Technote 1117

Open Transport STREAMS FAQ

CONTENTS

This Technote contains collected lore on writing STREAMS modules and drivers for use with Open Transport.

It is structured as a series of question and answer pairs, that answer Frequently Asked Questions about Open Transport STREAMS. However, this isn't just a collection of Q&As; a lot of the material is tutorial in nature.

This Technote is directed at developers who are writing OT kernel level plug-ins, such as protocol stacks, networking device drivers, and filtering and encryption software.

Getting Started

Q What is STREAMS?

A When written in upper case, **STREAMS** refers to a standard environment for loadable networking modules. This environment was first introduced as part of AT&T UNIX [UNIX is a registered trademark of UNIX Systems Laboratory, Inc., in the U.S. and other countries], but has since been ported to many platforms.

Q So what is Open Transport?

A **Open Transport** is an implementation of STREAMS on the Mac OS. OT contains a number of enhancements vis-a-vis a traditional STREAMS environment, but STREAMS lives at its core.

Q What is Mentat Portable Streams?

A **Mentat Portable Streams** (MPS) is a fast, portable implementation of STREAMS that is licensed to system vendors by Mentat. While MPS is compliant with the AT&T UNIX STREAMS at the API level, it contains many enhancements, both internal and external. Open Transport's STREAMS environment is based on MPS.

Q I'm just getting started with STREAMS. What should I read?

A There are a number of useful references that explain the STREAMS architecture in general:

- *Programmer's Guide: STREAMS, UNIX System V Release 4* , UNIX Press, ISBN 0-13-02-0660-1
- *STREAMS Modules and Drivers, UNIX System V Release 4.2* , UNIX Press, ISBN 0-13-066879-6
- *UNIX System V Network Programming* , Stephen Rago, Addison-Wesley, ISBN 0-20-156318-5

The "Open Transport Module Developer Note" (part of the OT Module SDK) describes the differences between a standard UNIX STREAMS implementation and the one provided by Open Transport. In general, the OT implementation is very close to UNIX, so if you're an experienced UNIX STREAMS programmer you will be in familiar territory.

Open Transport Advanced Client Programming explains many of the low-level client programming interfaces required to test and plumb your STREAMS plug-ins under Open Transport.

Another reference I find useful is UNIX `man` pages. If you have access to a UNIX machine that supports STREAMS, you might find that the STREAMS "man" pages are installed. To test this out, try typing `man putmsg` on the UNIX command line.

You should also keep on eye on the Open Transport web page, which contains news and information for Open Transport developers. In addition, there are a number of non-Apple STREAMS-related sites on the Internet, including:

- the Mentat home page.
- Sun Microsystems' STREAMS Programming Guide.
- Dennis Ritchie's original STREAMS paper.
- The Digital UNIX STREAMS Programmer's Guide.

Finally, you should join the OT mailing list, which is a mailing list dedicated to solving Open Transport programming questions, at all levels of experience. See the OT web page for instructions on how to join.

Q What's the relationship between STREAMS and XTI?

A **XTI** is a standard API for accessing network services. STREAMS is a standard way of implementing networking services. Traditionally machines running STREAMS support an XTI API, although it is possible to support other types of APIs. For example, Open Transport supports a standard XTI interface, an asynchronous XTI interface, and classic networking backward compatibility, all on top of STREAMS. Also, UNIX STREAMS implementations commonly support a Berkeley Sockets API on top of STREAMS.

Q Isn't STREAMS slow?

A A poorly implemented STREAMS framework can slow down STREAMS-based protocol stacks. This is not true of MPS. Actual detailed performance measurements of MPS on multiple platforms have shown MPS's overhead to be negligible, and have shown that Mentat's STREAMS-based TCP outperforms various BSD-based TCP implementations.

STREAMS Modules and Drivers

Q I'm reading the STREAMS Modules and Drivers book described above and I can't make head or tail of it. Any suggestions?

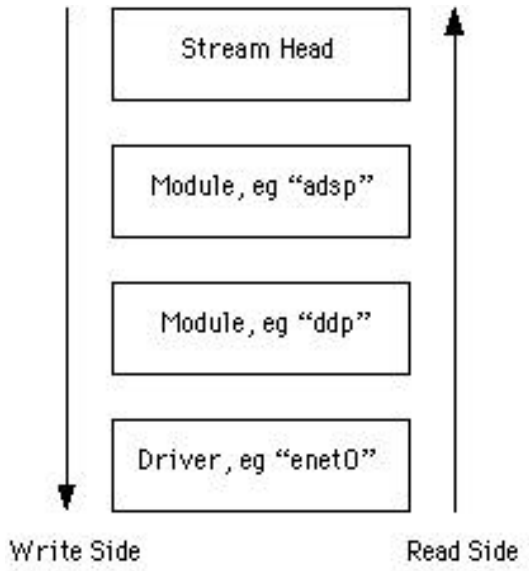
A I must admit that it wasn't until my third attempt at reading that book that I made any sense out of it. My secret? I found that if you print out a copy of the `mistream.h` header file and have it at hand while you're reading, it helps a lot.

Q What is a "stream"?

A In the most general definition, a stream (in lower case) is a connection oriented sequence of bytes sent between two processes. However, in the STREAMS environment, a **stream** normally refers to a connection between a client process and a network provider. For example, when you open a URL in a web browser, it creates a stream to the TCP module to transport connection information and data.

A stream carries the implication of instance. For example, there is only one Ethernet driver but it can support many different streams. One stream might be used by AppleTalk, one by TCP, and yet another by a network sniffing program.

Finally, a stream also implies a chain of modules, starting at the stream head and terminated at a driver. For example, if you open an endpoint "adsp,ddp,enet0", the system creates a new stream that looks like the one shown below.



