n");
702                return;
703            }
704            if (inet_pton(AF_INET, argv[cur_
705            ↪ idx+1], &netmask.sin_addr) < 0) {
706                printf("bad netmask: %s\n",
707                ↪ argv[cur_idx]);
708                return;
709            }
710            f_netmask = 1;
711            cur_idx += 1;
712        } else if (strcmp(argv[cur_idx], "broadcast"
713        ↪") == 0) {
714            if ((cur_idx + 1) >= argc) {
715                printf("No broadcast address\n");
716                return;
717            }
718            if (inet_pton(AF_INET, argv[cur_
719            ↪ idx+1], &broadcast.sin_addr) < 0) {
720                printf("bad broadcast: %s\n",
721                ↪ argv[cur_idx]);
722                return;
723            }
724            f_bcast = 1;
725            cur_idx += 1;
726        } else if (strcmp(argv[cur_idx], "pointopoint"
727        ↪") == 0) {
728            if ((cur_idx + 1) >= argc) {
729                printf("No pointopoint
730                ↪ address\n");
731                return;
732            }
733        }
734    }
735
736    if ((f_down != 0) && (f_ip != 0)) {
737        f_up = 1;
738    }
739
740    while(argc > cur_idx) {
741        if (strcmp(argv[cur_idx], "up") ==
742        ↪ 0) {
743            f_up = 1;
744        } else if (strcmp(argv[cur_idx], "down
745        ↪") == 0) {
746            f_down = 1;
747        } else if (strcmp(argv[cur_idx], "netmask"
748        ↪") == 0) {
749            if ((cur_idx + 1) >= argc) {
750                printf("No netmask address\n");
751                return;
752            }
753            if (inet_pton(AF_INET, argv[cur_
754            ↪ idx+1], &netmask.sin_addr) < 0) {
755                printf("bad netmask: %s\n",
756                ↪ argv[cur_idx]);
757                return;
758            }
759            f_netmask = 1;
760            cur_idx += 1;
761        } else if (strcmp(argv[cur_idx], "broadcast"
762        ↪") == 0) {
763            if ((cur_idx + 1) >= argc) {
764                printf("No broadcast address\n");
765                return;
766            }
767            if (inet_pton(AF_INET, argv[cur_
768            ↪ idx+1], &broadcast.sin_addr) < 0) {
769                printf("bad broadcast: %s\n",
770                ↪ argv[cur_idx]);
771                return;
772            }
773            f_bcast = 1;
774            cur_idx += 1;
775        } else if (strcmp(argv[cur_idx], "pointopoint"
776        ↪") == 0) {
777            if ((cur_idx + 1) >= argc) {
778                printf("No pointopoint
779                ↪ address\n");
780                return;
781            }
782        }
783    }
784
785    if ((f_down != 0) && (f_ip != 0)) {
786        f_up = 1;
787    }
788
789    while(argc > cur_idx) {
790        if (strcmp(argv[cur_idx], "up") ==
791        ↪ 0) {
792            f_up = 1;
793        } else if (strcmp(argv[cur_idx], "down
794        ↪") == 0) {
795            f_down = 1;
796        } else if (strcmp(argv[cur_idx], "netmask"
797        ↪") == 0) {
798            if ((cur_idx + 1) >= argc) {
799                printf("No netmask address\n");
800                return;
801            }
802            if (inet_pton(AF_INET, argv[cur_
803            ↪ idx+1], &netmask.sin_addr) < 0) {
804                printf("bad netmask: %s\n",
805                ↪ argv[cur_idx]);
806                return;
807            }
808            f_netmask = 1;
809            cur_idx += 1;
810        } else if (strcmp(argv[cur_idx], "broadcast"
811        ↪") == 0) {
812            if ((cur_idx + 1) >= argc) {
813                printf("No broadcast address\n");
814                return;
815            }
816            if (inet_pton(AF_INET, argv[cur_
817            ↪ idx+1], &broadcast.sin_addr) < 0) {
818                printf("bad broadcast: %s\n",
819                ↪ argv[cur_idx]);
820                return;
821            }
822            f_bcast = 1;
823            cur_idx += 1;
824        } else if (strcmp(argv[cur_idx], "pointopoint"
825        ↪") == 0) {
826            if ((cur_idx + 1) >= argc) {
827                printf("No pointopoint
828                ↪ address\n");
829                return;
830            }
831        }
832    }
833
834    if ((f_down != 0) && (f_ip != 0)) {
835        f_up = 1;
836    }
837
838    while(argc > cur_idx) {
839        if (strcmp(argv[cur_idx], "up") ==
840        ↪ 0) {
841            f_up = 1;
842        } else if (strcmp(argv[cur_idx], "down
843        ↪") == 0) {
844            f_down = 1;
845        } else if (strcmp(argv[cur_idx], "netmask"
846        ↪") == 0) {
847            if ((cur_idx + 1) >= argc) {
848                printf("No netmask address\n");
849                return;
850            }
851            if (inet_pton(AF_INET, argv[cur_
852            ↪ idx+1], &netmask.sin_addr) < 0) {
