RNET Manual

For NUFR Version 0.1

Copyright © 2018, Bernie Woodland

All rights reserved.

Redistribution and use, with or without

modification, are permitted provided that the following conditions are met:

1. Redistributions must retain the above copyright notice, this

list of conditions and the following disclaimer.

THIS DOCUMENTATION IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE

DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;

LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENTATION, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Arm® is a trademark of Arm Holdings

Arm® Cortex® is a trademark of Arm HoldingsTable of Contents

[Feature Set 4](#_Toc516329162)

[The Networking Model 5](#_Toc516329163)

[Interfaces and Subinterfaces 5](#_Toc516329164)

[Circuits 5](#_Toc516329165)

[RNET Buffers and Particle Chains 6](#_Toc516329166)

[The Packet Buffering Model 8](#_Toc516329167)

[Using Packet Windows 8](#_Toc516329168)

[Preparing and Transmitting Packets in Host-Mode, Overview 10](#_Toc516329169)

[Reading Data from an RNET Data Buffer 11](#_Toc516329170)

[Reading Data from a Particle 12](#_Toc516329171)

[Setting RNET Buffers in Client Mode 13](#_Toc516329172)

[Setting Particle Chains in Client Mode 14](#_Toc516329173)

[Servicing Packets in Server Mode 17](#_Toc516329174)

[Writing to Packet Buffers in Drivers 18](#_Toc516329175)

[Finding Circuit Indices 18](#_Toc516329176)

[RNET’s Packet Flow Model 20](#_Toc516329177)

[RNET Tasking Environment 20](#_Toc516329178)

[Packet Injection Messages 21](#_Toc516329179)

[These are the messages used for sending packets to RNET 21](#_Toc516329180)

[These will be used as parameters to rnet\_msg\_send(). Configuration 22](#_Toc516329181)

[Configuration 23](#_Toc516329182)

[Configuration Elements 23](#_Toc516329183)

[Configuring NUFR 24](#_Toc516329184)

[The RNET File Structure 24](#_Toc516329185)

[Configuring Interfaces and Subinterfaces 25](#_Toc516329186)

[The ROM Struct Interface Fields 25](#_Toc516329187)

[The ROM Struct Subinterface Fields 26](#_Toc516329188)

[Interface and Subinterface ROM Settings Example 26](#_Toc516329189)

[Messaging Configuration Items 30](#_Toc516329190)

[Dummy/No-Send-Message Value 30](#_Toc516329191)

[Example 30](#_Toc516329192)

[Expanded Multi-message Example 31](#_Toc516329193)

[Circuit Configuration 32](#_Toc516329194)

[RNET APIs 33](#_Toc516329195)

[Running PPP 34](#_Toc516329196)

[Running PPP on Linux 34](#_Toc516329197)

[Notes on Interface Drivers 35](#_Toc516329198)

Feature Set

The Raging Networking Stack (RNET) is an IP stack built using NUFR kernel and NUFR SL APIs. It features the following:

* A task-less, message-based API
* A packet buffering strategy
* Support for multiple interfaces
* Static and dynamic parameter configuration
* Socket-like constructs called *circuits*
* Extensible protocol support

RNET supports the following protocols:

* HDLC (AHDLC)
* PPP over AHDLC
* IPv4
* IPv6
* UDP
* ICMP for both IPv4 and IPv6 (echo request/response only)

Future releases of RNET may add support for Ethernet (with ARP and IPv6 Neighbor Discover), ICMP error messaging, routing, and who knows what else.

# The Networking Model

RNET has a few components that are used differently, slightly differently, or entirely differently than the way other routers or other networking stacks use them. This will explain how RNET defines them.

## Interfaces and Subinterfaces

Interfaces are statically configured: they are hard-coded and cannot be added to or deleted. There must be one interface, minimum, and there can be up to *N* interfaces. Normally, an interface corresponds to a physical interface, such as a serial port, a USB port, or an Ethernet port, but this is not a rule; interfaces can exist which have no physical counterpart.

Packets have one or more *subinterfaces* attached to them. An RNET subinterface is not the same as a router subinterface. An RNET subinterface is an individual IP address assigned to an interface. Subinterfaces are assigned IP addresses, and interfaces are not assigned IP addresses. Therefore, for each IP address associated with an interface, there must be a subinterface which owns that IP address. There is a fixed limit to the number of subinterfaces which can be assigned to an interface. Subinterfaces, like interfaces, are also statically defined, but their IP addresses can be modified at run-time.

In any networking topology, multiple IP addresses can be assigned to a single interface, and RNET supports this model through the use of subinterfaces. For example, one may want to configure a single IPv4 address to an interface and a single IPv6 link-local interface to the same interface. In RNET, this would require two subinterfaces: one for the IPv4 and one for the link-local.

## Circuits

*Circuits* are a concept unique to RNET. A circuit takes the place of a socket in a Unix/Linux programming environment. Packets injected into RNET must be marked as being sent out a specific circuit, and packets received into an interface must be matched to an existing circuit. It is through the use of circuits that packets are mapped to applications.

There is provision for statically defining circuits, but, in general, circuits are dynamically (i.e. at run-time) configured. There is a system-configurable limit to the number of circuits which can be configured, and a circuit can be added or deleted at runtime. A static circuit can be deleted at runtime as well.

Circuits must be attached to a subinterface, and can only be attached to a single subinterface. Any given subinterface may have zero to N circuits attached to it. Circuits specify peer (the opposite host which the device running RNET is talking to) IP addresses and UDP port numbers (and TCP port numbers too, if TCP support is ever added). The circuit specifies the peer IP address, the self and peer UDP ports. The fourth missing piece is the self IP address: this is specified by the subinterface which the circuit attaches to.

The IP addresses and UDP port numbers which are configured on a circuit can be either specific values or can be null values. A null value (an IP address of “0.0.0.0” for IPv4; “::” for IPv6; a UDP port number of “0”) is interpreted as a wildcard match by RNET. This enables one to configure circuits which will operate in either client mode or in server mode. In client mode, the circuit has non-null values configured. When a packet is sent by an application into RNET to be sent to a peer host in client mode, the packet is marked with the circuit is to be sent on, and RNET uses the circuit settings to determine destination IP address and port numbers. When a packet is received from a server, and it is received on a circuit which is configured for client mode (null IP address and port numbers), that packet is forwarded to the appropriate application. When that application replies to the packet, it sends the packet back to the originating circuit. The packet itself retains its IP and UDP headers (which means that the application must also retain the buffer in which it received the packet), and RNET uses the address+port in these headers, rather than the address+port in the circuit, for the peer IP address+port. The received packet’s source address+port becomes the reply packet’s destination address+port.

