STREAMS Utilities C

Introduction

This appendix specifies the set of utility routines provided by STREAMS to assist development of modules and drivers.

The general purpose of the utilities is to perform functions that are commonly used in modules and drivers. However, some utilities also provide the required interrupt environment.

Note – The utility routines contained in this appendix represent an interface that will be maintained in subsequent versions of SunOS 5.x. Other than these utilities, functions in the STREAMS kernel code may change between versions.

Structure definitions are contained in Appendix A, "STREAMS Data Structures". Routine references are found in this appendix. The following definitions are used:

Blocked

A queue that cannot be enabled due to flow control.

Enable

To schedule a queue's `service` procedure to run.

Free

To release a STREAMS message or other data structure.

Message block (bp)

A triplet consisting of an `msgb` structure, a `datab` structure, and a data buffer. It is referenced by its type definition `mblk_t`.

Message (mp)

One or more linked message blocks. A message is referenced by its first message block.

Message queue

Zero or more linked messages associated with a queue (`queue` structure).

Queue (q)

A `queue` structure. When it appears with "message" in certain utility description lines, it means "message queue".

Schedule

To place a queue on the internal linked list of queues which will subsequently have their `service` procedure called by the STREAMS scheduler.

The word module will generally mean "module and/or driver". The phrase "next/following module" generally refers to a module, driver, or Stream head.

Utility Descriptions

The STREAMS utility routines are described in the following section. A summary table is contained at the end of this appendix.

`adjmsg` – trim bytes in a message

```
int adjmsg(mblk_t *mp, int len);
```

`adjmsg()` trims bytes from either the head or tail of the message specified by *mp*. If *len* is greater than zero, it removes *len* bytes from the beginning of *mp*. If *len* is less than zero, it removes (-)*len* bytes from the end of *mp*. If *len* is zero, `adjmsg()` does nothing.

`adjmsg()` only trims bytes across message blocks of the same type. It fails if *mp* points to a message containing fewer than *len* bytes of similar type at the message position indicated.

adjmsg() returns 1 on success and 0 on failure.

allocb – allocate a message and data block

```
mblk_t *allocb(int size,unsigned int pri);
```

allocb() returns a pointer to a message block of type M_DATA, in which the data buffer contains at least *size* bytes. *pri* is one of BPRI_LO, BPRI_MED, or BPRI_HI and indicates how critically the module needs the buffer. *pri* is currently unused and is maintained only for compatibility with applications developed prior to SunOS 5.3. If a block can not be allocated as requested, allocb() returns a NULL pointer.

When a message is allocated via allocb() the *b_band* field of the mblk_t is initially set to zero. Modules and drivers may set this field if necessary.

backq – get pointer to the queue behind a given queue

```
queue_t *backq(queue_t *cq);
```

backq() returns a pointer to the queue behind a given queue. That is, it returns a pointer to the queue whose *q_next* (see queue structure in Appendix A, "STREAMS Data Structures") pointer is *q*. If no such queue exists (as when *q* is at a Stream end), backq() returns NULL.

bcanput – test for flow control in the given priority band

```
int bcanput(queue_t *q, unsigned char pri);
```

bcanput() provides modules and drivers with a way to test flow control in the given priority band. It returns 1 if a message of priority *pri* can be placed on the queue. It returns 0 if the priority band is flow controlled and sets the QWANTW flag for band zero (QB_WANTW for a nonzero band).

If no bands yet exist on the queue in question, 1 is returned. The call bcanput(q,0) is equivalent to the call canput(q).

bcanputnext- test for flow control in the given priority band

```
int bcanputnext(queue_t *q, unsigned char pri);
```

bcanputnext() provides modules and drivers with a way to test flow control in the given priority band for the queue pointed to by *q->q_next*. It returns 1 if the message of priority `pri` can be placed in the queue. It returns 0 if the priority band is flow controlled and sets the `QWANTW` flag for band zero (`QB_WANTW` for a nonzero band).

If the band does not yet exist on the queue in question, 1 is returned.

`bufcall` – *recover from failure of* `allocb`

```
int bufcall(unsigned int size,int pri,void (*func)(),long
arg, arg);
```

`bufcall()` is provided to assist in the event of a block allocation failure. If `allocb()` returns `NULL`, indicating a message block is not currently available, `bufcall()` may be invoked.

`bufcall()` arranges for (**func*)(*arg*) to be called when a buffer of *size* bytes is available. *pri* is as described in `allocb()`. When *func* is called, it has no user context and must return without sleeping. `bufcall()` does not guarantee that the desired buffer will be available when *func* is called since interrupt processing may acquire it.