Any data that you write to that endpoint starts at the stream head and proceeds first to the "adsp" module. That module can pass the data downstream (in this case towards the "ddp") with or without modifying it, or swallow the data completely, or reply to the data with a message sent upstream.

Q What is the stream head?

A The **stream head** is part of the STREAMS kernel. It is responsible for managing all interaction between the client and the modules. It works in concert with client side libraries that implement the actual networking APIs.

There are two keys areas of interaction: signals and memory copying.

Signals are a mechanism whereby the kernel can inform client code of certain events. Typically this is used for events like the arrival of data, but it is possible for modules to generate signals directly by sending the `M_SIG` message upstream. Obviously there is a connection between signals and OT's API-level notifiers.

Memory copying is the other main duty of the stream head. When you call an API routine (such as `OTSnd`), you're

actually calling the Open Transport client-side libraries. These libraries take the contents of your call (i.e. the data you want to send, or the address you want to connect to, etc.) and package it up into an STREAMS **message**. The client then calls the kernel to pass this messages to the stream head, and the stream head passes it down the stream. Once the data is packaged up into messages, no further data copying is done as these messages are passed around inside the kernel.

Because all data is transmitted between client and kernel using messages, there is only one point of entry between the client and the kernel. This means that STREAMS modules are not required to deal with client address spaces. This central location where the kernel accesses client memory decreases the risk of a protection violation on a protected memory system, and allows STREAMS modules to run in response to an interrupt without requiring a context switch.

Of course, there are some complications. For example, some API routines (especially `OTIoctl`) pass client addresses in message blocks. Modules can only gain access to the memory pointed to by these addresses by sending special messages up to the stream head. Remember, it's the stream head that does all of the interaction between client and kernel.

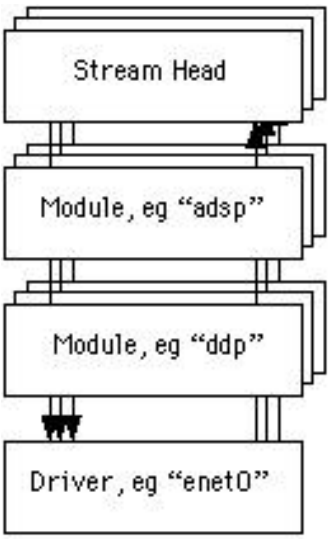
Q What's the difference between a module and a driver?

I asked this question when I was learning STREAMS and got the answer "A module can only be pushed, and a driver can only be opened." This answer is fundamentally correct, but it didn't help a lot at the time.

The real answer is that there isn't a lot of difference between the two; modules and drivers have a very similar structure. In most cases, STREAMS documentation says "module" when it mean "module or driver".

The big difference between a module and a driver is that a **driver** is the base of a stream. Streams pass through **modules**, but terminate at drivers. Thus modules must be **pushed** on top of an existing stream (because they need someone downstream of them), whereas drivers are always **opened** directly.

The following picture shows multiple AppleTalk streams all based on top of one Ethernet driver.

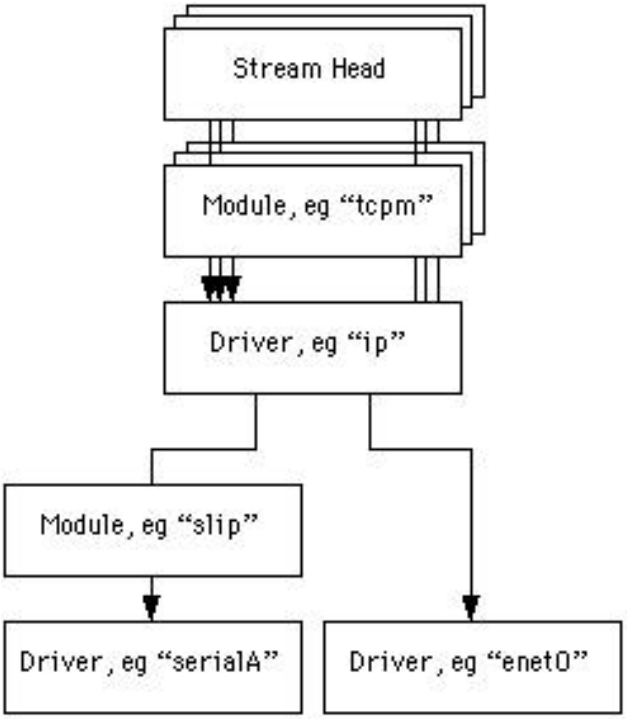


This is complicated by the existence of **multiplexing drivers**. Multiplexing drivers have both upper and lower interfaces. The upper interface looks like a driver, that is, it can be opened multiple times for multiple streams and appears to be the end of those streams. However you can also send a special `ioctl` call to the driver (`I_LINK`) to connect streams to the lower interface. At the lower interface, the multiplexing driver appears to be the stream head for those connected streams.

For example, you might implement the IP module as a multiplexing driver. IP has multiple upper streams (i.e. client processes using IP) and multiple lower streams (i.e. hardware interfaces over which IP is running) but there is no

one-to-one correspondence between these streams. IP uses one algorithm (routing) to determine the interface on which to forward outgoing packets. IP uses a second algorithm (protocol types) to determine which upper stream should receive incoming IP packets.

The following picture shows three TCP streams connected to a IP multiplexing driver, which is in turn connected to link layer ports, one run directly through an Ethernet driver, and the other through another stream that connects a SLIP module to a serial port.



NOTE:
In Open Transport, IP is not structured as a multiplexing driver, primarily for efficiency reasons. The above is just an example of how to think about multiplexing drivers. The next question explains how IP is really done.

Q I've noticed Open Transport has an "ip" driver and an "ipm" module. Why do some modules also appear as drivers?

A This is an implementation decision on the part of the module writer. In some cases, it's convenient to access a module as a module, and in other cases it's convenient to access it as a driver.