Server mode allows RNET-based applications to service requests from a large number of clients, even from clients which they have no previous configuration or knowledge of. Client mode enforces communication with a specific peer at a specific address+port.

In client mode, normally the IP address is null and the port number is non-null. Applications are normally bound to ports.

## RNET Buffers and Particle Chains

There are two types of packet buffering schemes which RNET supports: RNET buffers and Particles (*PCL*s for short). RNET’s entire feature set supports both models; the system engineer, however, can opt to enable or disable one or the other or both. At a minimum, of course, one of these two must be used. RNET can handle both kinds of buffers schemes, but will not convert an RNET buffer to a PCL chain and vice-versa.

The reason that there are two packet buffering schemes supported is that the two schemes are optimized for different ends. RNET packet buffers are optimized to save CPU cycles at the expense of wasting RAM; PCLs are optimized to economize on RAM at the expense of CPU cycles. Since RNET supports both models, it’s possible for a system to employ both schemes, having some applications and drivers use one model while other applications and drivers use the other model.

Both RNET buffers and PCLs rely on NUFR Service Layer (SL) APIs. RNET buffers are implemented by the use of a SL memory pool dedicated to RNET buffers. This RNET memory pool is configured in the RNET application layer (*rnet-app.c/h*). Particle support is built into the SL. There is only one particle pool, so the configuration for the particle pool must be done in the SL application space (*nsvc-app.c/h*), rather than in the RNET application space.

Any driver (including IRQ handlers, howbeit with CPU cycle hits), any application, and RNET itself have the ability to allocate and to free both RNET buffers and particle chains. In fact, there’s no restriction on having application components using allocate packet buffers for non-networking purposes. Naturally, with any memory pool, a memory leak will cause system-wide catastrophes, and any shared resource must be managed accordingly, but those are usage guidelines and not restrictions. Since memory on smaller embedded computing platforms is a scarce resource, the sharing of such pools is desirable for optimal memory utilization.

# The Packet Buffering Model

RNET conforms to a packetized programming model, rather than to a streaming programming model. Unix and Linux use a streaming model: data is streamed into sockets. In a packetized model, each packet is contained in a single packet buffer, and the packet buffer gets passed around. RNET forces this model on all drivers and on all applications which interface to it. NUFR has built-in support for packet buffers, and NUFR’s packet buffer support extends to drivers and IRQ handlers in drivers. Furthermore, NUFR’s messaging subsystem supports the passing of packet buffers attached to messages.

RNET makes use of these packet buffers in a couple of special ways. First, each packet buffer has a meta-data header. This header contains data about the packet. The header is embedded at the beginning of the packet buffer, so it gets passed around with the packet. If one has access to the packet, then he/she has access to the meta-data header also.

The meta-data header has the means to make a packet buffer which is *windowed.* Windowing is the use of a contiguous range of bytes within a packet buffer rather than the entire space available in the packet buffer. Windowing specifies and offset at which the packet begins and the length of the packet. This means that it is normal to have some (or several or many) bytes of unused space in the packet buffer. This unused space is variable, and will exist before or after the packet itself. This model provides flexibility between the consumers of packets: drivers, RNET itself, and applications. More on that later.

In addition to the windowing parameters, the meta-data header also specifies the interface, subinterface, and circuit which the packet arrived on or will be sent out on. Together with the windowing capability, the component which injects a packet into RNET must fill in the appropriate header parameters before injecting it; otherwise, RNET will drop the packet. In addition, there are windowing constraints which applications must adhere to. An application must populate its data at a safe starting offset within the packet. It cannot populate the packet without allotting enough space for RNET to pre-pend the L4 (UDP or ICMP), L3 (IPv4 or IPv6) and L2 (PPP and AHDLC) packet headers. This is a CPU optimization constraint.

## Using Packet Windows

We’ll use an RNET Buffer (type *rnet\_buf\_t* defined in *rnet-buf.h*) instead of a Particle Chain in this illustration of how packet windowing works. An RNET Buffer consists of the following:

* A meta-data header (an *rnet\_buf\_header\_t* which is the member named *header* in *rnet\_buf\_t*)
* The byte storage array (member *buf* of length *RNET\_BUF\_SIZE* in *rnet\_buf\_t*)

Let’s assume *RNET\_BUF\_SIZE* is defined to be 500. This means that the storage offset ranges from 0 to 499 inclusive. An actual packet will seldom consume all 500 bytes of storage. Some bytes in the RNET Buffer won’t be used. The unused bytes will be either before the start of the packet or after the end of the packet, typically both. The portion of the RNET Buffer which the packet occupies forms a “window” within the entire RNET Buffer.

To illustrate this with an example, given the *RNET\_BUF\_SIZE* of 500 bytes specified above, if there is a 200 byte packet payload (no UDP, IP headers, etc.) within the buffer, it might be placed as follows:

|  |  |  |
| --- | --- | --- |
|  | ***buf* offset** | **Purpose** |
|  | 0 | Start of *buf.* This area starts first unused area |
|  | 52 | End of first unused area |
| **Packet “Window” Here** | 53 | Start of 200 byte packet payload |
| 252 | End of 200 byte packet payload |
|  | 253 | Start of second unused area |
|  | 499 | Last byte in *buf*; end of second unused area. |

The meta-data header has two members: *offset* and *length.* The offset specifies the start of the window and the length specifies the length of the window, which in this case is the length of the packet payload. So the offset and length values for this example would be 53 and 200 respectively.

The packet payload was formed by an application, and has therefore not been sent out the RNET stack yet. The packet window starts at offset 53 and ends at offset 252 inclusive. The unused area spanning offsets 0 to 52 will be used by RNET to pre-pend a UDP+IPv6+PPP header (in this case, we assume that the packet is UDP+IPv6+PPP). RNET requires that the application which prepares a packet to be sent out the stack window the data so that there are enough unused bytes to pre-pend the header without running out of space. An exception for this rule is the AHDLC escape character translation (the 0x7D 2-byte escape sequence for escaping 0x7D and 0x7E). This translation must by necessity shift the data to accommodate the inclusion of the escape sequences, the quantity of which varies from packet to packet.

In addition to leaving room to prep-pend the L2/L3/L4 networking headers, the windowing of the packet within the packet buffer must allow room to append a 2-byte CRC16 for the L2 protocols which require this. As a rule, the packet windowing must be set so that RNET doesn’t have to shift any bytes around when it builds its packet headers, if the packet is sent from the application out the RNET stack. The exception to this rule, as mentioned above, is the AHDLC control character translation.