`bufcall()` returns a non-zero id on success, indicating that the request has been successfully recorded, and 0 on failure. The returned id should be kept in the event `unbufcall()` needs to be called. On a failure return, *func* will never be called. A failure indicates a (temporary) inability to allocate required internal data structures.

`canput` – *test for room in a queue*

```
int canput(queue_t *q);
```

`canput()` determines if there is room left in a message queue. If *q* does not have a `service` procedure, `canput()` will search further in the same direction in the Stream until it finds a queue containing a `service` procedure (this is the first queue on which the passed message can actually be queued). If such a queue cannot be found, the search terminates on the queue at the end of the Stream. `canput()` tests the queue found by the search. If the message queue in this queue is not full, `canput()` returns 1. This return indicates that a message can be put to queue *q*. If the message queue is full, `canput()` returns 0. In this case, the caller is generally referred to as blocked.

`canput()` only takes into account normal data flow control.

`canputnext` - test for room in the next queue

```
int canputnext(queue_t *q);
```

`Canputnext()` determines if there is room left in a message queue, pointed to by *q->q_next*. If the queue does not have a service procedure, `canputnext()` will search further in the same direction in the stream until it finds a queue containing a service procedure (this is the first queue on which the passed message can actually be queued.) If such a queue cannot be found, the search terminates on the queue at the end of the stream. `canputnext()` tests the queue found by the search. If the message queue in this queue is not full, `canputnext()` returns 1. This return indicates that a message can be put to queue *q*. If the message queue is full, `canputnext()` returns 0. In this case, the caller is generally referred to as blocked. `Canputnext()` only takes into account normal data flow control.

`copyb` – copy a message block

```
mblk_t *copyb(mblk_t *bp);
```

`copyb()` copies the contents of the message block pointed at by *bp* into a newly-allocated message block of at least the same size. `copyb()` allocates a new block by calling `allocb()`. All data between the *b_rptr* and *b_wptr* pointers of a message block are copied to the new block, and these pointers in the new block are given the same offset values they had in the original message block.

On successful completion, `copyb()` returns a pointer to the new message block containing the copied data. Otherwise, it returns a `NULL` pointer.

`copymsg` – copy a message

```
mblk_t *copymsg(mblk_t *mp);
```

`copymsg()` uses `copyb()` to copy the message blocks contained in the message pointed at by *mp* to newly-allocated message blocks, and links the new message blocks to form the new message.

On successful completion, `copymsg()` returns a pointer to the new message. Otherwise, it returns a `NULL` pointer.

`datamsg` – test whether message is a data message

```
int datamsg(unsigned char type);
```

The `datamsg()` macro returns `TRUE` if `mp->b_datap->db_type` (where *mp* is declared as `mblk_t *mp`) is a data type message (that is, not a control message). In this case, a data type is `M_DATA`, `M_PROTO`, `M_PCPROTO`, or `M_DELAY`. If `mp->b_datap->db_type` is any other message type, `datamsg()` returns `FALSE`.

`dupb` – duplicate a message block descriptor

```
mblk_t *dupb(mblk_t *bp);
```

`dupb()` duplicates the message block descriptor (`mblk_t`) pointed at by *bp* by copying it into a newly allocated message block descriptor. A message block is formed with the new message block descriptor pointing to the same data block as the original descriptor. The reference count in the data block descriptor (`dblk_t`) is incremented. `dupb()` does not copy the data buffer, only the message block descriptor.

On successful completion, `dupb()` returns a pointer to the new message block. If `dupb()` cannot allocate a new message block descriptor, or if any reference count exceeds the limit (currently 255), it returns `NULL`.

This routine allows message blocks that exist on different queues to reference the same data block. In general, if the contents of a message block with a reference count greater than 1 are to be modified, `copymsg()` should be used to create a new message block and only the new message block should be modified. This insures that other references to the original message block are not invalidated by unwanted changes.

`dupmsg` – duplicate a message

```
mblk_t *dupmsg(mblk_t *mp);
```

`dupmsg()` calls `dupb()` to duplicate the message pointed at by *mp*, by copying all individual message block descriptors, and then linking the new message blocks to form the new message. `dupmsg()` does not copy data buffers, only message block descriptors.

On successful completion, `dupmsg()` returns a pointer to the new message. Otherwise, it returns `NULL`. Like `dupb()`, `dupmsg()` failures are due to either lack of resources or any reference count exceeding the limit (currently 255).