In this specific case, the MPS IP module behaves differently depending on whether it is opened as a module or a driver. When OT is bringing up the TCP/IP networking stack, it first opens the "ip" driver. IP recognizes that this first connection, known as the **control stream**, is special, and responds to it in a special way. Later, when OT is bringing up interfaces under IP (e.g. an Ethernet card and a PPP link), it first opens the link-layer driver and then pushes the "ipm" module on top of it. Each time OT does this, the IP module recognizes this special case and prepares itself to handle this new interface. Finally, when a client process actually wants to access IP services, OT opens the "ip" driver to create a new stream to it for the client.

STREAMS gives you a lot of flexibility, and the designers of MPS IP chose to use it.

Q What is this q parameter that's passed into each of my routines?

A The `q` parameter (which points to a `queue_t` data structure) is the fundamental data structure within STREAMS. Each time a stream is opened, STREAMS allocates a pair of queues (a **queue pair**) for each module in the stream. It then hangs all the stream-specific information off the queue pair.

One queue is designated the **write-side queue**. Data that the client sends to the stream is handled on the write-side queue. The other queue is the **read-side queue**. Data that the stream generates and sends to the client is handled on the read-side queue.

Each queue has a **put routine**, which is called whenever a message is sent to the module. The put routine has the choice of sending the message on to the next module (with or without modification), temporarily queuing the message on the queue for processing later, replying to the message by queuing the reply on the other queue in the queue pair, or freeing the message.

Each queue also has an optional **service routine** that is called when there is queued data to be processed. The service routine is optional because the module's put routine may be written in such a way that it never queues messages for later processing.

Because these routines are specific to a queue, modules tend to contain two routines of each type, one for the read side and one for the write side. These routines are known as the **read put routine**, **read service routine**, **write put routine**, and **write service routine**.

In addition, multiplexing drivers can have both **upper** and **lower** queue pairs, implying a total of eight entry points.

When called, each of these routines is passed a `q` parameter. The read-side routines are always passed the read queue and the write side routines are always passed the write queue. It's important to remember that each queue denotes a specific connection to your module and that queues are always created in pairs. So the `q` parameter passed to your module is really just a way of distinguishing stream instances.

Q I'm executing in a read-side routine (either a put or service routine) and I need access to the write-side queue. How do I find it?

A Queue structures are actually allocated in memory as pairs, butted up right next to each other, with the read queue immediately preceding the write queue. Given that `q` is a pointer to the read queue, you can derive the write queue using the C construct `&q[1]`. However an even better solution is to use the macros `RD`, `WR` and `OTHERQ` defined in "mistream.h".

Q How do I store global data in my module?

The best way to store globals in your module is just to declare global variables. Because modules are shared libraries, you don't need to do anything special to access these globals. Note that these globals are shared across all instances of your module, i.e. all streams that run through your module.

NOTE:
There is one exception to the above statement. If you have two PCI (or PC Card 3.0) cards installed, OT will create a separate instance of the CFM-based driver for each card. So the driver will have a copy of it's global variables for each installed card. The driver distinguishes which card its driving by `RegEntryID`, passed as a parameter to its `InitStreamModule` routine.

If you want to store globals on a per-stream basis, you have to do a little more. The following snippet demonstrates the recommended technique.

```
// First declare a data structure that holds all of the
// data you need on a per-stream basis.

struct MyLocalData
```

```
{
    OSType magic;           // 'ESK1' for debugging
    long   currentState;    // TS_UNBND etc.
    [...]
};
typedef struct MyLocalData MyLocalData, *MyLocalDataPtr;

// Then declare a global variable that acts as the head of
// the list of all open streams.

static char* gModuleList = nil;

// In your open routine, call mi_open_comm to create
// a copy of the global data for this new stream.

static int MyOpen(queue_t* rdq,
                          dev_t* dev, int flag,
                          int sflag, cred_t* creds)
{
    MyLocalDataPtr locals;

    [...]
    err = mi_open_comm(&gModuleList,
                          sizeof(MyLocalData),
                          rdq,
                          dev, flag,
                          sflag, creds);
    if ( err == noErr ) {

        // mi_open_comm has put a pointer to our per-stream
        // data in the q_ptr field of both the read-side
        // and write-side queue.

        locals = (MyLocalDataPtr) rdq->q_ptr;
        locals->magic = 'ESK1';
        locals->currentState = TS_UNBND;
        [...]
    }
    [...]
}

// In your close routine, use mi_close_comm to destroy
// the per-stream globals.  Note that, if you have
// any pointers in your data, you must make sure to
// dispose of those before calling mi_close_comm.
//
// As an alternative to mi_close_comm, you might want to
// use mi_detach and mi_close_detached.

static int MyClose(queue_t* rdq, int flags, cred_t* credP)
{
    [...]
    (void) mi_close_comm(&gModuleList, rdq);
    [...]
}

// If you find that you need to loop through all the
// streams open through your module, use the mi_next_ptr
// routine as shown below.

static void MyForEachStream( [...] )
{
    MyLocalDataPtr aStreamLocals;

    aStreamLocals = (MyLocalDataPtr) gModuleList;
```

```
    while (aStreamLocals != nil) {
        [...]
        aStreamLocals = (MyLocalDataPtr) mi_next_ptr( (char *) aStreamLocals );
    }
}
```

The Open Transport Module Developer Note has a full description of the routines used in the above snippet.

Q How do I synchronize access to my global data?

A MPS provides support for synchronizing access to global or per-stream data. When you install your module, you must fill out the `install_info` structure. One of the fields in this structure is `install_sqlvl`, which you set to control your module's reentrancy.

NOTE:
When reading this description, it's important to keep the following abbreviations in mind. In the context of MPS, "SQ" stands for **synchronization queue**, which is the key data structure that MPS uses to guard against reentrancy. Also, "SQLVL" stands for **synchronization queue level**, which is the degree of mutual exclusion needed by a module.

