# *ROME_IF*

The ROME_IF module is a shared library providing an API for the Standard message set and support for message-passing in a STREAMS environment.

## Data Definitions

*rome_if.h* contains the following type definitions:

| | |
|---|---|
| queue_t | The standard STREAMS structure representing a queue of mblks between streams modules. In the ROME implementation it contains pointers to the upstream process, *src*, the current process, *me*, and the downstream process' file handle, *dest*. The structure also contains a queue of oustanding read requests, *reads*, and a pointer to the local (per-process) context for the queue *q_ptr*. |
| ROME_HANDLERS | The structure used by the generic message dispatching routine *rome_generic_hand* to process messages. It associates a message operation code *opcode* with a handler *routine*. |
| ROME_MQUEUE | A generic message queue containing a *head* and *tail* pointer. |
| ROME_URL | The parsed form derived from a string URL in ROME format. The *scheme* is used to contain the destination process name. The optional local *port* usually selects a particular protocol within the destination. The *user* and *password* fields identify and authenticate the opened file at the destination. For networked connections the *host* field contains the string form of the destination hostname, while the *ipaddr* fields contains its 32-bit network address. The *ip_port* field contains the remote port number, and the *urlpath* field contains the remainder of the URL as an unparsed string. |

## Shared Library Macros and Routines

### rome_addhead

**(void)** *rome_addhead*(
  **ROME_MQUEUE** **_q*,
  **ROME_MESSAGE** **_m*)

The *rome_addhead* macro adds the message *_m* to the front of the queue *_q*. No interlocks are taken out by the macro, it is the caller's responsibility to use the macro within a critical section if the queue is a shared data structure.

## rome_addtail

**(void)** *rome_addtail*(
  **ROME_MQUEUE** **_q*,
  **ROME_MESSAGE** **_m*)

The *rome_addtail* macro adds the message *_m* to the end of the queue *_q*. No interlocks are taken out by the macro, it is the caller's responsibility to use the macro within a critical section if the queue is a shared data structure.

## rome_fetmblk

**void** *rome_fetmblk*(
  **FILE** **stream*,
  **ROME_MESSAGE** **msg*)

The *rome_fetmblk* routine formats the supplied message as a *FETMBLK* request and sends it to the destination process specified by the (open) FILE *stream*. The routine does not wait for the reply.

## rome_fopen

**FILE** **rome_fopen*(
  **ROME_URL** **fileurl*,
  **const char** **mode*,
  **int** *ix*)

The *rome_fopen* routine formats and sends an *OPEN* message using the parsed structure pointed to by *url* parameter. The destination process is taken from the *scheme* field. The *mode* parameter is used to set the mode flags in the *OPEN* message. The routine handles the *EAGAIN* return from filing system requests, indicating a symbolic link, by repeating the *OPEN* to the new destination. If the request eventually succeeds a FILE structure is allocated, and the file-table entry at index *ix* in the current process is initialised to contain this structure, which is also returned to the caller. If any of the the *OPEN* requests fail, *NULL* is returned.
The *rome_fopen* routine is not normally called directly from application code, either *fopen* or *rome_open_url* should be used, as they take care of maintaining the per-process filesystem tables. This routine can be used to open an explicit file index, for example to ensure that *stdin* is at index 0 in the file table.

## rome_generic_handler

**void** *rome_generic_handler*(
  **ROME_MESSAGE** **mptr*,
  **ROME_HANDLERS** **list*,
  **int** *listc*)

The *rome_generic_handler* routine provides a general-purpose message demultiplexing routine for most processes. It matches the operation code in the supplied message, *mptr*, with one of the codes in the list of handlers, *list*, and calls the routine associated with that code. *listc* is the number of entries in the list.

The routine uses *rome_kprintf* to print an error if the message cannot be handled, and attempts to return the message to the originator as a reply. To prevent infinite system loops, the routine does not attempt to return messages that are already marked with the ‘reply’ flag, discarding them instead. This may eventually cause the system to freeze if a process is waiting for that message.

## rome_get_event

**void** *rome_get_event*(
  **FILE** **stream*,
  **ROME_MESSAGE** **msg*,
  **int** *pri*)

The *rome_get_event* routine formats the supplied message as an *EVENT* request at ROME scheduling priority *pri* and sends it to the destination process specified by the (open) FILE *stream*. The routine does not wait for the reply.

## rome_get_local_context

**ptr** *rome_get_local_context*(
  **ROME_MESSAGE** **msg*)

The *rome_get_local_context* routine returns the *dest_context* value for the message *msg* as though the message had been sent to the current process. This is for use with messages that are ‘in transit’ using *rome_pass_upstream*.

## rome_getcwd

**char** **rome_getcwd*(**void**)

The *rome_getcwd* routine returns the current value of the ‘working directory’ process variable as set by the *rome_setcwd* routine.

## rome_getmblk

**void** *rome_getmblk*(
  **FILE** **stream*,
  **ROME_MESSAGE** **msg*)

The *rome_getmblk* routine formats the supplied message as a *GETMBLK* request and sends it to the destination process specified by the (open) FILE *stream*. The routine does not wait for the reply.