`enableok` – *re-allow a queue to be scheduled for service /*

```
void enableok(queue_t *q);
```

`enableok()` cancels the effect of an earlier `noenable()` on the same queue *q*. It allows a queue to be scheduled for service that had previously been excluded from queue service by a call to `noenable()`.

`esballoc` – *allocate message and data blocks*

```
mblk_t *esballoc(unsigned char *base, int size, int pri,
frtn_t *fr_rtnp);
```

`esballoc()` allocates message and data blocks that point directly to a client-supplied buffer. `esballoc()` sets *db_base*, *b_rptr*, and *b_wptr* fields to *base* and *db_lim* to *base* + *size*. The pointer to `struct free_rtn` is placed in the *db_freep* field of the data block.

The method by which `free_func` is called is implementation specific. Do not assume that free_func will or will not be called directly from STREAMS utility routines like `freeb`(). The `free_func` function must not call another modules `put` procedure nor attempt to acquire a private module lock which may be held by another thread across a call to a STREAMS utility routine which could free a message block. Otherwise, the possibility for lock recursion and/or deadlock exists.

If an error occurs, `esballoc()` returns `NULL`.

Note – Modules and drivers using `esballoc` have to be prepared to handle calls to the `esballoc`'s `callback free` function after the last instance of the module or driver has been closed. In order to prevent the module text from being unloaded while there are outstanding `esballoc` callbacks, modules using `esballoc` should have their `_fini`() routine return `EBUSY`.

`esbbcall` - *call function when buffer is available*

```
int esbbcall(int pri, void (*func)(long arg),long arg);
```

`esbbcall`(), like `bufcall`(9F), serves as a `timeout`(9F) call of indeterminate length. If `esballoc`(9F) is unable to allocate a message and data block header to go with its externally supplied data buffer, `esbbcall`() can be used to schedule the routine `func`, to be called with the argument `arg` when a buffer becomes available.

`flushband`–*flush the messages in a given priority band*

```
void flushband(queue_t *q, unsigned char pri, int flag);
```

`flushband()` provides modules and drivers with the capability to flush the messages associated in a given priority band. *flag* is defined the same as in `flushq()`. If *pri* is zero, only ordinary messages are flushed. Otherwise, messages are flushed from the band specified by *pri* according to the value of *flag*.

If a queue behind *q* is blocked, `flushband()` may enable the blocked queue, as described in "`getq - get a message from a queue`"

`flushq`–*flush a queue*

```
void flushq(queue_t *q, int flag);
```

`flushq()` removes messages from the message queue in queue *q* and frees them, using `freemsg()`. If *flag* is set to `FLUSHDATA`, `flushq()` discards all `M_DATA`, `M_PROTO`, `M_PCPROTO`, and `M_DELAY` messages, but leaves all other messages on the queue. If *flag* is set to `FLUSHALL`, all messages are removed from the message queue and freed. `FLUSHALL` and `FLUSHDATA` are defined in `<sys/stream.h>`.

If a queue behind *q* is blocked, `flushq()` may enable the blocked queue, as described in "`getq - get a message from a queue`".

`freeb`–*free a single message block*

```
void freeb(mblk_t *bp);
```

freeb() will free (deallocate) the message block descriptor pointed at by *bp*, and free the corresponding data block if the reference count (see "dupb - duplicate a message block descriptor") in the data block descriptor (datab structure) is equal to 1. If the reference count is greater than 1, freeb() will not free the data block, but will decrement the reference count.

If the reference count is 1 and if the message was allocated by esballoc(), the function specified by the db_frtnp->free_func pointer is called with the parameter specified by db_frtnp->free_arg. freeb(). freeb() can't be used to free a multi-block message (see freemsg()).

Note – Results will be unpredictable if the freeb() is called with a null argument.You should always check that pointer is non-NULL before using freeb().

freemsg – *free all message blocks in a message*

```
void freemsg(mblk_t *mp);
```

freemsg() uses freeb() to free all message blocks and their corresponding data blocks for the message pointed at by *mp*.

freezestr - *freeze a stream*

```
void freezestr(queue_t *q);
```

freezestr() freezes the state of the entire STREAM containing the queue *q*. A frozen STREAM blocks any thread attempting to enter any open, close, put or service routine belonging to any queue instance in the STREAM, and blocks any thread currently within the STREAM if it attempts to put messages onto or take messages off of any queue within the STREAM (with the sole exception of the caller). Threads blocked by this mechanism remain so until the STREAM is thawed by a call to unfreezestr().

getq – *get a message from a queue*

```
mblk_t *getq(queue_t *q);
```

`getq()` gets the next available message from the queue pointed at by *q*. `getq()` returns a pointer to the message and removes that message from the queue. If no message is queued, `getq()` returns `NULL`.