It is the responsibility of the application engineer to initialize the packet buffer’s starting offset to an acceptable value when populating a packet with payload data, for transmission out the RNET stack. It is assumed that the data will be written at this starting offset, and that the data won’t overrun the end of an RNET Buffer. RNET Buffers have a hard limit on the amount of data which can be written to them. Particle chains, on the other hand, can be extended. The limit to a PCL chain is the number of PCLs in the PCL pool. This is an advantage that PCLs have over RNET Buffers.

## **Preparing and Transmitting Packets in Host-Mode, Overview**

An unsolicited packet which an application sends out of a physical interface, out of the device, by using RNET, destined for a peer host is a host-mode transmit (tx). The steps an application must take to accomplish this are as follows:

* The application must allocate a packet buffer
* The packet buffer’s window must be established: the buffer offset must be calculated, and the header offset value is set to this offset.
* The buffer must be populated with payload data
* The header length field is set to the number of bytes that were written into the payload data. The buffer window is now set.
* The application must know the circuit to send this packet to. If the circuit is unknown, it can be found by making an RNET call to look it up.
* The header circuit index is filled in according to the circuit this packet will be sent out to
* The application builds up a message and attaches the packet buffer to this message
* The application sends this message to the RNET stack, using the appropriate RNET message id (in this case, UDP tx)

RNET, having received this packet, does the following:

* RNET extracts the circuit index from the meta-data header. RNET retrieves the stored information concerning the circuit.
* The subinterface number is stored in the packet buffer’s header
* The UDP source and destination port numbers are filled in
* The UDP header is pre-pended to the packet
* The packet header’s offset is decremented by 8, and the header length is increased by 8, to reflect the addition of the UDP header.
* RNET looks up the parent subinterface for the circuit index on which the packet was set. From the subinterface, RNET determines whether the packet is IPv4 or IPv6 and sends it to the appropriate L3 message id destination for further processsing (RNET sends itself a message with the packet).
* Assuming this was an IPv6 packet, the RNET IPv6 tx handler processes the packet; if it were an IPv4 packet, the IPv4 tx handler processes it
* RNET uses the circuit and subinterface header values to look up the subinterface and circuit again. From these it fetches the source and destination IP addresses
* With the source and destination IP addresses in hand, RNET pre-pends the IP header to the packet
* The packet header’s offset and length are modified to reflect the addition of this new data
* Using the subinterface value, the RNET IP processor looks up the interface for that subinterface
* The interface indicates which L2 protocol that the packet will need to be processed by. In this case, it is PPP. RNET sends the packet to the PPP component for Tx processing.
* The PPP component builds a PPP header, updates the window values (offset and length), then sends this to the AHDLC component.
* The AHDLC component encodes this for AHDLC
* RNET is finished preparing the packet and will send it to a driver for Tx out of the physical interface it must be sent out. RNET, using the interface settings, looks up the message settings for Tx on this interface.
* RNET attaches the packet to the message and sends it to the next device which will receive it. In this case, we’ll assume it is a UART driver.

Steps take by a hypothetical UART driver to send the packet:

* The UART driver receives the packet from RNET by means of a message send. This infers that there is some task context which manages packet Tx’s on behalf of the UART. This may be the same task as the task which hosts the RNET stack—it’s up to the application developer’s discretion.
* The task message handler which receives the packet from RNET will invoke the driver to send this packet out the interface
* There are a variety of means which the driver might take takes transmit this packet. It is likely that this will involve interaction with an interrupt handler, at some point.
* The packet data is copied out of the packet buffer (perhaps a few bytes at a time) and sent to the driver for Tx.
* The packet buffer, no longer needed, must be freed by the UART driver or by some other task. In either case, the packet must be freed or there will be a packet buffer memory leak. RNET is not responsible for returning the packet buffer back to the memory pool.

## Reading Data from an RNET Data Buffer

A component such as the RNET stack, an application, or a drive, will write data into an RNET Data Buffer in a windowed fashion. When data is read, the caller has the option to shorten the window, so the next read will continue where the previous one left off. The example code shows how to read that data and shorten the window:

#include “rnet-buf.h”

// Copies up to 10 bytes or length of frame, whichever is

// shorter. Shortens window for bytes read.

// Returns bytes read.

unsigned read\_up\_to\_10\_bytes(rnet\_buf\_t \*buf,

uint8\_t \*read\_output)

{

rnet\_buf\_t \*buf;

uint8\_t \*ptr;

ptr = RNET\_BUF\_START\_FRAME\_PTR(buf);

if (buf->header.length < 10)

{

copy\_length = buf->header.length;

}

else

{

copy\_length = 10;

}

// equivalent to memcpy()

rutils\_memcpy(read\_output, ptr, copy\_length);

// shorten window, excluding bytes just read

buf->header.offset += copy\_length;

buf->header.length -= copy\_length;

return copy\_length;

}

## Reading Data from a Particle

The equivalent function as the above, but applied to a particle, is as follows:

#include “nsvc-api.h”

unsigned read\_up\_to\_10\_bytes(nsvc\_pcl\_t \*head\_pcl,

uint8\_t \*read\_output)

{

nsvc\_pcl\_header\_t \*pcl\_header;

nufr\_sema\_get\_rtn\_t alloc\_rv;

nsvc\_pcl\_chain\_seek\_t read\_posit;

unsigned copy\_length;

unsigned bytes\_read;

bool rv;

pcl\_header = NSVC\_PCL\_HEADER(head\_pcl);

rv = nsvc\_pcl\_set\_seek\_to\_headerless\_offset(

head\_pcl,

read\_posit,

pcl\_header->offset);

if (!rv)

{

return 0;

}

if (pcl\_header->total\_used\_length < 10)

{

copy\_length = buf->header.length;

}

else

{

copy\_length = 10;

}

bytes\_read = nsvc\_pcl\_read(&read\_posit,

read\_output,

copy\_length);

// shorten window, excluding bytes just read

pcl\_header->offset += bytes\_read;

pcl\_header->total\_used\_length -= bytes\_read;

return bytes\_read;

}

## Setting RNET Buffers in Client Mode

Here’s an example of an application allocating, adding payload data to, then sending a buffer to RNET, for transmission to a peer host. For brevity’s sake, the circuit index is not explained here; it’s assumed that the index is passed by the caller. Circuit indices will be explained later.

