Standard MessageSet

The *Standard* messageset contains the basic dataflow messages and interface messages supported by the C runtime library (Clib) and the rome interface library (rome_if).

Message Definitions

ROME uses a STREAMS model for dataflows, where a single unit of data in the flow is represented by a linked list of **mblk_t** data structures. As this is the most frequent use of ROME messages (in terms of the number of messages processed) the **mblk_t** is the default parameter type of all messages, so there is no need to write an *RCAST* macro to access the parameters. This format is used by the *COMMAND, FETMBLK, GETMBLK, NEWMBLK, OUTMBLK, PUTMBLK* and *RETMBLK* messages below.

Unlike standard STREAMS, the **mblk_t** type is actually identical to the complete **ROME_MESSAGE** data type. However, the routines which process these messages should only access the fields specifically designated as mblk fields. The data definition appears as if the following structure were declared:

```
typedef struct
{
        mblk_t  *b_cont                 continuation_block
        uchar   *b_rptr                 current read pointer
        uchar   *b_wptr                 current write pointer
        uchar   *b_base                 original base of buffer
        uchar   *b_lim                  absolute end of buffer
        uint    *b_type                 buffer type
        uint    *immed                  immediate data
        uint    *immed1                 more immediate data
}mblk_t;
```

The data area of this buffer extends from *b_base* to *b_lim*. Within this area, new data should be written starting at *b_wptr* and existing data should be read from *b_rptr*. For those not familiar with the STREAMS dataflow model, the following explanation may serve; for further examples see the ROME Programmer's manual. An empty buffer is characterised by having *b_rptr == b_wptr*, not necessarily at the start of the buffer. A data producer places data in the buffer and increments *b_wptr.* The number of bytes of data in the buffer is *b_wptr - b_rptr.* In general, a data consumer reads the buffer in the order in which the data were written, starting at the byte pointed to by *b_rptr* and advancing the pointer until it meets *b_wptr*.

However, the mblk is more flexible than a simple producer/consumer model. It's main advantage lies in processing protocol stacks which encapsulate higher-level data with headers and/or trailers. An application (data producer), requests a buffer from the system. In the ROME dataflow implementation, the request is passed down through the protocol stack, allowing each layer to modify the size of the buffer being requested. The lowest layer (usually a device driver) creates an empty buffer, with *b_rptr = b_wptr = b_base* and *b_lim* set to (at least) *b_base + length*. As the buffer is returned towards the applications, protocol layers can reserve space at the head of the buffer by incrementing both *b_rptr* and *b_wptr*. This increment can be dynamically calculated for each request, allowing variable-sized headers to be generated. Finally the application receives an 'empty' buffer into which it can place its data, updating *b_wptr* as it goes and leaving *b_rptr* unchanged. As the filled buffer passes back towards the driver, the protocol layers can retrieve their reserved space at the head of the buffer by subtracting the requested amount from *b_rptr*. Trailers can be added directly at *b_wptr*. The result is that a single buffer can be processed at multiple layers without needing to copy data.

The same approach can be used for data passing upwards towards the application. By modifying *b_rptr*, sucessive protocol headers can be stripped off the data before reaching the application, which then sees only its own data between *b_rptr* and *b_wptr*.

The method of passing the buffer requests all the way to the driver allows a device driver to allocate buffer space according to its specific needs, usually this means in uncached memory, but some devices (for example on a VMEbus) must have their data in a completely separate memory region. This cannot guarantee a zero-data-copy architecture in the case of device–device transfers if the requirements of the input and output devices are different.

The *b_cont* field allows a single operation to span multiple buffers, each with its own read and write pointers. This approach is rarely used in ROME, as it requires scatter/gather DMA support in device drivers to avoid a data copy. One place where multiple buffers are used is in the Console application, to avoid copying lines of data from user-supplied buffers.

The *b_type* field allows the contents of the buffer to be further identified. Usually this value is set to *M_DATA* indicated uninterpreted data. The *b_type* field may also be used to identify the use to which the two *immed* fields are being put. For example setting the type to *M_IP* indicates that the first *immed* field contains a 32bit IPv4 address to which this datagram should be sent.

CLOSE

The *CLOSE* message is sent to terminate an open message path. It does not use the parameter area. At the time the reply to the *CLOSE* is generated there should be no more messages outstanding on this path (for example buffered reads or writes). No further messages are expected on this path, and any context-dependent data at the receiving process may be freed.