`getq()`, and certain other utility routines, affect flow control in the Stream as follows: If `getq()` returns `NULL`, the queue is marked with `QWANTR` so that the next time a message is placed on it, it will be scheduled for service (enabled, see `qenable()`). If the data in the queued messages in the queue drop below the low water mark, *q_lowat*, and a queue behind the current queue had previously attempted to place a message in the queue and failed (that is, was blocked, see `canput()`), then the queue behind the current queue is scheduled for service.

The queue count is maintained on a per-band basis. Priority band 0 (normal messages) uses *q_count*, *q_lowat*, etc. Nonzero priority bands use the fields in their respective `qband` structures (*qb_count*, *qb_lowat*, etc.). All messages appear on the same list, linked via their *b_next* pointers.

q_count does not reflect the size of all messages on the queue; it only reflects those messages in the normal band of flow.

`insq`–*put a message at a specific place in a queue*

```
int insq(queue_t *q, mblk_t *emp, mblk_t *nmp);
```

`insq()` places the message pointed at by *mp* in the message queue contained in the queue pointed at by *q* immediately before the already queued message pointed at by *emp*. If *emp* is `NULL`, the message is placed at the end of the queue. If *emp* is non-`NULL`, it must point to a message that exists on the queue *q*, or a system panic could result.

If an attempt is made to insert a message out of order in a queue via `insq()`, the message will not be inserted and the routine fails.

The queue class of the new message is ignored. However, the priority band of the new message must adhere to the following ordering:

```
emp->b_prev->b_band >= mp->b_band >= emp->b_band.
```

This routine returns 1 on success and 0 on failure.

The stream must be frozen by the caller when calling `insq()`.

`linkb` – *concatenate two messages into one*

```
void linkb(mblk_t *mp, mblk_t *bp);
```

`linkb()` puts the message block pointed at by *bp* at the tail of the message pointed at by *mp*.

`msgdsize` – *get the number of data bytes in a message*

```
int msgdsize(mblk_t *mp);
```

`msgdsize()` returns the number of bytes of data in the message pointed at by *mp*. Only bytes included in data blocks of type `M_DATA` are included in the total.

`msgpullup` – *concatenate bytes in a message*

mblk_t *msgpullup(mblk_t *mp, int len);

`msgpullup`() concatenates and aligns the first `len data` bytes of the message pointed to by `mp`, copying the data into a new message. Any remaining bytes in the remaining message blocks will be copied and linked onto the new message. The original message is unaltered.

`noenable` – *prevent a queue from being scheduled*

```
void noenable(queue_t *q);
```

`noenable()` prevents the queue *q* from being scheduled for service by `putq()` or `putbq()` when these routines queue an ordinary priority message, or by `insq()` when it queues any message. `noenable()` does not prevent the scheduling of queues when a high priority message is queued, unless it is queued by `insq()`.

`OTHERQ` – *get pointer to the mate queue*

```
queue_t *OTHERQ(queue_t *q);
```

The `OTHERQ()` function returns a pointer to the mate queue of *q*. If *q* is the read queue for the module, it returns a pointer to the module's write queue. If *q* is the write queue for the module, it returns a pointer to the read queue.

`pullupmsg` – *concatenate and align bytes in a message*

```
int pullupmsg(mblk_t *mp,int len);
```

`pullupmsg()` concatenates and aligns the first *len* data bytes of the passed message into a single, contiguous message block. Proper alignment is hardware dependent. `pullupmsg()` only concatenates across message blocks of similar type. It fails if *mp* points to a message of less than *len* bytes of similar type. If *len* is -1 `pullupmsg()` concatenates all the like-type blocks in the beginning of the message pointed at by *mp*.

On success, `pullupmsg()` returns 1 and, as a result of the concatenation, it may have altered the contents of the message pointed to by *mp*. On failure, it returns 0.

`put` – *call a STREAMS put procedure*

void put(queue_t *q, mblk_t *mp);

`put` calls the `put` procedure (`put`(9E) entry point) for the STREAMS queue specified by *q*, passing it the message block referred to by `mp`. It is typically used by a driver or module to call its own `put` procedure.

`putbq` – *return a message to the beginning of a queue*

```
int putbq(queue_t *q, mblk_t *bp);
```

`putbq()` puts the message pointed at by *mp* at the beginning of the queue pointed at by *q*, in a position in accordance with the message type. High priority messages are placed at the head of the queue, followed by priority band messages and ordinary messages. Ordinary messages are placed after all high priority and priority band messages, but before all other ordinary messages already in a queue. The queue will be scheduled in accordance with the same rules described in `putq()`. This utility is typically used to replace a message on a queue from which it was just removed.

