Using RMR

RMR intends to provide a simple message sending and receiving interface that insulates the user application from the underlying mechanics of the message/packet transport system, and provides for the selection of message destination (endpoint) based on message type or combination of message type and subscription ID. Highlights of using RMR are described in the paragraphs below, however for the most up to date, and detailed, description of the RMR interface, the developer is encouraged to install the development package and read the manual pages (e.g. man rmr, or man rmr_send_msg).

RMR Function Overview

The following table lists all of the external RMR functions and a brief description of each.

Function	Description
rmr_alloc_msg	Allocates a new message buffer that can be populated by the user application and passed to rmr_send_msg(), rm_call() for transmission.
rmr_bytes2meid	Accepts a fixed size buffer of bytes and copies those to the managed equipment ID field in the message header.
rmr_bytes2payload	Accepts a pointer which refers to the start of n bytes which are copied to the payload portion of a message (mostly for wrappers which are not able to write based on a C pointer).
rmr_bytes2xact	Accepts a pointer which refers to the start of n bytes which are copied to the transaction ID portion of the the message header.
rmr_call	Accepts a pointer to an RMr message buffer and causes the associated message to be transmitted to an endpoint based on subscription ID and/or message type, then waits for a message to be received which matches the transaction ID set in the outgoing message.
rmr_free_msg	Is used to release the storage associated with a message, including returning any memory associated with the message which was allocated by the underlying transport system.
rmr_get_rcvfd	Returns a file descriptor from the underlying transport system. This file descriptor is valid only for polling (e.g. with epoll) and cannot be directly read or written to. Some underlying transport mechanisms do not support direct polling, and if one of those is in use a -1 is returned. The epoll flag EPOLLIN should be used. NNG does not seem to support the EPOLLET (edge trigger) capabilities.
rmr_get_src	Copies the bytes in source field (hostname:port) from the header of a message to a buffer provided by the caller.
rmr_get_trace	Copies the bytes in the trace field to a buffer provided by the caller.
mr_get_rtlen	Returns the number of bytes (length) of the trace data contained in the message.
rmr_init	Initializes the RMr environment. In order to use the mt_call() function (multi-threaded call support) the RMRFL_MTCALL flag must be added to the flags passed to this function. The multi-threaded mode is supported ONLY when using NNG.
rmr_mt_call	Send a message and wait for the associated response in the same manner as rmr_call() except that this function allows for multiple threads to invoke concurrently with the guarenteed delivery of the correct response to the thread. Requires that multi-threaded receive processing be set up with a special flag on the rmr_init() call. NOT available when running on top of the Nanomsg environment; attempting to link a programme which uses this function with the nanomsg version of RMr will fail.
rmr_mt_rcv	This function will block until a message is received. It is specifically desinged to work when multi-threaded receive mode is enabled.
rmr_payload_size	Returns the number of bytes in the payload which are available to the user program. The user program may write this many bytes using the payload pointer as a reference to the start of the payload. This is not always the same as the message (data) length which indicates the number of bytes which the sender placed into the payload; length could be less than this size.
rmr_rcv_msg	This function will block until a message is received. The message buffer referencing the newly arrived message is returned. If multi-threaded receive mode is enabled, this funciton will act as a front end to the rmr_mt_rcv() call.
rmr_ready	This function checks the state of the routing table, and if a valid routing table has been received, or loaded from disk, it will return true. If a valid routing table is not loaded, then false is returned. A valid routing table is required for applications which wish to send messages; applications which are only receivers, do not need to wait on a routing table to function correctly.
rmr_rts_msg	Returns a message to the endpoint which sent it. Before sending the caller may adjust the message (new/altered payload, message type, subscription id), but should not change the transaction ID unless it is absolutely known that the sender did not use rmr_call() to send the message.
rmr_send_msg	Accepts a message and will transmit it to an endpoint based on the subscription id and/or message type.
rmr_set_stimeout	Allows the user application to configure the number of retry loops that RMr will execute when the underlying transport mechanism reports a transient send failure (e.g. EAGAIN). By default (if this funciton is not invoked) RMr will run one retry loop which consists of approximately 1000 attempts to send the message before reporting a failue (< 1ms on most systems). The user application may use this function to disable send retries within RMr (setting the retry value to 0). (The function name is a hold over from the original implementaiton on Nanomsg which implemented this as a Nanomsg controlled timeout.)