The legal values for the sync queue level are:

`SQLVL_QUEUE`
: Your module can be entered once per read or write queue. This means that you must guard your global data from access by multiple threads running in you module, and you must guard your per-stream data from access by threads running on the read and write sides of the stream simultaneously.

`SQLVL_QUEUEPAIR`
: Your module can be entered once per queue pair. You must still guard your global data from access by multiple threads running in your module, but your per-stream data is safe from simultaneous access by the read and write sides.

`SQLVL_SPLITMODULE`
: [This sync queue level is not yet supported in Open Transport, and is documented here for completeness only.] Your module can be entered once from an upper queue and once from a lower queue. With this sync queue level, the `mps_become_writer` function is relatively cheap, and this is the recommended sync queue level for network and link-layer drivers.

`SQLVL_MODULE`
: Your module can only be entered once, no matter which instance of the module is entered.

`SQLVL_GLOBAL`
: Between all modules that use `SQLVL_GLOBAL`, only one will be entered at a time.

In the above list, sync queue levels are given from least exclusive (`SQLVL_QUEUE`) to most exclusive (`SQLVL_GLOBAL`). In general, the least exclusive sync queue level also yields the best system performance, while the most exclusive value leads to the worst system performance. However this is not guaranteed. If, by setting your sync queue level to `SQLVL_QUEUE`, you are forced to make a significant number of calls to `mps_become_writer`, you may find better performance with a more exclusive sync queue level.

NOTE:
If your module is using sync queue levels `SQLVL_QUEUE`, `SQLVL_QUEUEPAIR`, or `SQLVL_SPLITMODULE`, you can use the `mps_become_writer` function to ensure that only one thread of execution is inside a particular part of your module at any given time. See the Open Transport Module Developer Note for a description of `mps_become_writer`.

So, what does this mean in practical terms? Before OT calls your module (either the put routine or the service routine), it checks to see whether there is a thread of execution already running in your module. If there is, it checks the sync queue level of the module to see whether calling your module would be valid at this time. It uses these two factors to decide whether to call your module immediately, or queue the call for some later task to execute.

The sync queue levels fall into two categories:

1. Queue-based sync queue levels (i.e. `SQLVL_QUEUE` and `SQLVL_QUEUEPAIR`) are centred around the queue pairs associated with each stream that's opened to your module. If you use `SQLVL_QUEUE`, your module can be reentered as long as the put or service routine for *that queue* isn't already running. If you use `SQLVL_QUEUEPAIR`, your module can be reentered as long as a put or service routine for *that queue pair* (i.e. the stream) isn't already running
2. Module-based sync queue levels (`SQLVL_SPLITMODULE`, `SQLVL_MODULE` and `SQLVL_GLOBAL`) work on a module-by-module basis. For the moment, you can ignore `SQLVL_SPLITMODULE`. With `SQLVL_MODULE`, your module cannot be reentered at all. With `SQLVL_GLOBAL`, your module is mutually excluded against all other module that are marked as `SQLVL_GLOBAL`. [This can be useful if you're trying to bring up a suite of modules that talk to each other.]

Of course, these mutual exclusion guarantees are for when STREAMS calls you, i.e. your open, close, put and service routines. If you are called by other sources (such as a hardware interrupts), you have to take additional measures to ensure data coherency. Of course, OT provides support for this too. See the Open Transport Module Developer Note for a description of the routines you can call from your hardware interrupt handler.

In general, I recommend that you first use `SQLVL_MODULE` in order to get your module working. Then, once you understand the data coherence issues in the final code, analyze the code to see if you can use a better sync queue level.

IMPORTANT:
If your are pushing your module into an existing protocol stack, you should be sure to check the sync queue level of the other modules in the stack. If the existing modules have a very exclusive sync queue level, there is nothing to gain by engineering your module to have a non-exclusive level. Conversely, if the existing modules have a non-exclusive sync queue level, you could affect the performance of the entire protocol stack by adopting a very exclusive level.

Q I'm confused by the `qinit` structures. I need to have two `qinit` structures, one for the read side and one for the write side, but that implies two open and close routines. Two open routines seems like a recipe for confusion. What the full story?

A For the open and close routines, STREAMS only looks at the read-side `qinit` structure.

Q How should I structure my STREAMS module?

STREAMS modules have two primary entry points, the put routine and the service routine. In general, you should try to do all the work you can in your put routine. This is contradictory to most of the STREAMS documentation, and is an important factor in making your modules fast.

Every time you use `putq` to put a message on your queue, STREAMS must schedule a task to run your service routine in order to service that message. While OT's internal task scheduler is fast, it still takes time.

The alternative is to process the message in your put routine and then immediately send the message on to the next module (using `putnext`) or reply to the message (using `qreply`). This can make your put routine complicated. If you find that your put routine is getting too complicated, simply break it up into subroutines. The cost a subroutine call is much less than the cost of scheduling your service routine.

Of course you can still use `putq` in flow control conditions because, if you're flow controlled, you don't really care about speed.

Q How does flow control work?