A `service` procedure must never put a high-priority message back on its own queue, as this would result in an infinite loop.

`putbq()` returns 1 on success and 0 on failure.

`putctl` – *put a control message*

```
int putctl(queue_t *q, int type);
```

`putctl()` creates a control message of type *type*, and calls the `put` procedure of the queue pointed at by *q*, with a pointer to the created message as an argument. `putctl()` allocates new blocks by calling `allocb()`.

On successful completion, `putctl()` returns 1. It returns 0, if it cannot allocate a message block, or if *type* `M_DATA`, `M_PROTO`, `M_PCPROTO`, or `M_DELAY` was specified.

`putctl1` – *put a control message with a one-byte parameter*

```
int putctl1(queue_t *q, int type,int p);
```

`putctl1()` creates a control message of type *type* with a one-byte parameter *param*, and calls the `put` procedure of the queue pointed at by *q*, with a pointer to the created message as an argument. `putctl1()` allocates new blocks by calling `allocb()`.

On successful completion, `putctl1()` returns 1. It returns 0, if it cannot allocate a message block, or if *type* `M_DATA`, `M_PROTO`, or `M_PCPROTO` was specified. `M_DELAY` is allowed.

`putnext` – *put a message to the next queue*

```
int putnext(queue_t *q, mblk_t *mp);
```

`putnext`() calls the `put` procedure of the next queue in a Stream and passes it a message pointer as an argument. *q* is the calling queue (not the next queue) and *mp* is the message to be passed. `putnext()` is the typical means of passing messages to the next queue in a Stream.

`putnextctl` - *put a control message*

```
int putnextctl(queue_t *q, int type);
```

putnextctl() calls putctl(), passing the queue pointed at by *q->q_next*, and the message type *type*.

putnextctl1 - put a control message

```
int putnextctl1(queue_t *q, int type, int p);
```

putnextctl() calls putctl1(), passing the queue pointed at by *q->q_next,* the message type *type* and the one byte parameter *param.*

putq–put a message on a queue

```
int putq(queue_t *q, mblk_t *bp);
```

putq() puts the message pointed at by *mp* on the message queue contained in the queue pointed at by *q* and enables that queue. putq() queues messages based on message queueing priority.

The priority classes are high priority (type >= QPCTL), priority band (type < QPCTL && band >0), and normal (type < QPCTL && band == 0).

putq() always enables the queue when a high-priority message is queued. putq() is allowed to enable the queue (QNOENAB is not set), if the message is the priority band message, or the QWANTR flag is set indicating that the service procedure is ready to read the queue.

Note – The service procedure must never put a priority message back on its own queue, as this would result in an infinite loop.

putq() enables the queue when an ordinary message is queued if the following condition is set, and enabling is not inhibited by noenable(): the condition is set if the module has just been pushed, or if no message was found on the last getq() call, and no message has been queued since.

putq() looks only at the priority band in the first message block of a message. If a high priority message is passed to putq() with a nonzero *b_band* value, *b_band* is reset to 0 before placing the message on the queue. If the message is passed to putq() with *b_band* value that is greater than the number of qband structures associated with the queue, putq() tries to allocate a new qband structure for each band up to and including the band of the message.

`putq()` is intended to be used from the `put` procedure in the same queue in which the message will be queued. A module should not call `putq()` directly to pass messages to a neighboring module. `putq()` may be used as the *qi_putp*() `put` procedure value in either or both of a module's `qinit` structures. This effectively bypasses any `put` procedure processing and uses only the module's `service` procedure(s).

`putq()` returns 1 on success and 0 on failure.

`qenable` – *enable a queue*

```
void qenable(queue_t *q);
```

`qenable()` places the queue pointed at by *q* on the linked list of queues that are ready to be called by the STREAMS scheduler.

`qprocsoff` - *disable the put and service routines of a driver or module*

```
void qprocsoff(queue_t *q);
```

`qprocsoff`() disables the `put` and `service` routines of the STREAMS driver or module. When the routines are disabled in a module, messages flow around the module as if it were not present in the stream. `qprocsoff`() must be called by the `close` routine of an MT-(Multi-thread) safe driver or module before releasing any resources on which the driver/module's `put` and `service` routines depend. `qprocsoff`() will remove the queue's service routines from the list of service routines to be run, and waits until any concurrent `put` or `service` routines are finished.