## rome_getroot

**char** **rome_getroot*(**void**)

The *rome_getroot* routine returns the current value of the ‘device root’ process variable as set by the *rome_setroot* routine.

## rome_make_full_path

**void** *rome_make_full_path*(
**char** **to*,
**const char** **from*)

The *rome_make_full_path* routine converts the supplied string in *from* into a full ROME-URL style string in *to*, by adding the current device and working directory for the process, if needed. If *from* contains as ‘:’ separator, it is assumed that the string is already a full URL, and is copied unchanged to the output.

## rome_make_url

**int** *rome_make_url*(
**char** **output*,
**ROME_URL** **url*,
**int** *maxl*)

The *rome_make_url* routine converts the data structure form of a URL, pointed to by the *url* parameter, to the string form in the *output* parameter, up to a maximum of $maxl - 1$ characters (leaving room for the NUL to terminate the string). The routine returns 0 if the URL was converted correctly and within the supplied length, and 1 otherwise.

## rome_newmblk

**void** *rome_newmblk*(
**FILE** **stream*,
**ROME_MESSAGE** **msg*,
**int** *max*)

The *rome_newmblk* routine formats the supplied message as a *NEWMBLK* request for *max* bytes and sends it to the destination process specified by the (open) FILE *stream*. The routine does not wait for the reply.

## rome_open_url

**FILE** **rome_open_url*(
**ROME_URL** **url*,
**const char** **mode*)

The *rome_open_ur*l routine locates a spare file entry in the process’ file table and calls *rome_fopen* to initialise the file structure with the supplied parameters.

## rome_outmblk

**void** *rome_outmblk*(
**FILE** **stream*,
**ROME_MESSAGE** **msg*)

The *rome_outmblk* routine formats the supplied message as an *OUTMBLK* request and sends it to the destination process specified by the (open) FILE *stream*. The routine does not wait for the reply.

## rome_parse_url

**int** *rome_parse_url(*
**const char** **input*,
**ROME_URL** **url*)

The *rome_parse_url* routine converts *input*, the string form of a URL, to the parsed structure into the *url* parameter. The routine returns 0 if the URL was parsed correctly, and 1 otherwise.

## rome_pass_downstream

**void** *rome_pass_downstream*(
**queue_t** **queue*,
**ROME_MESSAGE** **msg*)

The *rome_pass_downstream* routine is used in a module pushed into a sequence of modules to pass a message to the next downstream (ie. towards the driver) module. Note that when the reply is received it must be handled with *rome_pass_upstream*, in order to release the context information stored with the message.

## rome_pass_upstream

**void** *rome_pass_upstream*(
**ROME_MESSAGE** **msg*)

The *rome_pass_upstream* routine is used in a module pushed into a sequence of modules to pass a message to the next upstream (ie. towards the application) module. The message must have been previously handled in this process by an equivalent call to *rome_pass_downstream*.

## rome_putmblk

**void** *rome_putmblk*(
**FILE** **stream*,
**ROME_MESSAGE** **msg*)

The *rome_putmblk* routine formats the supplied message as a *PUTMBLK* request and sends it to the destination process specified by the (open) FILE *stream*. The routine does not wait for the reply.

## rome_remhead

**ROME_MESSAGE** **rome_remhead*(
**ROME_MQUEUE** **qp*)

The *rome_remhead* routine removes the first message from the head of the *qp* message queue and returns it as the result. If this is the last message on the queue, the head and tail pointers are both set to NULL. The routine return NULL if the queue is empty. No interlocks are taken out by the routine, it is the caller's responsibility to call it within a critical section if the queue is a shared data structure.

## rome_retmblk

**void** *rome_retmblk*(
**ROME_MESSAGE** **msg*)

The *rome_retmblk* routine formats the supplied message as a *RETMBLK* request and sends it back to the process which last sent the message. The routine waits for the reply (since this is usually handled within a queue handler).

## rome_send_command

**int** *rome_send_command*(
  **char** **proc*,
  **char** **command*)

The *rome_send_command* routine sends the string *command* as a *COMMAND* message to the process named in *proc* and waits for the reply. The routine returns either the response code to the message or ENOPROCESS if the process *proc* does not exist in the system.

## rome_sendwait

**int** *rome_sendwait*(
  **ROME_MESSAGE** **msg*,
  **ROME_PROCESS** **dest*)

The *rome_sendwait* routine sends the message *msg* to the process identified by *dest* and waits for the reply. It returns either the return code from the reply or EBADREPLY if the wrong reply is received.

## rome_setcwd

**void** *rome_setcwd*(
  **const char** **where*)

The *rome_setcwd* routine replaces the current value of the per-process ‘working directory’ variable with a copy of the string pointed to by the *where* variable.

## rome_setroot

**void** *rome_setroot*(
  **const char** **where*)

The *rome_setroot* routine replaces the current value of the per-process ‘device root’ variable with a copy of the string pointed to by the *where* variable.

## rome_setup_mblk

**void** *rome_setup_mblk(*
  **mblk_t** **n*,
  **uchar** **buff*,
  **int** *size*)

The *rome_setup_mblk* routine initialises the read, write and data pointers of the supplied mblk, *n*, to point to the start of the buffer *buff* with a buffer limit set at $buff + size$.