#include “rnet-ahdlc.h”

#include “rnet-ppp.h”

#include “rnet-ip.h”

#include “rnet-dispatch.h”

#define OFFSET\_FOR\_UDP\_IPV6\_PPP\_TX \

(AHDLC\_FLAG\_CHAR\_SIZE + PPP\_PREFIX\_LENGTH + \

IPV6\_HEADER\_SIZE + UDP\_HEADER\_SIZE)

uint8\_t payload1[] = {0, 1, 2};

uint8\_t payload2[] = {3, 4, 5};

// This builds an app packet and copies in the fixed

// 6 byte payload contained in ‘payload1’ and ‘payload2’

// Packet is sent to RNET, which will send it out stack.

//

// ‘circuit\_index’ is calculated by caller

//

void my\_packet\_accessor\_for\_bufs(unsigned circuit\_index)

{

rnet\_buf\_t \*buf;

uint8\_t \*ptr;

unsigned payload\_length;

payload\_length = sizeof(payload1) + sizeof(payload2);

// Get an RNET Buffer. Blocks indefinitely until buffer

// is available.

buf = rnet\_alloc\_bufW();

// Set the meta-data header fields

buf->header.offset = OFFSET\_FOR\_UDP\_IPV6\_PPP\_TX;

buf->header.length = payload\_length;

buf->header.circuit = circuit\_index;

ptr = RNET\_BUF\_START\_FRAME\_PTR(buf);

rutils\_memcpy(ptr, payload1, sizeof(payload1));

ptr += sizeof(payload1);

rutils\_memcpy(ptr, payload2, sizeof(payload2));

// Inject packet into RNET:

// This is entry point for UDP sends of RNET Buffers

rnet\_msg\_send(RNET\_ID\_TX\_BUF\_UDP, buf);

}

## Setting Particle Chains in Client Mode

The same example as before, except PCLs are used instead of RNET buffers:

#include “rnet-ahdlc.h”

#include “rnet-ppp.h”

#include “rnet-ip.h”

#include “rnet-dispatch.h”

#include “nsvc-api.h”

#include “nufr-api.h”

#define OFFSET\_FOR\_UDP\_IPV6\_PPP\_TX \

(AHDLC\_FLAG\_CHAR\_SIZE + PPP\_PREFIX\_LENGTH + \

IPV6\_HEADER\_SIZE + UDP\_HEADER\_SIZE)

#define PACKET\_TRAILER\_SIZE \

(AHDLC\_FLAG\_CHAR\_SIZE + RUTILS\_CRC16\_SIZE)

uint8\_t payload1[] = {0, 1, 2};

uint8\_t payload2[] = {3, 4, 5};

// This builds an app packet and copies in the fixed

// 6 byte payload contained in ‘payload1’ and ‘payload2’

// Packet is sent to RNET, which will send it out stack.

//

// ‘circuit\_index’ is calculated by caller

//

void my\_packet\_accessor\_for\_pcls(unsigned circuit\_index)

{

nsvc\_pcl\_t \*head\_pcl;

nsvc\_pcl\_header\_t \*pcl\_header;

nufr\_sema\_get\_rtn\_t alloc\_rv;

nsvc\_pcl\_chain\_seek\_t write\_posit;

nufr\_sema\_get\_rtn\_t wr\_rc;

unsigned payload\_length;

unsigned start\_offset;

payload\_length = sizeof(payload1) + sizeof(payload2);

start\_offset = NSVC\_PCL\_OFFSET\_PAST\_HEADER(

OFFSET\_FOR\_UDP\_IPV6\_PPP\_TX);

// Commented out code is expected size of total packet.

// You can try to allocate the entire chain in

// one API call, instead of finding out later that,

// maybe, that the pool is depleted.

// Uncommented out code is most convenient method:

// Let the APIs lengthen the chain as needed.

// minimum\_length = start\_offset +

// payload\_length +

// PACKET\_TRAILER\_SIZE;

minimum\_length = 1;

alloc\_rv = nsvc\_pcl\_alloc\_chainWT(&head\_pcl,

NULL,

payload\_length,

NSVC\_PCL\_NO\_TIMEOUT);

if (NUFR\_SEMA\_GET\_OK\_NO\_BLOCK != alloc\_rv)

{

return; // error

}

pcl\_header = NSVC\_PCL\_HEADER(head\_pcl);

pcl\_header->offset = start\_offset;

pcl\_header->total\_used\_length = payload\_length;

pcl\_header->circuit = circuit\_index;

// Initialize seek struct to point to start of

// frame window (offset)

rv = nsvc\_pcl\_set\_seek\_to\_headerless\_offset(head\_pcl,

&write\_posit,

start\_offset);

if (!rv)

{

return; // error

}

// If chain were pre-allocated for correct size

// (used commented-out code for minimum\_length

// instead of a “1”), we could’ve used the

// simpler API ‘nsvc\_pcl\_write\_data\_continue’

wr\_rc = nsvc\_pcl\_write\_dataWT(&head\_pcl,

&write\_posit,

payload1,

sizeof(payload1),

NSVC\_PCL\_NO\_TIMEOUT);

if (NUFR\_SEMA\_GET\_OK\_NO\_BLOCK != wr\_rc)

{

// Have to clean up existing chain, or will

// cause a memory leak

nsvc\_pcl\_free\_chain(head\_pcl);

return;

}

// ‘write\_posit’ automatically advanced after

// previous write.

wr\_rc = nsvc\_pcl\_write\_dataWT(&head\_pcl,

&write\_posit,

payload2,

sizeof(payload2),

NSVC\_PCL\_NO\_TIMEOUT);

if (NUFR\_SEMA\_GET\_OK\_NO\_BLOCK != wr\_rc)

{

nsvc\_pcl\_free\_chain(head\_pcl);

return;

}

// Inject packet into RNET:

// This is entry point for UDP sends of PCLs

rnet\_msg\_send(RNET\_ID\_TX\_PCL\_UDP, head\_pcl);

}

## Servicing Packets in Server Mode

An application acts as a server when it receives packets from a host—a peer device, either known or unknown—processes the packet then sends a response back to it. In order to accomplish this, the client’s IP address (source IP) and its source UDP port must be swapped in the response packet: the destination IP address and destination port number are taken from the request packet’s source IP address and source port. RNET accomplishes this by having the server application re-use the originating packet. The packet still has in it the IP and UDP headers; it simply has been windowed down to present the payload to the application. All that is necessary is for the server application to change the payload data, adjust the length for the payload data change, and send the packet back to RNET for transmission. The circuit index in the packet header is unchanged; the circuit index will, pointing to a server-type circuit, will inform RNET to swap the source and destination IP address and UDP Port number.