`qprocson` - *enable the put and service routines of a driver or module*

```
void qprocson(queue_t *q);
```

`qprocson`() enables the `put` and `service` routines of the STREAMS driver or module. Prior to the call to `qprocson`(), the `put` and `service` routines of a newly pushed module or newly opened driver are disabled. For the module, messages flow around it as if it were not present in the stream. `qprocson`()

must be called by the first `open` of an MT-safe module or driver after allocation and initialization of any resource on which the `put` and `service` routines depend.

`qreply` – send a message on a Stream in the reverse direction

```
void qreply(queue_t *q, mblk_t *mp);
```

`qreply()` sends the message pointed at by *mp* up (or down) the Stream in the reverse direction from the queue pointed at by *q*. This is done by locating the partner of *q* (see `OTHERQ`()), and then calling the `put` procedure of that queue's neighbor (as in `putnext()`). `qreply()` is typically used to send back a response (`M_IOCACK` or `M_IOCNAK` message) to an `M_IOCTL` message.

`qsize` – find the number of messages on a queue

```
int qsize(queue_t *q);
```

`qsize()` returns the number of messages present in queue *q*. If there are no messages on the queue, `qsize()` returns 0.

`RD` – get pointer to the read queue

```
queue_t * RD(queue_t *q);
```

`RD()` accepts either a read queue or write queue pointer, *q*, as an argument and returns a pointer to the read queue for the same module.

`rmvb` – remove a message block from a message

```
mblk_t *rmvb(mblk_t *mp, mblk_t *bp);
```

`rmvb()` removes the message block pointed at by *bp* from the message pointed at by *mp*, and then restores the linkage of the message blocks remaining in the message. `rmvb()` does not free the removed message block. `rmvb()` returns a pointer to the head of the resulting message. If *bp* is not contained in *mp*, `rmvb()` returns a -1. If there are no message blocks in the resulting message, `rmvb()` returns a `NULL` pointer.

`rmvq` – remove a message from a queue

```
void rmvq(queue_t *q, mblk_t *mp);
```

`rmvq()` removes the message pointed at by *mp* from the message queue in the queue pointed at by *q*, and then restores the linkage of the messages remaining on the queue. If *mp* does not point to a message that is present on the queue *q*, a system panic could result.

The stream must be frozen by the caller when calling `rmvq`().

`strlog` – submit messages for logging

```
int strlog(short mid, short sid, char level, unsigned short
flags, char *fmt, unsigned arg1, ...);
```

`strlog()` submits messages containing specified information to the `log`(7) driver. Required definitions are contained in `<sys/strlog.h>` and `<sys/log.h>`. *mid* is the STREAMS module id number for the module or driver submitting the log message. *sid* is an internal sub-id number usually used to identify a particular minor device of a driver. *level* is a tracing level that allows selective screening of messages from the tracer. *flags* are any combination of:

- `SL_ERROR` (the message is for the error logger),
- `SL_TRACE` (the message is for the tracer),
- `SL_FATAL` (advisory notification of a fatal error),
- `SL_NOTIFY` (request that a copy of the message be mailed to the system administrator)

Note – `SL_NOTIFY` is not an option by itself, but rather a modifier to the `SL_ERROR` flag

- `SL_CONSOLE` (log the message to the console),
- `SL_WARN` (warning message), and
- `SL_NOTE` (notice the message).

fmt is a `printf`(3S) style format string, except that %s, %e, %E, %g, and %G conversion specifications are not handled. Up to `NLOGARGS` numeric or character arguments can be provided. (See `log`(7).)

`strqget` – obtain information about a queue or band of the queue

```
int strqget(queue_t *q, qfields_t what, unsigned char pri,
long *valp);
```

`strqget()` allows modules and drivers to get information about a queue or particular band of the queue. The information is returned in the *long* referenced by *valp*. The fields that can be obtained are defined by the following: `QHIWAT QLOWAT QMAXPSZ QMINPSZ QCOUNT QFIRST QLAST QFLAG QBAD`

`strqget()` returns 0 on success and an error number on failure.

The stream must be frozen by the caller when calling `strqget`().

`strqset` – change information about a queue or band of the queue

```
int strqset(queue_t *q, qfields_t what, unsigned char pri,
long val);
```

`strqset()` allows modules and drivers to change information about a queue or particular band (pri) of the queue. The updated information is provided by *val*. This routine returns 0 on success and an error number on failure. If the field is intended to be read-only, then the error EPERM is returned and the field is left unchanged.

See `<sys/stream.h>` for valid values of `what`.