rmr_set_trace	Copies user supplied trace bytes into the trace field in the message header. If rmr_init_trace() was not used, or the size of the bytes being copied is different than was supplied to rmr_init_trace(), the message header will be adjusted to accommodate the bytes supplied. This will cause the message, and payload, to be copied which will add to the overall latency of the application.
rmr_init_trace	the message header will be adjusted to accommodate the bytes supplied. This will cause the message, and payload, to be copied which will add to the overall latency of the application.
rmr_str2meid	Allows the managed equipment ID (meid) to be supplied as a nil terminated ASCII string. The bytes of the string, including the final nil terminator, will be copied to the meid field of the message header. If the string is longer than the field size it will be truncated; the calling program is responsible for checking state and taking appropriate action in this case. Also note that the function which retrieves bytes from this field always returns the full field size; if a string was used to set the field, bytes between the nil terminator and end of field will not be set, but will be returned.
rmr_str2xact	Allows the transaction ID field to be set from a nil terminated string. The same truncation and fetch caveats apply as described in rmr_str2meid().
rmr_tralloc_msg	Allocates a message and copies the supplied trace data to the message. See rmr_alloc_msg(). rmr_tralloc_msg
rmr_wh_close	Closes a wormhole connection.
rmr_wh_open	Opens a wormhole connection. Wormhole connections allow messages to be sent directly to a known endpoint (host:port or IP:port); message type and session ID are not used to determine the endpoint.
rmr_wh_send_msg	Sends a message to an open wormhole connection.

Sending Messages

By default, the RMr send functions all use a non-blocking method when attempting to send a message (see the non-retry send description below). This is done to prevent the underlying transport mechanism from blocking for longer than is tolerable by the user application. Because RMr's sending functions, with the execption of rmr_call() are synchronous, if a message cannot be successfully handed over to the transport mechanism the message is returned to the calling application with the state in the message buffer set to something other than RMR_OK. There are cases where the inability to pass the message to the transport mechanism is a transient, or soft, failure, and the user application may wish to retry the send operation. RMr signals these transient error states by setting the status in the message buffer to RMR_ERR_RETRY; all other states indicate a hard failure which will not ever be successful if retried. If the user application does not check the message status after a send, and attempt a retry until the message is accepted, it will appear that messages are being lost as these will never reach their destination.

The following pseudo code is recommended, but some user applications may have different approaches based on other limitations/expectations and may actutally queue a retry until sometime in the future. This, or similar, methods should be coded directly in C/C++ user applications, or into wrappers if this behaviour is to be hidden from the higher level language programmer.

```
msg = create_new_message( buffer )
msg = rmr_send_msg( ctx, msg )
while(  msg != nil && msg.state == RMR_ERR_RETRY )
    msg = rmr_send_msg( ctx, msg )
```

The user application may wish to limit the number of retries, issue warnings, and certainly trap error conditions that are hard errors. For additional examples, please refer to the examples directory of the RMr source code.

Disabling RMR Retries

Because the underlying transport is configured in non-blocking mode, it is expected that the transport will not immediatly accept a message for sending, and will indicate to the caller (RMr) that the failure is "soft" and liklely would be successful if retried later. In an attempt to reduce the complexity of the user application RMr will default to retrying "soft failing" sends for a small period of time (actually measured by number of attempts) before reporting the failure to the user application. By default, RMr will execute one "retry loop" consisting of a maximum of 1000 attempts to send the message; the user application may disable this retry feature (via rmr_set_stimeout), or extend it as is needed/desired. To disable the retry feature, the following may be used:

```
mr_ctx = rmr_init( port, max_def_msize, NO_FLAGS );
if( mr_ctx != NULL ) {
   rmr_set_stimeout( mr_ctx, 0 ); // disable retries for any send operation
}
```

CAUTION: When reties are disabled, the user programme must expect that some messages will not be accepted by the underlying transport mechanism (NNG), and that the state of these send attempts will be reported as RMR_ERR_RETRY. These situations are completely under the control of the transport system, and are not generated or controllable by RMr. Even with retries are enabled, there are situations (e.g. during session reconnect) where the underlying transport will not accept a message for sending, even after many hundreds of retries. Again, these will be returned with the message state of RMR_ERR_RETRY, and the user application must be prepared to handle them accordingly.