RTEMS Network Supplement

Edition 4.0.0, for RTEMS 4.0.0

October 1998

On-Line Applications Research Corporation

On-Line Applications Research Corporation
TEXinfo 1.1.1.1

Any inquiries concerning RTEMS, its related support components, or its documentation should be directed to either:

On-Line Applications Research Corporation
4910-L Corporate Drive
Huntsville, AL 35805
VOICE: (256) 722-9985
FAX: (256) 722-0985
EMAIL: rtems@OARcorp.com

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://wwwhost.ots.utexas.edu/ethernet/index.html)

- *TCP/IP Illustrated, Volume 1 : The Protocols* by by W. Richard Stevens (ISBN: 0201633469)

  This book provides detailed introduction to TCP/IP and includes diagnostic programs which are publicly available.

- *TCP/IP Illustrated : 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.

1 Network Task Structure and Data Flow

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

```
NO TEX VERSION OF THE TASKING FIGURE IS AVAILABLE
```

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

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

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

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

2 Writing RTEMS Network Device Drivers

2.1 Introduction

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

2.2 Learn about the network device

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

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

2.3 Understand the network scheduling conventions

When writing code for your driver transmit and receive tasks you must 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`). Your transmit and receive tasks must abide by this protocol which means you must be careful to avoid 'deadly embraces' with the other network tasks. A number of routines are provided to make it easier for your code to conform to the network task scheduling conventions.

- `void rtems_bsdnet_semaphore_release(void)`
  This function releases the network semaphore. Your task must call this function immediately before making any blocking RTEMS request.
- `void rtems_bsdnet_semaphore_obtain(void)`
  This function obtains the network semaphore. If your task has released the network semaphore to allow other network-related tasks to run while your task blocks you must call this function 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 *)` Your task should call this function when it wishes to wait for an event. This function releases the network semaphore, calls `rtems_event_receive` to wait for the specified event or events and reobtains the semaphore. The value returned is the value returned by the `rtems_event_receive`.

2.4 Write your 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 name should not contain digits as these will be assumed to be part of the unit number and not part of the device name.

`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`'.

`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`.

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

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

2.5 Write your driver start function.

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

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

2.6 Write your driver initialization function.

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

```
rtems_id
rtems_bsdnet_newproc (char *name,
      int stacksize,
```

```
void(*entry)(void *),
void *arg);
```

should be used to start the driver tasks.

Note that the network stack may call the driver initialization function more than once. Make sure you don't start multiple versions of the receive and transmit tasks.

2.7 Write your driver transmit task.

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

2.8 Write your driver receive task.

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

2.9 Write your driver interrupt handler.

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

2.10 Write your driver ioctl function.

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

`SIOCGIFADDR`

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

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

- `IFF_RUNNING` Stop the device.
- `IFF_UP` Start the device.

`IFF_UP|IFF_RUNNING`	Stop then start the device.
`0`	Do nothing.

2.11 Write Your Driver Statistic-Printing Function

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

3 Using Networking in an RTEMS Application

3.1 Makefile changes

3.1.1 Including the required managers

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

```
MANAGERS = io event semaphore
```

3.1.2 Increasing the size of the heap

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

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

This sets aside 512 kbytes of memory for the heap.

3.2 System Configuration

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

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

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

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

3.3 Initialization

3.3.1 Additional include files

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

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

3.3.2 Network configuration

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

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

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

`void (*bootp)(void)`
: This entry should be set to `rtems_bsdnet_do_bootp` if your application will use BOOTP to obtain network configuration information. It should be set to `NULL` if your application does not use BOOTP.

`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 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.

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

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

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

3.3.3 Network device configuration

Network devices are specified and configured by declaring and initializing a `struct rtems_bsdnet_ifcontig` 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 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 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 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 server, and uses the default values for all driver parameters.

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

3.3.4 Network initialization

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

```
rtems_bsdnet_initialize_network ();
```

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

3.4 Application code

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.
- There is no `select` function.
- 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).

3.4.1 Network Statistics

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

`rtems_bsdnet_show_if_stats`
: Display statistics gathered by network interfaces.

`rtems_bsdnet_show_ip_stats`
: Display IP packet statistics.

`rtems_bsdnet_show_icmp_stats`
: Display ICMP packet statistics.

`rtems_bsdnet_show_tcp_stats`
: Display TCP packet statistics.

`rtems_bsdnet_show_udp_stats`
: Display UDP packet statistics.

`rtems_bsdnet_show_mbuf_stats`
: Display mbuf statistics.

`rtems_bsdnet_show_inet_routes`
: Display the routing table.

4 Testing the Driver

4.1 Preliminary Setup

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

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

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

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

4.2 Driver basic operation

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

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

  Verify that the program continues to run once the driver has been attached.
- Issue a '`u`' command to send UDP packets to the 'discard' port. Verify that the packets appear on the network.
- Issue a '`s`' command to print the network and driver statistics.
- On a workstation, add a static route to the target system.
- On that same workstation try to 'ping' the target system. Verify that the ICMP echo request and reply packets appear on the net.
- Remove the static route to the target system. Modify `networkconfig.h` to attach the driver with reception of broadcast packets enabled. Try to 'ping' the target system again. Verify that ARP request/reply and ICMP echo request/reply packets appear on the net.

- Issue a '`t`' command to send TCP packets to the 'discard' port. Verify that the packets appear on the network.
- Issue a '`s`' command to print the network and driver statistics.
- Verify that you can telnet to ports 24742 and 24743 on the target system from one or more workstations on your network.

4.3 BOOTP operation

Set up a BOOTP 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 server and that all the above tests succeed.

4.4 Stress Tests

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

4.4.1 Giant packets

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

4.4.2 Resource Exhaustion

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

4.4.3 Cable Faults

- Run the `netdemo` program.

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

4.4.4 Throughput

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

Command and Variable Index

There are currently no Command and Variable Index entries.

Concept Index

There are currently no Concept Index entries.