The stream must be frozen by the caller when calling `strqset`().

`testb` – check for an available buffer

```
int testb(int size, unsigned int pri);
```

`testb()` checks for the availability of a message buffer of size *size* without actually retrieving the buffer. `testb()` returns 1 if the buffer is available, and 0 if no buffer is available. A successful return value from `testb()` does not guarantee that a subsequent `allocb()` call will succeed (for example, in the case of an interrupt routine taking buffers).

pri is as described in `allocb()`.

`unbufcall` – *cancel a* `bufcall` *request*

```
void unbufcall(int id);
```

`unbufcall()` cancels a `bufcall` request. *id* identifies an event in the `bufcall` request.

Caution – If a `bufcall()` request is made and the callback has not occurred before closing the driver, an `unbufcall()` must be made to cancel the scheduled callback. Otherwise a system crash may occur.

`unfreezestr` – *unfreeze a stream*

```
void unfreezestr(queue_t *q);
```

Unfreeze the entire STREAM pointed to by the queue *q*;

`unlinkb` – *remove a message block from the head of a message*

```
mblk_t *unlinkb(mblk_t *mp);
```

`unlinkb()` removes the first message block pointed at by *bp* and returns a pointer to the head of the resulting message. `unlinkb()` returns a `NULL` pointer if there are no more message blocks in the message.

`WR` – *get pointer to the write queue*

```
queue_t *WR(queue_t *q);
```

`WR()` accepts a read queue or write queue pointer, *q*, as an argument and returns a pointer to the write queue for the same module.

DKI Interface

With the DKI interface the following STREAMS utilities are implemented as functions: `datamsg`, `OTHERQ`, `putnext`, `RD`, `splstr`, and `WR`. `<sys/ddi.h>` must be included after `<sys/stream.h>` to get function definitions.

C

New MT perimeter utility routines

Note – The utility routines contained in this section represent facilities that were new to SunOS 5.3. These interfaces are subject to minor changes.

`qbufcall` – recover from failure of `allocb`

```
int qbufcall(queue_t *q, unsigned int size, int pri, void
(*func)(long arg), long arg);
```

`qbufcall()` is provided to assist in the event of a block allocation failure. If `allocb()` returns `NULL`, indicating a message block is not currently available, `qbufcall()` may be invoked.

`qbufcall()` arranges for (**func*)(*arg*) to be called when a buffer of *size* bytes is available. *pri* is as described in `allocb()`. The framework enters the perimeters associated with the queue *q* prior to calling *func.* Thus the only difference between `qbufcall()` and `bufcall()` is that the former schedules a callback that is synchronous whereas the latter schedules an asynchronous callback. When *func* is called, it has no user context and must return without sleeping.

`qbufcall()` does not guarantee that the desired buffer will be available when *func* is called since interrupt processing may acquire it.

`qbufcall()` returns a non-zero id on success, indicating that the request has been successfully recorded, and zero on failure. The returned id should be kept in the event that `qunbufcall()` needs to be called. On a failure return, *func* will never be called. A failure indicates a (temporary) inability to allocate required internal data structures.

`qtimeout` – execute a function after a specified length of time

```
int qtimeout(queue_t *q, void(*ftn)(), caddr_t arg, long
ticks);
```

`qtimeout()` arranges for (**func*)(*arg*) to be called after a specified time interval. The framework enters the perimeters associated with the queue *q* prior to calling *func.* Thus the only difference between `qtimeout()` and `timeout()` is that the former schedules a callback that is synchronous whereas the latter schedules an asynchronous callback. When *func* is called, it has no user context and must return without sleeping.

The exact time interval over which the timeout takes effect cannot be guaranteed, but the value is a close approximation.

`qtimeout()` returns a non-zero id on success, indicating that the request has been successfully recorded. Otherwise, if the `timeout()` table is full, the following panic message results: `PANIC: Timeout table overflow`.

The returned id should be kept in the event that `quntimeout()` needs to be calcanled.

`qunbufcall` – *cancel a* `qbufcall` *request*

```
void qunbufcall(queue_t *q, int id);
```

`qunbufcall()` cancels a `qbufcall` request. *q* is the queue that was passed to `qbufcall` and *id* is the identifier returned by `qbufcall`.

Caution – If a `qbufcall()` request is made and the callback has not occurred before closing the driver, a `qunbufcall()` must be made to cancel the scheduled callback. Otherwise a system crash may occur.

`quntimeout` – *cancel a* `qtimeout` *request*

```
int quntimeout(queue_t *q, int id);
```

`quntimeout()` cancels a `qtimeout` request. *q* is the queue that was passed to `qtimeout` and *id* is the identifier returned by `qtimeout`.