COMMAND

The *COMMAND* message contains a string between the *b_rptr* and *b_wptr* pointers of the supplied message block. It is expected that the receiving process will act upon the contents of the string in some way. The contents of the data buffer should not be modified, and the buffer should not be freed.

EVENT

```
typedef struct
{
        int          event_type                event code for this message
}ROME_T_EVENT;
```

The *EVENT* message is sent by a process to indicate a registered event to the destination. Mostly, these events are defined by separate event types in other messagesets, with their own parameter areas. This datatype is a placeholder type to allow first-level interpretation of incoming *EVENT* messages (i.e. to extract the event type which is always the first word of the parameter area).

FETMBLK

The *FETMBLK* message requests that the supplied mblk be filled with data. For this message, the caller must supply an initialised **mblk_t** with the buffer already allocated. The buffer will be filled, starting at *b_wptr* and not exceeding *b_lim*. There is no guarantee how much data will be returned (exept that it will not overflow the buffer), but processes should, in general, try to put as much data as possible into the buffer.

FLUSH

The *FLUSH* message requests that any pending output for the message path be sent to the destination before the *FLUSH* reply is generated. This message may be used to synchronise data transmissions within a system, or to ensure that data that would otherwise be buffered (for example partial output lines awaiting a newline character) are sent ‘asis’.

GETMBLK

The *GETMBLK* message requests a buffer of data from the destination. The caller supplies an uninitialised **mblk_t** structure which is filled in by the destination with the received data. There is no guarantee about the amount of data returned. The buffer is owned by the destination process and the caller must return it with a *RETMBLK* message once it has processed the contents.

NEWMBLK

The *NEWMBLK* messages requests that the fields of the supplied **mblk_t** data structure be initialised to point to a buffer supplied by the destination. The minimum size of this buffer is passed in the *b_wptr* field by the caller (in bytes). The buffer is owned by the destination process and the caller must return it with a *PUTMBLK* message once it has been filled with data.

OPEN

```
    typedef struct
    {
            ROME_URL   *openurl              parsed URL of open string
            uint       mode                  parsed mode bits
    }ROME_T_OPEN;
```

The *OPEN* message is used to create a new message path from the caller to the destination process. The *openurl* parameter contains a parsed structure which the destination process may use to identify internal destinations (for example a specific file within the filing system). The *mode* parameter is a bitfield of *FILE_READ_FLAG, FILE_WRITE_FLAG* and *FILE_BINARY_FLAG*, usually derived from the mode string passed to the *fopen* routine.

Depending upon the processing model of the destination, it may allocate context-specific information to represent this message path, which may be returned in the *dest_context* field of the message. As long as the caller uses the correct interface routines, this value will appear in the *dest_context* field of all subsequent messages on this path.

OUTMBLK

The *OUTMBLK* message requests that the data in the supplied **mblk_t** be transmitted by the destination process to an external device or a further downstream process. When transmission is complete (or an error indication is to be returned) the buffer is returned unchanged to the caller on the reply. The reply is not generated until the destination process has no further references to the buffer. The *OUTMBLK* message is for use when the caller or another process owns the data buffer, or when the caller may wish to re-transmit the buffer.

PUTMBLK

The *PUTMBLK* message requests that the data in the supplied **mblk_t** be transmitted by the destination process to an external device or a further downstream process. When transmission is complete (or an error indication is to be returned) the buffer is returned to the buffer pool of the (ultimate) destination process. The buffer must have previously been allocated by a *NEWMBLK* message on the same message path (i.e. the buffer must be sent to the process which allocated it). If the buffer is not owned by the destination, an *OUTMBLK* message should be used instead. Once the reply is generated, the caller should not access any of the fields in the **mblk_t**.

A *PUTMBLK* message with zero data length (*b_rptr == b_wptr*) is used to return a buffer to its allocating process without transmitting any data. This may be used to free a buffer previously transmitted with an *OUTMBLK* message (for example when the acknowledgement arrives), to free a buffer sent to a different process, or to return a buffer that the sender decided it did not need.

RETMBLK

The *RETMBLK* message returns the buffer in the supplied **mblk_t** to the free pool of the destination process. The buffer must have been previously obtained from that process by a *GETMBLK* message. Buffers obtained through *NEWMBLK* must be returned by a *PUTMBLK*.