Here’s an example:

#include “rnet-buf.h”

// Pass in a buffer received from a circuit configured

// for server mode. This fcn. will process it, then

// send it back out.

void my\_server(rnet\_buf\_t \*buf)

{

rnet\_buf\_t \*buf;

uint8\_t \*ptr;

uint8\_t single\_byte;

ptr = RNET\_BUF\_START\_FRAME\_PTR(buf);

// Read 1 byte from the packet, increment it

// by 1, write 2 bytes back to payload

single\_byte = \*ptr;

\*ptr++ = single\_byte + 1;

\*ptr = 2;

// Reset length. This excludes any extra

// bytes in rx packet’s payload, if necessary.

buf->header.length = 2;

// We don’t have to set the circuit—it was set

// by RNET in the rx processing. Same one will be

// be used on tx

rnet\_msg\_send(RNET\_ID\_TX\_BUF\_UDP, buf);

}

## Writing to Packet Buffers in Drivers

The above examples should show one how to use RNET Buffers and Particles well enough to figure out how a driver should use it. There are some things that are different. At a high level, they are:

* On an Rx packet, the driver must must set the packet header’s *intfc* (interface) field to indicate which interface the packet came in on
* Rx packets don’t have any data pre-pended to them, unlike Tx packets sent from an application. They don’t have to reserve bytes for the packet headers, as the Rx packets will have headers.

As part of the packet processing, RNET will find what subinterface and circuit (if UDP) that the packet matches and will fill in the meta data accordingly.

## Finding Circuit Indices

Packets which are sent by applications in client mode must have the meta-data circuit index value set, or else RNET will not know where to send the packet. In order to find the circuit index that the packet needs to have set, one must first know the IP address of self and peer hosts and the UDP port number of the peer where to the packet will be sent. Naturally, a circuit must exist for this destination peer, otherwise, RNET will drop the packet.

RNET provides an API for looking up the circuit index from the parameters described:

#include “rnet-intfc.h”

#include “rnet-ip.h”

// Assume Addresses are IPv6.

// Addresses passed in as strings. This is a convenience.

// Must know interface also.

int lookup\_destination(rnet\_intfc\_t intfc,

const char\* src\_ip\_addr,

const char\* dest\_ip\_addr,

uint16\_t udp\_dest\_port\_num)

{

rnet\_ip\_addr\_union\_t src\_ip\_addr\_binary;

rnet\_ip\_addr\_union\_t dest\_ip\_addr\_binary;

int rc;

bool rv;

uint8\_t subi\_value;

// Only have to do this once

rc = rnet\_ipv6\_ascii\_to\_binary(&src\_ip\_addr\_binary,

src\_ip\_addr,

true);

rc &= rnet\_ipv6\_ascii\_to\_binary(&dest\_ip\_addr\_binary,

dest\_ip\_addr,

true);

if (rc < 0)

{

return rc;

}

// Find subinterface first

rv = rnet\_subi\_lookup(intfc,

&src\_ip\_addr\_binary,

true,

&subi\_value);

if (!rv)

{

return RFAIL\_NOT\_FOUND;

}

index =

rnet\_circuit\_index\_lookup((rnet\_subi\_t)subi\_value,

RNET\_IP\_PROTOCOL\_UDP,

0, // wildcard match

udp\_dest\_port\_num,

&dest\_ip\_addr\_binary);

return index;

}

## **RNET’s Packet Flow Model**

## RNET Tasking Environment

Unlike other networking stacks, RNET itself does not constitute a task. The application or system engineer is responsible for creating a task wrapper under which RNET will run, or for integrating RNET into an existing task. The reason why RNET does not have its own task environment is to widen the flexibility of its usage in a real platform and to save RAM on RAM-scarce systems.

The below code shows RNET integrated into a task of its own which does nothing but run RNET:

>>>>>> start nsvc-app.h >>>>>>>

...

typedef enum

{

NSVC\_MSG\_PREFIX\_local = 0,

NSVC\_MSG\_PREFIX\_RNET = 50, // arbitrary value

} nsvc\_msg\_prefix\_t;

...

>>>>>> end nsvc-app.h >>>>>>>>

>>>>>> start my-task-file.c >>>>>>

#include “raging-global.h”

#include “rnet-dispatch.h”

#include “nsvc-api.h”

#include “nsvc-app.h”

#include “nufr-api.h”

// Assume ‘nufr\_tid\_t’ for this task is ‘NUFR\_TID\_MY\_TASK’

my\_task\_entry\_point(unsigned parm)

{

nufr\_msg\_t \*msg;

uint16\_t id;

uint32\_t parameter;

nsvc\_msg\_prefix\_t prefix\_to\_match =

NSVC\_MSG\_PREFIX\_RNET;

nsvc\_msg\_prefix\_t prefix;

// This initializes RNET

rnet\_set\_msg\_prefix(prefix\_to\_match, NUFR\_TID\_MY\_TASK);

rnet\_intfc\_init();

while (true)

{

nsvc\_msg\_get\_argsW(NULL,

&prefix,

&id,

NULL,

NULL,

&parameter);

if (prefix\_to\_match == prefix)

{

rnet\_msg\_processor(id, parameter);

}

}

}

>>>>>> end my-task-file.c >>>>>>

There are a few extra steps in creating this task, such as adding an enum to *nufr\_tid\_t*, adding a stack and adding an entry in *nufr-platform-app.c*, etc. Refer to the NUFR Manual for instructions for creating a task.

There will be no example here of how to integrate RNET into an existing task. This exercise is left to the reader.

But the essentials of how RNET runs under-the-cover is exposed in the example shown above. RNET sends and receives packets which are attached to NUFR message blocks. RNET itself sends itself messages, as it parses a packet up and down the stack (from L2 to L3 to L4 protocol and the reverse). Message with packets attached are sent from drivers, from applications, from tasks, from IRQ handlers—from any source—and injected into the stack. The packet can be injected on the receive/decode path, from device to application, or the reverse: injected on the transmit/encode path, from application to device.

## Packet Injection Messages

## These are the messages used for sending packets to RNET

|  |  |  |
| --- | --- | --- |
| *RNET Message ID* |  | *Desc.* |
| *RNET\_ID\_RX\_BUF\_ENTRY* |  | Driver insertion of RNET Buffer into RNET stack |
| *RNET\_ID\_RX\_PCL\_ENTRY* |  | Driver insertion of PCL into RNET stack |
| *RNET\_ID\_TX\_BUF\_UDP* |  | App insertion of RNET Buffer to be transmitted out device |
| *RNET\_ID\_TX\_PCL\_UDP* |  | App insertion of PCL to be transmitted out device |

## These will be used as parameters to rnet\_msg\_send(). **Configuration**

RNET has mandatory static initializations. *Static* means hard-coded configuration settings. The static configuration is handled in the files *rnet-app.c/h.* Much of the mechanics adding configuration items to the enums and structs presented there will follow the pattern set down by NUFR, so that is not re-explained here.