853                printf("bad netmask: %s\n",
854                ↪ argv[cur_idx]);
855                return;
856            }
857            f_netmask = 1;
858            cur_idx += 1;
859        } else if (strcmp(argv[cur_idx], "broadcast"
860        ↪") == 0) {
861            if ((cur_idx + 1) >= argc) {
862                printf("No broadcast address\n");
863                return;
864            }
865            if (inet_pton(AF_INET, argv[cur_
866            ↪ idx+1], &broadcast.sin_addr) < 0) {
867                printf("bad broadcast: %s\n",
868                ↪ argv[cur_idx]);
869                return;
870            }
871            f_bcast = 1;
872            cur_idx += 1;
873        } else if (strcmp(argv[cur_idx], "pointopoint"
874        ↪") == 0) {
875            if ((cur_idx + 1) >= argc) {
876                printf("No pointopoint
877                ↪ address\n");
878                return;
879            }
880        }
881    }
882
883    if ((f_down != 0) && (f_ip != 0)) {
884        f_up = 1;
885    }
886
887    while(argc > cur_idx) {
888        if (strcmp(argv[cur_idx], "up") ==
889        ↪ 0) {
890            f_up = 1;
891        } else if (strcmp(argv[cur_idx], "down
892        ↪") == 0) {
893            f_down = 1;
894        } else if (strcmp(argv[cur_idx], "netmask"
895        ↪") == 0) {
896            if ((cur_idx + 1) >= argc) {
897                printf("No netmask address\n");
898                return;
899            }
900            if (inet_pton(AF_INET, argv[cur_
901            ↪ idx+1], &netmask.sin_addr) < 0) {
902                printf("bad netmask: %s\n",
903                ↪ argv[cur_idx]);
904                return;
905            }
906            f_netmask = 1;
907            cur_idx += 1;
908        } else if (strcmp(argv[cur_idx], "broadcast"
909        ↪") == 0) {
910            if ((cur_idx + 1) >= argc) {
911                printf("No broadcast address\n");
912                return;
913            }
914            if (inet_pton(AF_INET, argv[cur_
915            ↪ idx+1], &broadcast.sin_addr) < 0) {
916                printf("bad broadcast: %s\n",
917                ↪ argv[cur_idx]);
918                return;
919            }
920            f_bcast = 1;
921            cur_idx += 1;
922        } else if (strcmp(argv[cur_idx], "pointopoint"
923        ↪") == 0) {
924            if ((cur_idx + 1) >= argc) {
925                printf("No pointopoint
926                ↪ address\n");
927                return;
928            }
929        }
930    }
931
932    if ((f_down != 0) && (f_ip != 0)) {
933        f_up = 1;
934    }
935
936    while(argc > cur_idx) {
937        if (strcmp(argv[cur_idx], "up") ==
938        ↪ 0) {
939            f_up = 1;
940        } else if (strcmp(argv[cur_idx], "down
941        ↪") == 0) {
942            f_down = 1;
943        } else if (strcmp(argv[cur_idx], "netmask"
944        ↪") == 0) {
945            if ((cur_idx + 1) >= argc) {
946                printf("No netmask address\n");
947                return;
948            }
949            if (inet_pton(AF_INET, argv[cur_
950            ↪ idx+1], &netmask.sin_addr) < 0) {
951                printf("bad netmask: %s\n",
952                ↪ argv[cur_idx]);
953                return;
954            }
955            f_netmask = 1;
956            cur_idx += 1;
957        } else if (strcmp(argv[cur_idx], "broadcast"
958        ↪") == 0) {
959            if ((cur_idx + 1) >= argc) {
960                printf("No broadcast address\n");
961                return;
962            }
963            if (inet_pton(AF_INET, argv[cur_
964            ↪ idx+1], &broadcast.sin_addr) < 0) {
965                printf("bad broadcast: %s\n",
966                ↪ argv[cur_idx]);
967                return;
968            }
969            f_bcast = 1;
970            cur_idx += 1;
971        } else if (strcmp(argv[cur_idx], "pointopoint"
972        ↪") == 0) {
973            if ((cur_idx + 1) >= argc) {
974                printf("No pointopoint
975                ↪ address\n");
976                return;
977            }
978        }
979    }
980
981    if ((f_down != 0) && (f_ip != 0)) {
982        f_up = 1;
983    }
984
985    while(argc > cur_idx) {
986        if (strcmp(argv[cur_idx], "up") ==
987        ↪ 0) {
988            f_up = 1;
989        } else if (strcmp(argv[cur_idx], "down
990        ↪") == 0) {
991            f_down = 1;
992        } else if (strcmp(argv[cur_idx], "netmask"
993        ↪") == 0) {
994            if ((cur_idx + 1) >= argc) {
995                printf("No netmask address\n");
996                return;
997            }
998            if (inet_pton(AF_INET, argv[cur_
999            ↪ idx+1], &netmask.sin_addr) < 0) {
1000                printf("bad netmask: %s\n",
1001                ↪ argv[cur_idx]);
1002                return;
```