Caution – If a `qtimeout()` request is made and the callback has not occurred before closing the driver, a `quntimeout()` must be made to cancel the scheduled callback. Otherwise a system crash may occur.

`qwait/qwait_sig` – *STREAMS perimeter wait routines*

```
void qwait(queue_t *q);
```

```
int qwait_sig(queue_t *q);
```

`qwait()` and `qwait_sig()` are used to wait for a message to arrive to the `put` or `service` procedures. They can be used only in the `open` and `close` procedures in a STREAMS module. `qwait()` and `qwait_sig()` atomically exit the perimeters associated with the queue *q*, and wait for the next occurrence of a thread leaving the module's `put` or `service` procedures. Upon return `qwait()` and `qwait_sig()` re-enter the perimeters for the queue.

The difference between `qwait()` and `qwait_sig()` is that the latter can be interrupted by a signal. `qwait_sig()` normally returns non-zero, but when interrupted by a signal it returns zero.

`qwait()` and `qwait_sig()` serve the function, respectively, of `cv_wait()` and `cv_wait_sig()` for STREAMS modules that use perimeters.

`qwriter` - asynchronous STREAMS perimeter upgrade

```
void qwriter(queue_t *qp, mblk_t *mp, void(*func)(), int
perimeter);
```

`qwriter()` is used to upgrade the access at either the inner or the outer perimeter from shared to exclusive, when processing messages that require exclusive access in the `put` or `service` procedures.

`qwriter()` arranges for (**func*)(*q, mp*) to be called when exclusive access has been granted at the specified *perimeter*.

`qwriter()` will be upgraded to exclusive access immediately if possible, in which case *func* will be called before `qwriter()` returns. If it is not possible to upgrade without blocking, `qwriter()` will defer the upgrade until later and return before *func* has executed. Thus modules cannot assume that *func* has executed when `qwriter()` returns. One way to avoid dependencies on this is for the module to immediately return after calling `qwriter()` and let *func* perform all the remaining processing of the message *mp*.

When `qwriter()` defers executing *func*, the STREAMS framework will prevent other messages from entering the inner perimeter associated with the queue until the asynchronous upgrade has completed and *func* has finished executing.

C

Utility Routine Summary

Table C-1 Summary of Utility Routines

ROUTINE	DESCRIPTION
adjmsg	trim bytes in a message
allocb	allocate a message block
backq	get pointer to the queue behind a given queue
bcanput	test for flow control in a given priority band
bufcall	recover from failure of allocb
canput	test for room in a queue
copyb	copy a message block
copymsg	copy a message
datamsg	test whether message is a data message
dupb	duplicate a message block descriptor
dupmsg	duplicate a message
enableok	re-allow a queue to be scheduled for service
esballoc	allocate message and data blocks
flushband	flush messages in a given priority band
flushq	flush a queue
freeb	free a message block
freemsg	free all message blocks in a message
freezestr	disable changes to the state of the stream
getq	get a message from a queue
insq	put a message at a specific place in a queue
linkb	concatenate two messages into one
msgdsize	get the number of data bytes in a message
noenable	prevent a queue from being scheduled
OTHERQ	get pointer to the mate queue
pullupmsg	concatenate and align bytes in a message

Table C-1 Summary of Utility Routines

ROUTINE	DESCRIPTION
putbq	return a message to the beginning of a queue
putctl	put a control message
putctl1	put a control message with a one-byte parameter
putnext	put a message to the next `queue`
putq	put a message on a queue
qbufcall	call a function when a buffer becomes available
qprocsoff	turn off queue processing
qprocson	turn on queue processing
qreply	send a message on a Stream in the reverse direction
qsize	find the number of messages on a queue
qtimeout	execute a function after a specified length of time
qunbufcall	cancel a pending qbufcall request
quntimeout	cancel a pending qtimeout request
qwait	perimeter wait routine
qwait_sig	perimeter wait routine
qwriter	asynchronous perimeter upgrade
RD	get pointer to the read queue
rmvb	remove a message block from a message
rmvq	remove a message from a queue
splstr	set processor level
strlog	submit messages for logging
strqget	obtain information on a queue or a band of the queue
strqset	change information on a queue or a band of the queue
testb	check for an available buffer
unbufcall	cancel `bufcall` request

Table C-1 Summary of Utility Routines

ROUTINE	DESCRIPTION
unfreezestr	enable changes to the state of the stream
unlinkb	remove a message block from the head of a message
WR	get pointer to the write queue