STREAMS flow control is quite hard to understand. The basic tenets of STREAMS flow control are:

- Your module either takes part in flow control, or it doesn't. If it doesn't take part in flow control (i.e. it's a simple filter module), you should let STREAMS know by having no service routine. You can then ignore the other rules given below.
- High priority messages are not subject to flow control. It's important that your module avoid enqueuing them because of flow control because this can cause a deadlock situations (i.e. you can't flush the messages out of a stream because the stream is flow controlled).
- Flow control is governed by two values in the queue, the high and low water marks. If the number of bytes of messages stored on a queue is greater than the **high water mark**, the queue is flow controlled. The queue stays flow controlled until the number of bytes of messages enqueued falls below the **low water mark**.
- Bytes go on to your queue when you call `putq`. This has the side effect of scheduling your service routine. [You can also schedule your service routine directly using `qenable`.] Your service routine only gets run once regardless of how many times you schedule it.
- Bytes come off your queue when you call `getq` in your service routine. When you get a message like this, you should call `canputnext` to see if it's possible to put the data on the next queue that has a service procedure. If it is, call `putnext` to put it on the next queue. If it isn't, call `putbq` ("put back on queue") to return the message to your queue. Calling `putbq` puts the data back on to your queue without rescheduling your service routine.
- Because your service routine can only be scheduled once, it is critical that your service routine finish either by calling `putbq` or by completely draining the queue (i.e. `getq` returns nil). A sample service routine is shown below.

```
// A standard read service routine that follows the
// above guidelines.

static int MyReadService(queue_t* q)
{
    mblk_t *mp;

    while ( (mp = getq(q)) != nil ) {

        // Never putbq a high-priority message.

        if ((mp->b_datap->db_type < QPCTL) && !canputnext(q)) {
            putbq(q, mp);
            return (0);
        }

        // Handle the message then put it on the next queue

        [...]

        putnext(q, mp);
    }

    return (0);
}
```

- When your queue is flow controlled, the previous module's read service routine will stop being able to put messages on your queue (because its calls to `canputnext` will return false). This causes that module to call `putbq`, which puts the data on their queue without scheduling their read service routine. Eventually this causes the number of bytes in their queue to exceed their high water mark, which causes them to be flow controlled as well. This process proceeds up the stream until you get to the stream head or the driver.
- When the stream head gets flow controlled, it stops accepting data from the client, and the client blocks waiting for data to be sent.
- When the driver gets flow controlled, it either

a) starts dropping packets (for unreliable services, such as Ethernet), or

b) it raises the link-layer flow control (for reliable services, such as serial).

- When flow control is lifted (this happens when the number of bytes in the flow controlled queue drops below the low water mark -- for the read side, this is because the client reads some data; for the write side, it happens when the driver transmits some data), STREAMS automatically reschedules the service procedure of the previous queue that has a service routine. Like the propagation of flow control, this **back enabling** continues until it reaches the beginning of the stream.

Finally, there is one important hint for using flow control. In certain special case situations, such as constructing a sequence of messages, it may be extremely inconvenient to deal with flow control. At times like this, you always have the option of ignoring it. While not strictly legal, this will work and is unlikely to get you into trouble. But it is important that you deal with flow control in the general case, otherwise messages will pile up on queues, and STREAMS will run out of memory.

Q Which should I use, `canputnext` or `putnext(q->q_next)`?

A STREAMS Modules and Drivers contains a number of code samples that look like:

```
#ifdef MP
        if ( canputnext(q, mp) ) {
#else
        if ( putq(q->q_next, mp) ) {
#endif
```

This is an anachronism from UNIX STREAMS's support of multi-processor (MP) systems. MPS STREAMS has full support for MP built-in, so `canputnext` is always available. In addition, MPS automatically handles synchronization across multiple processors using sync queues (see the question How do I synchronize access to my global data?), so you do not have to worry about MP issues in your OT modules.

Q I've notice that some STREAMS routines return `int` even though there is no defined returned value. When I check the returned values, I find that they are random. What's going on?

A The STREAMS internal routines were imported wholesale from UNIX and, in some cases, the prototypes do not match the semantics. In these cases, you should make sure to ignore the returned value.

Q STREAMS Modules and Drivers talks a lot about **bands**. Is this of any use?

A Not really. Some protocol modules (such as TCP and ADSP) have the concept of expedited data and typically these are supported using band 0 (normal data) and band 1 (expedited data). No one has ever found a use for all 255 bands!

Also, note the band structures inside STREAMS are allocated as an array, so if you use more than one band, make sure you allocate them sequentially from 0. Otherwise you might find yourself using a lot more memory than you expect.

Finally, you should remember that bands only affect the order in which messages are queued, and hence the order in which they are returned by `getq` to the service routine. As an efficient STREAMS protocol stack will rarely queue messages, bands are rarely useful. One case where they have a significant effect is on the stream head, where the band affects the order in which data is delivered to the client. However, this effect may not be the effect you are looking for!

Q What fields of the `queue_t` structure can I modify?

A There are a number of rules related to the fields in a queue:

- You should only modify your own queues. You should not modify the queue of another module.
- The `q_ptr` field is specifically reserved for the module's own use. The module can read or write that value at any time. Note, however, that if you use `mi_open_comm` (which I strongly recommend), the `q_ptr` field of both queues in the queue pair contains a pointer to your per-stream data, and you should not use if for anything else.
- Although it is normally OK to just read the queue's fields directly, you really should read them using `strqget`. This avoids some possible synchronization issues.
- You must modify any fields (other than `q_ptr`) using `strqset`.

`strqget` and `strqset` are defined with the following prototypes in "mistream.h":

```
extern int strqget(queue_t*, qfields_t what, uchar_p pri, long* valp);
extern int strqset(queue_t*, qfields_t what, uchar_p pri, long val);
```

`strqget` is used to read a field, putting the value in the long pointed to by `valp`. `strqset` is used to set a field. The field that is modified is determined by the `what` parameter, whose value can be:

qfields_t	field in queue_t	read-only?
QHIWAT	q_hiwat	no
QLOWAT	q_lowat	no
QMAXPSZ	q_maxpsz	no
QMINPSZ	q_minpsz	no
QCOUNT	q_count	yes
QFIRST	q_first	yes
QLAST	q_last	yes
QFLAG	q_flag	yes

The `pri` parameter determines which priority band is used. A band of 0 indicates the value held in the queue itself, a higher value refers to the band data structure referenced by the queue.

The functions can return the following errors:

- `ENOENT` if an invalid `what` is specified
- `EINVAL` if the specified band is not currently defined
- `EPERM` if you are not allowed to modify the specified field

IMPORTANT:
You should not modify fields that are marked as read-only in the above table. While it may seem like a convenient shortcut, it will cause you problems in the long run. This warning applies specifically to the `q_flag` field.