## Configuration Elements

Here is a table showing the various configuration elements. These are mostly contained in the RNET application component (*rnet-app.c/h*).

|  |  |  |
| --- | --- | --- |
| *Identifier* | *Type* | *Description* |
|  |  |  |
| RNET\_BUF\_SIZE | #define | The max size of a single RNET buffer |
| RNET\_NUM\_BUFS | #define | The number of buffers in the RNET buffer pool |
| NSVC\_PCL\_SIZE | #define | The max size of a single SL particle (in nsvc-app.h) |
| NSVC\_PCL\_NUM\_PCLS | #define | The number of particles in the SL particle pool (in nsvc-app.h) |
| RNET\_NUM\_CIR | #define | The maximum number of RNET circuits |
|  |  |  |
|  |  |  |
|  |  |  |
| rnet\_intfc\_t | enum | Specifies all RNET interfaces |
| rnet\_subi\_t | enum | Specifies all RNET subinterfaces |
| rnet\_persist\_cir\_t | enum | Specifies a static circuit configuration block, for those circuits which desire static configuration parameters/want to be created at init time |
|  |  |  |
| RNET\_EVENT\_LIST\_SIZE\_INIT\_COMPLETE | #define | Number of callbacks in init complete event list |
| RNET\_EVENT\_LIST\_SIZE\_INTFC\_UP | #define | Number of callbacks in interface up event list |
| RNET\_EVENT\_LIST\_SIZE\_INTFC\_DOWN | #define | Number of callbacks in interface down event list |
| rnet\_event\_list\_init\_complete | struct | Init complete event list |
| rnet\_event\_list\_intfc\_up | struct | Interface up event list |
| rnet\_event\_list\_intfc\_down | struct | Interface down event list |
|  |  |  |

## Configuring NUFR

Configuration of the NUFR compile switches:

|  |  |  |
| --- | --- | --- |
| *NUFR Compile Switch* | *Value* |  |
| NUFR\_CS\_LOCAL\_STRUCT | (don’t care) |  |
| NUFR\_CS\_MESSAGING | 1 |  |
| NUFR\_CS\_MSG\_PRIORITIES | 1 or greater |  |
| NUFR\_CS\_TASK\_KILL | (don’t care) |  |
| NUFR\_CS\_SEMAPHORE | 1 |  |
| CONTRACT\_ENFORCEMENT\_LEVEL | (don’t care) |  |

All portions of the Services Layer (SL) are used except for mutexes. To review, the initialization of the SL uses these function calls:

nsvc\_init();

nsvc\_msg\_bpool\_init();

nsvc\_pcl\_init();

nsvc\_timer\_init();

## The RNET File Structure

The RNET files are set up similar to the NUFR kernel and NUFR SL files. There is a single set of RNET files which exist in the ./*source*/ and ./*includes*/. All RNET files (files matching *rnet-\**) should be included in the build. Like the NFUR kernel and SL, there exists a set of RNET application files, the files *rnet-app.c/h*  and the file *rnet-compile-switches.h.* Like the other app files, there will likely be multiple copies of the RNET app files. In general, the RNET app files should be placed in the same directory as the other app files (*nufr-platform.c/h*, *nufr-platform-app.c/h, nsvc-app.c/h*, etc.).

## Configuring Interfaces and Subinterfaces

There must be 1 to N interfaces configured in order to use RNET. There must be at least one subinterface attached to an interface, in order to use that interface.

### The ROM Struct Interface Fields

A variable, which is an array, called *rnet\_static\_intfc* of type *rnet\_intfc\_rom\_t* must be created in *rnet-app.c.* The members of *rnet\_intfc\_rom\_t,* and notes on how to set them, are:

|  |  |
| --- | --- |
| *rnet\_intfc\_rom\_t member* | *Description* |
| l2\_type | The L2 protocol this interface runs |
| subi1, subi2, subi3 | The subinterfaces attached to this interface. 1 to 3 can/must be configured. If only 1 is configured, use subi1, etc. To specify “no subi”, use RNET\_SUBI\_null. |
| timer | Attachment point for a SL app timer. There must be one timer per interface. The timer cannot be shared between interfaces, etc. |
| counters | Attachment point for a set of interface timers. The timer struct time matches the interface L2 protocol type, as the timers vary according to L2 protocol. |
| counters\_size | Size of *counters* |
| tx\_packet\_api | Function pointer to call a driver API, when transmitting a packet. Packet is passed to driver in this call. This is final output stage of RNET. |
| option\_flags | Bit field of options, configured per interface. |

The *option\_flags* member uses these options:

|  |  |
| --- | --- |
| *Option* | *Description* |
| RNET\_IOPT\_RX\_AHDLC\_PRE\_TRANSLATED | For AHDLC, Driver (IRQ handler) does 0x7D character translations, instead of having RNET stack do it. |
| RNET\_IOPT\_RX\_AHDLC\_PRE\_CRC\_VERIFIED | For AHDLC, Driver (IRQ handler) does CRC16 verification, instead of having RNET stack do it. |
| RNET\_IOPT\_OMIT\_TX\_AHDLC\_TRANSLATION | For AHDLC, RNET stack will omit 0x7d character translations: driver will do it. |
| RNET\_IOPT\_OMIT\_TX\_AHDLC\_CRC\_APPEND | For AHDLC, RNET stack will omit CRC16 appending: driver will do it. |
| RNET\_IOPT\_PPP\_IPCP | For PPP, run IPCP protocol (i.e., interface will run IPv4) |
| RNET\_IOPT\_PPP\_IPV6CP | For PPP, run IPV6CP protocol (i.e., interface will run IPv6) |

### The ROM Struct Subinterface Fields

Same as the previous section, except for the subinterface ROM struct members: the array variable *rnet\_static\_subi* of type *rnet\_subi\_rom\_t.* This describes member of *rnet\_subi\_rom\_t*.

|  |  |
| --- | --- |
| *rnet\_subi\_rom\_t member* | *Description* |
| type | Type of subinterface: IPv4 or IPv6. If it’s an IPv6, then type of IPv6: global or link-local |
| acquisition\_method | Method whereby the subinterface’s IP address was derived |
| parent | Parent interface |
| prefix length | Prefix length. For example, 192.168.1.15/24 has a prefix length of 24. |
| ip\_addr | Subinterface’s IP address. This the self-IP address (not peer). |