```

91         }
92         if (inet_pton(AF_INET, argv[cur_idx+1], &dstaddr.sin_addr) < 0) {
93             printf("bad pointopoint: %s\n", argv[cur_idx]);
94             return;
95         }
96         f_ptp = 1;
97         cur_idx += 1;
98     } else {
99         printf("Bad parameter: %s\n", argv[cur_idx]);
100         return;
101     }
102     cur_idx += 1;
103 }
104
105 printf("ifconfig ");
106
107 if (iface != NULL) {
108     printf("%s ", iface);
109     if (f_ip != 0) {
110         char str[256];
111         inet_ntop(AF_INET, &ipaddr.sin_addr, str, 256);
112         printf("%s ", str);
113     }
114     if (f_netmask != 0) {
115         char str[256];
116         inet_ntop(AF_INET, &netmask.sin_addr, str, 256);
117         printf("netmask %s ", str);
118     }
119     if (f_bcast != 0) {
120         char str[256];
121         inet_ntop(AF_INET, &broadcast.sin_addr, str, 256);
122         printf("broadcast %s ", str);
123     }
124     if (f_ptp != 0) {
125         char str[256];
126         inet_ntop(AF_INET, &dstaddr.sin_addr, str, 256);
127         printf("pointopoint %s ", str);
128     }
129     if (f_up != 0) {
130         printf("up\n");
131     } else if (f_down != 0) {
132         printf("down\n");
133     } else {
134         printf("\n");
135     }
136 }
137
138 if ((iface == NULL) || ((f_ip == 0) && (f_down == 0) && (f_up == 0))) {
139     rtems_bsdnet_show_if_stats();
140
141     return;
142 }
143
144 flags = 0;
145 if (f_netmask) {
146     rc = rtems_bsdnet_ifconfig(iface, SIOCSIFNETMASK, &netmask);
147     if (rc < 0) {
148         printf("Could not set netmask: %s\n", strerror(errno));
149         return;
150     }
151 }
152 if (f_bcast) {
153     rc = rtems_bsdnet_ifconfig(iface, SIOCSIFBRDADDR, &broadcast);
154     if (rc < 0) {
155         printf("Could not set broadcast: %s\n", strerror(errno));
156         return;
157     }
158 }
159 if (f_ptp) {
160     rc = rtems_bsdnet_ifconfig(iface, SIOCSIFDSTADDR, &dstaddr);
161     if (rc < 0) {
162         printf("Could not set destination address: %s\n", strerror(errno));
163         return;
164     }
165     flags |= IFF_POINTOPOINT;
166 }
167
168 /* This must come _after_ setting the netmask, broadcast addresses */
169 if (f_ip) {
170     rc = rtems_bsdnet_ifconfig(iface, SIOCSIFADDR, &ipaddr);
171     if (rc < 0) {
172         printf("Could not set IP address: %s\n", strerror(errno));
173         return;
174     }
175 }
176 if (f_up != 0) {
177     flags |= IFF_UP;
178 }
179 if (f_down != 0) {
180     printf("Warning: taking interfaces down is not supported\n");
181 }
182
183 rc = rtems_bsdnet_ifconfig(iface, SIOCSIFFLAGS, &flags);
184 if (rc < 0) {
185     printf("Could not set interface flags: %s\n", strerror(errno));

```