Q The standard UNIX STREAMS books do not contain any information about the routines that begin with the prefix `mi_`, for example `mi_open_comm`. Where are these documented?

A These are utilities routines provided by Mentat to make STREAMS programming easier. They are documented in the Open Transport Module Developer Note. I strongly recommend that you use these routines because they help cut down on silly programming errors.

Messages and Memory Allocation

Q Can I modify the message blocks that are passed to my module?

A Yes, as long as you are careful. To start with, you must distinguish between message blocks (`mblk_t`) and data blocks (`dblk_t`). Message blocks are always wholely owned by you. STREAMS passes you the message block, and you are expected to remember it, free it, or pass it on. No one else has a reference to that message block. For this reason, you are always allowed to modify the fields in the message block, even if you aren't allowed to modify the data block.

The following fields of the `mblk_t` are commonly modified: `b_cont`, `b_rptr`, `b_wptr`. You should not directly change the other fields in the `mblk_t`; there are STREAMS routines that let you change them indirectly.

Data blocks are slightly different. A single data block can be referenced by multiple message blocks, so you are only allowed to modify the fields in the data block (or indeed its contents) if you are the sole owner of the block. You determine this by looking at the `db_ref` field of the data block. If it is set to 1, you are free to modify the data block and its contents. If it is greater than one, some other message block has a reference to this data block, and you should avoid modifying the data block or its contents.

If you wish to write to a read-only data block, you should copy the block using one of the allocation functions described below.

The only field of the `dblk_t` that is commonly modified is `db_type`. You should not directly change the other fields in the `dblk_t`, although there are STREAMS routines that let you change them indirectly.

Q How do I allocate new messages within my module?

A There are a lot of techniques. If you just want to allocate a raw message along with its data block, use the STREAMS function `allocb`. Given a size, `allocb` will create a message that pointers to a data buffer of at least that size.

`copyb` returns a new message block that's identical to the input message block. The data block that the message block points to is also copied.

`copymsg` returns a new message that's identical to the input message. Like `copyb`, it also copies the data that the message block points to. In addition, it copies all of the message blocks linked to this message through the `b_cont` field, and all their data blocks.

`dupb` duplicates the message block you passed into it without copying the data block that the message points to. The new message continues to reference the old data block. The function also increments the `db_ref` field of the data block to record the new copy.

`dupmsg` duplicates the message block you passed into it without copying the data block that the message points to. In addition, it duplicates all of the message blocks linked to this message through the `b_cont` field.

`esballoc` creates a message block that references a data block which you provide. You also pass in a function that will be called when the message is freed. This allows DMA-based network drivers to implement no-copy receives by passing their real DMA buffers upstream. See Open Transport Module Developer Note for more hints and tips on `esballoc`.

There are also a number of utilities routines for allocating TPI messages that you might find useful. These include:

```
mi_tpi_conn_con                 mi_tpi_uderror_ind
mi_tpi_conn_ind                 mi_tpi_unitdata_ind
mi_tpi_conn_req                 mi_tpi_unitdata_req
mi_tpi_data_ind                 mi_tpi_exdata_ind
mi_tpi_data_req                 mi_tpi_exdata_req
mi_tpi_discon_ind               mi_tpi_ordrel_ind
mi_tpi_discon_req               mi_tpi_ordrel_req
mi_tpi_info_req
```

See the Open Transport Module Developer Note for more details on these routines.

Q Why do I get a link error when I try to use `mi_tpi_data_ind` from my module?

A It appears that someone forgot to export that routine. Fortunately, it's very easy to write you own version:

```
static mblk_t* qmi_tpi_data_ind(mblk_t* trailer_mp, int flags, long type)
{
        mblk_t* mp;

        mp = mi_tpi_data_req(trailer_mp, flags, type);
        if (mp)
                ((struct T_data_ind *)mp->b_rptr)->PRIM_type = T_DATA_IND;
        return mp;
}
```

Q How do I reuse an existing message?

A In writing a module, you often find yourself in the situation where you want to free a message and then allocate a new message in reply to the original message. In these cases, it's much better to reuse the first message rather than suffer the overhead of the freeing one message and allocating another.

You can reuse a message block as long as both of the following conditions are true:

- You are the sole owner of the message, i.e. the messages's data block field `db_ref` is 1.
- The message is big enough for your needs.

STREAMS guarantees that all control messages generated by `putmsg` (typically `M_PROTO` and `M_PCPROTO`) reference data blocks that are at least 64 bytes long.

OT provides utility routines for reusing messages. The most general purpose one is:

```
mblk_t* mi_reuse_proto(mblk_t* toReuse,
                       size_t sizeDesired,
                       boolean_p keepOnError);
```

This routine attempts to reuse the message pointed to by `toReuse`, making sure that the message can contain `sizeDesired` bytes. It return a pointer to the new message, or nil if it fails. If `keepOnError` is false, `toReuse` is freed regardless of whether we fail or not. Otherwise, `toReuse` is preserved if we fail.

There are also a number of utilities routines specific to TPI that you might find useful including:

```
mi_tpi_ack_alloc                    mi_tpi_err_ack_alloc
mi_tpi_ok_ack_alloc
```

See the Open Transport Module Developer Note for more details on these routines.

WARNING:
All of these reuse routines can return nil if you run out of memory. The reason is that the message you're trying to reuse may be read-only, in which case the routine is required to create a copy of the message. This copy can fail if you run out of memory. You must be prepare for these routines to fail.

Q How much data is in a message?

A If you just want to know how much data there is in a single message block, you can simply calculate `b_wptr - b_rptr`. If you want to find the total size of all the messages in a chain, use the STREAMS function `msgdsize`. Note that this function returns the number of data bytes in the message, and does not take into account `M_PROTO` and `M_PCPROTO` message blocks.