### Interface and Subinterface ROM Settings Example

Here’s an example of creating two interfaces called *RNET\_INFC\_LEFT* and *RNET\_INTFC\_RIGHT*, along with creating three subinterfaces, *RNET\_SUBI\_LEFT\_LINK\_LOCAL*, *RNET\_SUBI\_LEFT\_GLOBAL*, *RNET\_SUBI\_RIGHT\_IPV4*:

>>>>>>>>>>> start rnet-app.h

typedef enum

{

RNET\_INTFC\_null = 0,

RNET\_INTFC\_LEFT,

RNET\_INTFC\_RIGHT,

RNET\_INTFC\_max

} rnet\_infc\_t;

#define RNET\_NUM\_INTFC (RNET\_INTFC\_max - 1)

typedef enum

{

RNET\_SUBI\_null = 0,

RNET\_SUBI\_LEFT\_LINK\_LOCAL,

RNET\_SUBI\_LEFT\_GLOBAL,

RNET\_SUBI\_RIGHT\_IPV4,

RNET\_SUBI\_max

} rnet\_subi\_t;

#define RNET\_NUM\_SUBI (RNET\_SUBI\_max - 1)

>>>>>>>>>>> end rnet-app.h

Since there are two interfaces and three subinterfaces, the configuration in *rnet-app.c* must align with this by having two initializers for *rnet\_static\_intfc* and three initializers for *rnet\_static\_subi*. The ordering between the enums and the initializers must match.

The interface initializations do the following:

* Bind RAM structures to the interface: a single counter struct and a single timer struct
* Bind subinterfaces to the interface. A total of three subinterfaces can be bound to an interface. If any subinterface slot is not used, a null enum is put in it splace.
* Map a driver callback function to the interface, to be used on transmit.
* Allow for option bits to be set for that interface

The subinterface initializations do the following:

* Bind a subinterface to an interface
* Specify traffic type
* Specify IP address and prefix (non-subnet) bit length
* Specify a hard-coded IP address for that subinterface

>>>>>>>>>>> start rnet-app.c

rnet\_ppp\_counters\_t rnet\_counters\_left;  
rnet\_ppp\_counters\_t rnet\_counters\_right;  
nsvc\_timer\_t rnet\_timer\_left;  
nsvc\_timer\_t rnet\_timer\_right;

// Two initializers for two interfaces

const rnet\_intfc\_rom\_t rnet\_static\_intfc[RNET\_NUM\_INTFC] = {  
 // RNET\_INTFC\_LEFT  
 {RNET\_L2\_PPP, // L2 type

RNET\_SUBI\_LEFT\_LINK\_LOCAL, // subi #1

RNET\_SUBI\_LEFT\_GLOBAL, // subi #2

RNET\_SUBI\_null, // subi #3  
 &rnet\_timer\_left,

&rnet\_counters\_left,

sizeof(rnet\_counters\_left),  
 my\_driver\_left, // driver callback  
 RNET\_IOPT\_PPP\_IPV6CP}, // interface options

// RNET\_INTFC\_RIGHT  
 {RNET\_L2\_PPP,

RNET\_SUBI\_RIGHT\_IPV4,

RNET\_SUBI\_null,

RNET\_SUBI\_null,  
 &rnet\_timer\_right,

&rnet\_counters\_right,

sizeof(rnet\_counters\_right),  
 my\_driver\_right,

RNET\_IOPT\_PPP\_IPCP},

};

const rnet\_subi\_rom\_t rnet\_static\_subi[RNET\_NUM\_SUBI] = {

// RNET\_SUBI\_LEFT\_LINK\_LOCAL  
 {RNET\_TR\_IPV6\_LINK\_LOCAL, // traffic type

RNET\_IPACQ\_HARD\_CODED, // method of assignment

RNET\_INTFC\_LEFT, // which intfc belongs to

64, // IP address prefix length

"FE80::2"}, // IP address

// RNET\_SUBI\_LEFT\_GLOBAL  
 {RNET\_TR\_IPV6\_GLOBAL,

RNET\_IPACQ\_HARD\_CODED,

RNET\_INTFC\_LEFT,

64,

"8001::2"},

// RNET\_SUBI\_RIGHT\_IPV4  
 {RNET\_TR\_IPV4\_UNICAST,

RNET\_IPACQ\_HARD\_CODED,

RNET\_INTFC\_RIGHT,

16,

"192.168.1.2"},

};

>>>>>>>>>>> end rnet-app.c

The

>>>>>>>>>>> start my-driver.c

void my\_driver\_left(rnet\_intfc\_t intfc,

void \*packet,

bool is\_pcl)

{

rnet\_buf\_t \*buf;

nsvc\_pcl\_t \*head\_pcl;

if (!is\_pcl)

{

buf = packet;

// transmit packet

rnet\_free\_buf(buf);

}

else

{

head\_pcl = packet;

// transmit packet

nsvc\_pcl\_free\_chain(head\_pcl);

}

}

// Repeat code from left callback fcn.

void my\_driver\_right(rnet\_intfc\_t intfc,

void \*packet,

bool is\_pcl)

{

...

}

>>>>>>>>>>> end my-driver.c

## Messaging Configuration Items

There are various configuration slots where a NUFR message field value is placed. For example, each callback in an event callback list has a message field value. This is a single *uint32\_t* value which specifies all the message fields for the message to be built up for sending upon triggering the event. Since these message field types are compressed message information, the application engineer must understand what macros to use to build these fields.

There is a dummy value called RNET\_LISTENER\_MSG\_DISABLED. This indicates a null message, one which won’t be sent. Apart from that, here are some examples of setting these fields

### Dummy/No-Send-Message Value

const rnet\_notif\_list\_t rnet\_event\_list\_init\_complete[

RNET\_EVENT\_LIST\_SIZE\_INIT\_COMPLETE] =  
{  
 {RNET\_LISTENER\_MSG\_DISABLED, NUFR\_TID\_null},  
};

### Example

This example configuration specifies that a single message will be sent whenever RNET initialization is done (completes). This single message will be sent to a hypothetical task called *NUFR\_TID\_MYTASK.* The message will be sent at a priority of *NUFR\_MSG\_PRI\_MID,* and will have a message prefix of 10 and a message ID of 2 (recommended practice is for the application developer to use his/her defined enums in place of the hard-coded numbers. Hard-coded numbers used only for the illustration sake.)

>>>>> in rnet-app.h >>>>>

#define RNET\_EVENT\_LIST\_SIZE\_INIT\_COMPLETE 1

>>>>>> in rnet-app.c >>>>

#include “nufr-api.h”

const rnet\_notif\_list\_t rnet\_event\_list\_init\_complete[

RNET\_EVENT\_LIST\_SIZE\_INIT\_COMPLETE] =  
{  
 {NUFR\_SET\_MSG\_FIELDS(

10, // message prefix

2, // message ID

NUFR\_TID\_null, // ‘nufr\_tid\_t’ sending task.