```

185     return;
186 }
187 }
188
189 void mon_route(int argc, char *argv[],
190               ↪ unsigned32 command_arg, bool verbose)
191 {
192     int cmd;
193     struct sockaddr_in dst;
194     struct sockaddr_in gw;
195     struct sockaddr_in netmask;
196     int f_host;
197     int f_gw = 0;
198     int cur_idx;
199     int flags;
200     int rc;
201
202     memset(&dst, 0, sizeof(dst));
203     memset(&gw, 0, sizeof(gw));
204     memset(&netmask, 0, sizeof(netmask));
205     dst.sin_len = sizeof(dst);
206     dst.sin_family = AF_INET;
207     ↪ dst.sin_addr.s_addr = inet_addr("0.0.0.0");
208     gw.sin_len = sizeof(gw);
209     gw.sin_family = AF_INET;
210     ↪ gw.sin_addr.s_addr = inet_addr("0.0.0.0");
211     netmask.sin_len = sizeof(netmask);
212     netmask.sin_family = AF_INET;
213     ↪ netmask.sin_addr.s_addr = inet_addr("255.255.0.0");
214
215     if (argc < 2) {
216         rtems_bsdnet_show_inet_routes();
217         return;
218     }
219
220     if (strcmp(argv[1], "add") == 0) {
221         cmd = RTM_ADD;
222     } else if (strcmp(argv[1], "del") == 0) {
223         cmd = RTM_DELETE;
224     } else {
225         printf("invalid command: %s\n", ↪ argv[1]);
226         printf("\tit should be 'add' or 'del'\n");
227         return;
228     }
229
230     if (argc < 3) {
231         printf("not enough arguments\n");
232         return;
233     }
234
235     if (strcmp(argv[2], "-host") == 0) {
236         f_host = 1;
237     } else if (strcmp(argv[2], "-net") == 0) {
238         ↪ f_host = 0;
239     } else {
240         printf("Invalid type: %s\n", argv[1]);
241         printf("\tit should be '-host' or '-net'\n");
242         return;
243     }
244
245     if (argc < 4) {
246         printf("not enough arguments\n");
247         return;
248     }
249
250     inet_pton(AF_INET, argv[3], &dst.sin_↪ addr);
251
252     cur_idx = 4;
253     while (cur_idx < argc) {
254         if (strcmp(argv[cur_idx], "gw") == ↪ 0) {
255             if ((cur_idx + 1) >= argc) {
256                 printf("no gateway address\n↪ ");
257                 return;
258             }
259             f_gw = 1;
260             inet_pton(AF_INET, argv[cur_idx_↪ + 1], &gw.sin_addr);
261             cur_idx += 1;
262         } else if (strcmp(argv[cur_idx], ↪ "netmask") == 0) {
263             if ((cur_idx + 1) >= argc) {
264                 printf("no netmask address\n↪ ");
265                 return;
266             }
267             f_gw = 1;
268             inet_pton(AF_INET, argv[cur_idx_↪ + 1], &netmask.sin_addr);
269             cur_idx += 1;
270         } else {
271             printf("Unknown argument\n");
272             return;
273         }
274         cur_idx += 1;
275     }
276
277     flags = RTF_STATIC;
278     if (f_gw != 0) {
279         flags |= RTF_GATEWAY;
280     }
281     if (f_host != 0) {
282         flags |= RTF_HOST;
283     }

```