Q How much space is there in a message?

A If you just want to know how much space is available in a single message block, you can simply calculate `db_lim - db_base`. As far as I know, there is no way to calculate this for all the messages in a chain.

Q Are there any invariants that I can use to keep my message blocks straight?

A Yes. The invariants are that:

```
mp->b_datap->db_base < mp->b_datap->db_lim
mp->b_datap->db_base <= mp->b_rptr < mp->b_datap->db_lim
mp->b_datap->db_base <= mp->b_wptr <= mp->b_datap->db_lim
mp->b_rptr <= mp->b_wptr
```

These invariants imply that:

- there is always at least one byte of space in a message
- the read and write pointers always point within the data
- the amount of valid data in the message is always non-negative

---

Q A lot of STREAMS allocation functions (e.g. `allocb`) take a buffer allocation priority value. What should I use?

A At the moment, STREAMS is defined to ignore these values. There are two reasonable approaches:

1. Ignore priorities and always pass the unspecified priority, i.e. 0.
2. Analyze your buffer needs and set your priorities appropriately on the assumption that one day someone will pay attention to them. For example, most data messages would default to `BPRI_MED`, but high priority control messages like TPI ACKs should use `BPRI_HI`.

I recommend the first approach.

Q What do I do when an allocation fails?

A The approach you take depends on the type of module you are writing. If you are writing a module that provides an unreliable service (such as a DLPI device driver), the best thing to do when you run out of memory is to just drop the current packet on the floor. Because you are providing an unreliable service, the upper-layer protocol is required to implement some error correction anyway, so there's no point complicating your module with intricate error handling.

If you're writing a reliable service, you must be prepared to deal with running out of memory. Your primary weapon should be the `mi_bufcall` routine. This routine allows you to stop your current operation and schedule your queue's service routine to be called when a certain amount of memory is available. You then have a flag in your per-stream data that allows your service routine to pick up the stalled operation before continue on with its normal duties.

See the Open Transport Module Developer Note for more details on `mi_bufcall`.

IMPORTANT:
You should use `mi_bufcall` in preference to the more traditional `bufcall`. See the developer note for the reasons why.

Transport Provider Interface (TPI)

Q I'm writing a STREAMS TPI module or driver. Where should I start?

A The best book to read is STREAMS Modules and Drivers. In terms of sample code, there are a number of samples to look at:

- TPIFile -- Available on the Developer CDs, this sample is a TPI device driver that allows you to read a Mac OS file as if it was an OT serial port.
- StreamNOP -- Available on the Developer CDs, this is a cut down version of TPIFile that serves as good starting point for new module development.
- tilisten -- Part of the OT Module SDK, this sample contains the full source to the "tilisten" module (a helper module used to simplify the listen/accept process for clients).

None of these samples are perfect, but they do give a flavour of what STREAMS programming is like.

Q I'm receiving a TPI message. Can I reuse that message to send the ACK?

A See the question How do I reuse an existing message?

Q I'm writing a TPI module and I successfully respond to a `T_CONN_REQ` message by sending a `T_OK_ACK` message upstream, but my client code never leaves `OTConnect`. What did I do wrong?

A The short answer is that you need to send a connection confirmation message (`T_CONN_CON`) upstream to indicate that the connection is in place.

The long answer is that you need to study the TPI specification more closely, paying special attention to the state diagrams. When your module is in `sta_3 (idle)` and receives a `conn_req` event, it should proceed to `sta_5 (w_ack_c_req)`. When your module replies with the `ok_ack1`, it proceeds to `sta_6 (w_con_c_req)`. At this point the client is still waiting for a connection confirmation message. To

complete the connection sequence, you need to issue a `conn_con` event and proceed to `sta_9 (data_t)`.

I find it useful to think of the `T_OK_ACK` as simply saying that the primitive being acked was correctly formed; it says nothing about whether the request worked. If a response is needed, TPI typically has a different message (e.g., `T_BIND_ACK` or `T_INFO_ACK`). In the case where something needs to be done, like connection setup, a distinct message `T_CONN_CON` is used to 'confirm' the connection is established.

Q The TPI specification says that the address to connect to is pointed to by the `DEST_offset` and `DEST_length` fields of the `T_CONN_REQ` message. I know how to find the address of this information (using `mi_offset_paramc`) but what is its format?

A There are two aspects to this question. First, how do Open Transport clients provide address information. Second, how does Open Transport translate that client representation into a TPI message.

Open Transport uses a standard format for address information that's based on the `OTAddress` type. This type is an abstract record that contains only one interesting field:

```
struct OTAddress
{
    OTAddressType fAddressType;
    UInt8         fAddress[1];
};
```

The `fAddressType` field is a two-byte quantity that determines the format of the remaining fields. All Open Transport addresses are derived from this basic structure. For example, in the TCP/IP world, OT has two different address formats, namely `InetAddress` and `DNSAddress`.

```
struct InetAddress
{
    OTAddressType  fAddressType;  // always AF_INET
    InetPort       fPort;         // port number
    InetHost       fHost;         // host address in net byte order
    UInt8          fUnused[8];    // traditional unused bytes
};
struct DNSAddress
{
    OTAddressType  fAddressType;  // always AF_DNS
    InetDomainName fName;         // ASCII DNS name
};
```

These are distinguished by the first two bytes. An `InetAddress` starts with `AF_INET`, while a `DNSAddress` starts with `AF_DNS`. These type bytes are followed by an address-format specific number of bytes of data. This general layout is common to all address formats under OT.

When you call an OT API routine and pass in an address like this, OT simply copies the entire address into a message block without interpreting it. When the message reaches the appropriate TPI module, that module is responsible for interpreting the specified address. It can determine that the address is in the appropriate format simply by looking at the first two bytes of the address buffer. The snippet of code in the next Q&A shows how to do this.