// Filled in by RNET

NUFR\_MSG\_PRI\_MID), // ‘nufr\_msg\_pri\_t’ select

// (message priority)

NUFR\_TID\_MYTASK}, // ‘nufr\_tid\_t’ type

// msg destination task.  
};

### Expanded Multi-message Example

The previous example can be expanded to send two message to two different tasks, instead of just the one. The second message is sent to task *NUFR\_TID\_OTHERTASK,* and has its own message prefixes and IDs (20 and 15):

const rnet\_notif\_list\_t rnet\_event\_list\_init\_complete[

RNET\_EVENT\_LIST\_SIZE\_INIT\_COMPLETE] =  
{  
 {NUFR\_SET\_MSG\_FIELDS(

10, // message prefix

2, // message ID

NUFR\_TID\_null, // ‘nufr\_tid\_t’ sending task.

// Filled in by RNET

NUFR\_MSG\_PRI\_MID), // ‘nufr\_msg\_pri\_t’ select

// (message priority)

NUFR\_TID\_MYTASK}, // ‘nufr\_tid\_t’ type

// msg destination task.

{NUFR\_SET\_MSG\_FIELDS(

20, // message prefix

15, // message ID

NUFR\_TID\_null, // ‘nufr\_tid\_t’ sending task.

// Filled in by RNET

NUFR\_MSG\_PRI\_MID), // ‘nufr\_msg\_pri\_t’ select

// (message priority)

NUFR\_TID\_OTHERTASK}, // ‘nufr\_tid\_t’ type

// msg destination task.  
};

## Circuit Configuration

This describes the ROM struct members for circuits, which is the global array *rnet\_static\_cir* in *rnet-app.c.* These are the members of the struct *rnet\_cir\_rom\_t.*

|  |  |
| --- | --- |
| *rnet\_cir\_rom\_t member* | *Description* |
| type | IP traffic type |
| protocol | UDP or TCP |
| self\_port | UDP/TCP port (self/us) |
| peer\_port | UDP/TCP port (peer/them) |
| subi | Subinterface this circuit is bound to |
| peer\_ip\_addr | IP address of peer/them host |
| buf\_listener\_msg | When an RNET Packet Buffer is received by RNET, this indicates message to send, to pass packet to application. Note this is formatted as a *nufr\_msg\_t* ‘msg->fields’ word, and therefore requires that one use the NUFR message packing macro *NUFR\_SET\_MSG\_FIELDS().* Set to *RNET\_LISTENER\_MSG\_DISABLED* if packet should be dropped/no app to send packet to. |
| pcl\_listener\_msg | Same as above, except for PCL chains |
| listener task | Task ID (*nufr\_tid\_t*) to send Rx packet message to. *NUFR\_TID\_null* means send message to same task as RNET is running under (self-task). |

# RNET APIs

These are the RNET API calls that are provided for applications or drivers to do RNET operations:

|  |  |  |
| --- | --- | --- |
| *API* | *.h file* | *Purpose* |
| rnet\_intfc\_init() | rnet-intfc.h | Initialization of the RNET stack. Call this before running RNET |
| rnet\_set\_msg\_prefix() | rnet-dispatch.h | Initializes RNET messaging |
| rnet\_create\_buf\_pool() | rnet-dispatch. | Initializes RNET Buffers |
| rnet\_subi\_lookup() | rnet-intfc.h | Find a subinterface |
| rnet\_circuit\_index\_lookup() | rnet-intfc.h | Find a circuit index |
| rnet\_circuit\_add() | rnet-intfc.h | Add a circuit |
| rnet\_circuit\_delete() | rnet-intfc.h | Delete a circuit |
| rnet\_msg\_send() | rnet-dispatch.h | Inject a packet into RNET |
| rnet\_alloc\_bufW() | rnet-dispatch.h | Allocate an RNET buffer, block until one is obtained. Can be called from task level or IRQ level. |
| rnet\_alloc\_bufT() | rnet-dispatch.h | Same as above, but with a timeout. |
| rnet\_alloc\_pclW() | rnet-dispatch.h | Allocate a particle chain, block until complete. Can be called from task or IRQ level. |
| rnet\_alloc\_pclT() | rnet-dispatch.h | Same as above, but with timeout. |
| rnet\_free\_buf() | rnet-dispatch.h | Free an RNET buffer |

# Running PPP

Some notes for an interface configured with PPP as the L2.

* The RNET PPP implementation is very simple: it does the minimum to bring up PPP. As many PPP options are ignored as possible.
* Be wary of exceeding MTU sizes on the device side when running PPP. The PPP MTU value is not sent on negotiations.
* The MTU is determined by the define RNET\_BUF\_SIZE, if packets are pushed up the stack from a driver that uses RNET Buffers.
* If particle chains are used, the MTU is flexible, as particle chains by design can/will grow as data is appended. Therefore, the chain size is limited by the size of the pool, and not the size of any single PCL.

## Running PPP on Linux

Install *pppd*:

#sudo apt-get install pppd

To run in IPv6 mode only (no IPv4) at a baud rate of 115200:

# sudo pppd -detach lcp-echo-interval 0 debug noauth nopcomp noaccomp nocrtscts noip ipv6 ::1,::2 /dev/ttyUSB0 115200

This will create the network interface ppp0

To ping an IPv6 RNET device from Linux:

# ping6 FE80::2 –I ppp0

To ping with packets spaced out at 200 millisec intervals:

# ping6 FE80::2 –I ppp0 –i .2

To send 100 ping packets quietly:

# ping6 FE80::2 –I ppp0 –q –c 100

# Notes on Interface Drivers

RNET itself cannot send and receive packets over a serial interface—or any interface—without a driver. RNET itself does not include any drivers, at least not officially. Drivers vary from platform to platform, making it difficult to satisfy anyone with a single driver implementation. But since drivers can be challenging to write, especially for resource-constrained systems, some notes are presented here.

First, before writing any driver, the application or system engineer must decide what the goals are for the driver design. Is the driver intended to save RAM? Is it intended to save CPU cycles? What kind of performance do we expect out of it? Must it be portable, be flexible, or be inserted into some kind of driver class. What are the trouble-shooting strategies?

These goals will become the basis for the driver’s design. Whatever design is chosen, there are a few approaches that are commonly used when implementing a serial driver for a PPP-mounted interface:

#1 Use a circular buffer for the Rx channel