```
284     rc = rtems_bsdnet_rtrequest(cmd, &dst, &  
    ↪ gw, &netmask, flags, NULL);  
285     if (rc < 0) {  
286         printf("Error adding route\n");  
287     }  
288 }
```

Thanks to Jay Monkman <<mailto:jtm@smoothmsmoothie.com>> for this example code.

16.6 Time Synchronization Using NTP

```
1 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.

Part V

Testing the Driver

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.

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 enqueueing 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.

MONITOR COMMANDS

There are a number of command available in the shell / monitor to aid in tracing the behavior of the network stack. The following is a list of them:

- **inet** This command shows the current routing information for the TCP/IP stack. Following is an example showing the output of this command.

1	Destination	Gateway/Mask/Hw			
	↪Flags	Refs	Use	Expire	
	↪Interface				
2	10.0.0.0	255.0.0.0		U	
	↪	0	17 smc1		
3	127.0.0.1	127.0.0.1		UH	
	↪	0	0 lo0		

In this example, there is only one network interface with an IP address of 10.8.1.1. This link is currently not up. Two routes that are shown are the default routes for the Ethernet interface (10.0.0.0) and the loopback interface (127.0.0.1). Since the stack comes from BSD, this command is very similar to the netstat command. For more details on the network routing please look the following URL: (http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/network-routing.html) For a quick reference to the flags, see the table below:

‘U’

Up: The route is active.

‘H’

Host: The route destination is a single host.

‘G’

Gateway: Send anything for this destination on to this remote system, which will figure out from there where to

send it.

‘S’

Static: This route was configured manually, not automatically generated by the system.

‘C’

Clone: Generates a new route based upon this route for machines we connect to. This type of route is normally used for local networks.

‘W’

WasCloned: Indicated a route that was auto-configured based upon a local area network (Clone) route.

‘L’

Link: Route involves references to Ethernet hardware.

- **mbuf** This command shows the current MBUF statistics. An example of the command is shown below:

1	*****	MBUF	STATISTICS	
	↪*****			
2	mbufs:4096	clusters: 256	free:	
	↪241			
3	drops: 0	waits: 0	drains: 0	
4	free:4080	data:16		
	↪header:0	socket:0		
5	pcb:0	rtable:0		
	↪htable:0	atable:0		
6	soname:0	soopts:0		
	↪ftable:0	rights:0		
7	ifaddr:0	control:0		
	↪oobdata:0			

- **if** This command shows the current statistics for your Ethernet driver as long as the ioctl hook SIO_RTEMS_SHOW_STATS has been implemented. Below is an example:

```

1 ***** INTERFACE STATISTICS
2 *****
3 ***** smc1 *****
4 Ethernet Address: 00:12:76:43:34:25
5 Address:10.8.1.1 Broadcast
6 Address:10.255.255.255 Net mask:
7 255.0.0.0
8 Flags: Up Broadcast Running Simplex
9 Send queue limit:50 length:0
10 Dropped:0
11 SMC91C111 RTEMS driver A0.01 11/03/
12 2002 Ian Caddy (ianc@microsol.iinet.
13 net.au)
14 Rx Interrupts:0 Not First:
15 0 Not Last:0
16 Giant:0 Runt:
17 0 Non-octet:0
18 Bad CRC:0 Overrun:
19 0 Collision:0
20 Tx Interrupts:2 Deferred:
21 0 Missed Hearbeat:0
22 No Carrier:0 Retransmit Limit:
23 0 Late Collision:0
24 Underrun:0 Raw output wait:
25 0 Coalesced:0
26 Coalesce failed:0
27 Retries:0
28 ***** lo0 *****
29 Address:127.0.0.1 Net mask:255.0.
30 0.0
31 Flags: Up Loopback Running Multicast
32 Send queue limit:50 length:0
33 Dropped:0

```

- `ip` This command show the IP statistics for the currently configured interfaces.
- `icmp` This command show the ICMP statistics for the currently configured interfaces.
- `tcp` This command show the TCP statistics for the currently configured interfaces.
- `udp` This command show the UDP statistics for the currently configured interfaces.

DRIVER BASIC OPERATION

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

or more workstations on your network.

- 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

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.

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.

22.1 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.

22.2 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.

22.3 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.

22.4 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.

Part VI

Network Servers

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).

23.1 Configuration Parameters

The configuration structure for FTPD is as follows:

```
1 struct rtems_ftpd_configuration
2 {
3     rtems_task_priority    priority;
4     ↪ /* FTPD task priority */
5     unsigned long          max_hook_filesize;
6     ↪ /* Maximum buffersize */
7     ↪ /*   for hooks   */
8     int                   port;
9     ↪ /* Well-known port */
10    struct rtems_ftpd_hook *hooks;
11    ↪ /* List of hooks   */
12 };
```

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.

23.2 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 follow.

```
1 struct rtems_ftpd_hook ftp_hooks[] =
2 {
3     {"untar", Untar_FromMemory},
4     {NULL, NULL}
5 };
6
7 struct rtems_ftpd_configuration rtems_ftpd_
8   ↪ configuration =
9 {
10     40,                               /* FTPD task_
11   ↪ priority */
12     512*1024,                         /* Maximum hook
13   ↪ 'file' size */
14     0,                               /* Use default_
15   ↪ port */
16     ftp_hooks                         /* Local ftp_
17   ↪ hooks */
18 };
```

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

23.3 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:

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

An example FTP transcript which exercises this hook is:

```
1 220 RTEMS FTP server (Version 1.0-JWJ) ready.
2 Name (dcomm0:janovetz): John Galt
3 230 User logged in.
4 Remote system type is RTEMS.
5 ftp> bin
6 200 Type set to I.
7 ftp> dir
8 200 PORT command successful.
9 150 ASCII data connection for LIST.
10 drwxrwx--x      0      0          268 dev
11 drwxrwx--x      0      0           0 TFTP
12 226 Transfer complete.
13 ftp> put html.tar untar
14 local: html.tar remote: untar
15 200 PORT command successful.
16 150 BINARY data connection.
17 210 File transferred successfully.
18 471040 bytes sent in 0.48 secs (9.6e+02 ↪
   ↪Kbytes/sec)
19 ftp> dir
20 200 PORT command successful.
21 150 ASCII data connection for LIST.
22 drwxrwx--x      0      0          268 dev
23 drwxrwx--x      0      0           0 TFTP
24 drwxrwx--x      0      0         3484 public_
   ↪html
25 226 Transfer complete.
26 ftp> quit
27 221 Goodbye.
```

Part VII

DEC 21140 Driver

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 .

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.

DEC21140 PCI BOARD GENERALITIES

This chapter describes rapidly 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 `pci_initialize()` 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 Figure). 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 detailed here. You can find them in *DIGITAL Semiconductor 21140A PCI Fast Ethernet LAN Controller - Hardware Reference Manual*.

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.

27.1 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 outgoing frame to send on the physical link.

27.2 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.

OWN	Status	
	Control bits	Byte-Count Buffer 2
		Byte-Count Buffer 1
Buffer address 1		
Buffer address 2		

27.3 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).

27.4 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.

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 outgoing 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 : $(7+1) \times (1520+16) = 12288$ 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.

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.

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 transceiver. Some ZX345 cards made in 1996 have an ICS 1890 transceiver instead.
- ZNYX ZX348 Two 21140-A chips using ICS 1890 transceivers and either a 21052

or 21152 bridge. Early versions used National 83840 transceivers, but later versions are depopulated ZX346 boards.

- ZNYX ZX351 21140 chip with a Broadcom 100BaseT4 transceiver.

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*.*

Part VIII

Command and Variable Index

There are currently no Command and Variable Index entries.

- `genindex`
- `search`