Q TPI messages often contain "offset" and "count" parameters to reference variable length data. Every time I access these, I find myself dying the 'death of a thousand pointers'. Is there a better way?

A I'm glad you asked. MPS provides two useful utility routines that you can call to access these variable

length structures. There prototypes are:

```
UInt8* mi_offset_param(mblk_t* mp, long offset, long len);
UInt8* mi_offset_paramc(mblk_t* mp, long offset, long len);
```

If you have a simple TPI message (one with a single message block), you can call `mi_offset_param` to get a pointer to the structure whose size is `len` at the given `offset` into the message data. The routine returns nil if `offset` and `len` are inconsistent with the size of the message.

If there's a possibility that the data you're looking for is not in the first message block of the TPI message, you can use `mi_offset_paramc` to look for it in the entire message chain.

The following snippet shows how you can use `mi_offset_paramc` to find the address in a `T_CONN_REQ` message.

```
static void DoConnectRequest(queue_t* q, mblk_t* mp)
{
    T_conn_req *connReq;
    OTAddress *connAddr;

    [...]

    connReq = (T_conn_req *) mp->b_rptr;

    [...]

    connAddr = (OTAddress *) mi_offset_paramc(mp,
                                connReq->DEST_offset,
                                connReq->DEST_length);
    if (connAddr == nil || connReq->DEST_length < sizeof(OTAddress)) {
        ReplyWithErrorAck(q, mp, TBADADDR, 0);
        return;
    }
    switch ( connAddr->fAddressType ) {
        [...]
    }

    [...]
}
```

Q In my TPI module I send data messages but they never arrive on the wire/at the client. Any ideas?

A You have most probably forgotten to set the `b_wptr` field of the message that you are sending. If you allocate a new message block, the `b_rptr` and `b_wptr` both default to pointing at the start of the data block (i.e. `db_base`). Given that the amount of valid data in the message is defined to be `b_wptr - b_rptr`, if you forget to set the `b_wptr` on messages you will find that the receiver ignores them.

Data Link Provider Interface (DLPI)

Q I'm writing a STREAMS DLPI driver. Where should I start?

A You should start with one of the generic STREAMS references listed above, then continue with the following Open Transport-specific material.

- If you're writing a Ethernet-style device driver (Ethernet, Token Ring, FDDI, Fibre Channel, etc.), you should base it on the Apple Enet Framework, which is included in OT Module SDK. This framework significantly reduces the amount of work you have to do, and guarantees the best performance for high-speed devices.
- On the other hand, if you're writing something other than an Ethernet-style device driver (PPP, SLIP, virtual private network (VPN), etc.), you should read the book Open Transport Advanced Client Programming. The Implementation Notes chapter gives specific advice for PPP and VPN developers.

Q What's this stuff about connection-oriented DLPI drivers?

A I have no idea! As far as the OT mainstream is concerned, all DLPI drivers are connection-less (`DL_CLDLS`). In fact, when OT needs a connection-oriented device driver (e.g. serial), it uses TPI instead of DLPI. However, connection-oriented DLPI drivers may be useful in some environments, such as X.25 or ATM.

Q I'm writing the code to fill out the `DL_INFO_ACK` message, and I can't decide what to put in the `dl_provider_style` field. I'd like to use `DL_STYLE1` (because then I don't have to mess with Physical Points of Attachment (PPAs)) but it seems I should be using the later `DL_STYLE2`. What do you recommend?

A Unless you have an overriding reason to use PPAs, you should return `DL_STYLE1` in your `DL_INFO_ACK` message. This will make your life easier and there's little need for PPAs on the Mac OS.

Q What is this stuff about major and minor device numbers?

A The short answer is: an anachronism from UNIX. Major device numbers represent the device driver controlling a device. This is traditionally an index into a table of drivers maintained internally by STREAMS. Under Open Transport, drivers are loaded into this table on demand, so there's no way you can know what major device number your driver is going to get.

Minor device numbers are used to distinguish between multiple functions controlled by a single device driver, for example, multiple serial ports controlled by the serial port driver. However, this definition breaks down in the face of networking, even on UNIX systems.

It turns out that minor device numbers are used to distinguish between different streams connected to a driver. Each stream is given a unique minor device number by the driver's open routine. This is accomplished by means of the `sflag` parameter. The three possible cases are:

- `0` -- This value indicates that the module is being opened as a driver. A specific minor device number -- specified by the `devp` parameter -- is being opened.
- `CLONEOPEN` -- This value indicates that the driver is being cloned, i.e. the driver should return a unique minor device number. You can do this simply by calling `mi_open_comm`, which does this automatically.
- `MODOPEN` -- This value indicates that the module is being pushed; there is no minor device number in this case.

So how does this affect you? It doesn't! If you call `mi_open_comm` in your module's open routine, it takes care of all these details. Your open routine might also want to check that you are being opened as a module (i.e. `sflag == MODOPEN`) or as a driver (`sflag != MODOPEN`), just to be paranoid. But, otherwise, you should not worry about device numbers and distinguish your streams using the `q` parameter.

Summary

Open Transport is based on an industry standard STREAMS networking kernel. Open Transport STREAMS is documented in a number of UNIX books. and in the Open Transport Module Developer Note . This Note answers some Frequently Asked Questions about issues that are not adequately covered in the other documentation.

Further References

- *Programmer's Guide: STREAMS, UNIX System V Release 4* , UNIX Press, ISBN 0-13-02-0660-1
- *STREAMS Modules and Drivers, UNIX System V Release 4.2* , UNIX Press, ISBN 0-13-066879-6
- "Open Transport Module Developer Note" (part of the OT Module SDK)
- UNIX "man" pages for "putmsg", "getmsg", etc.
- Open Transport web page
- Open Transport programmers mailing list

Downloadables

Acrobat version of this Note (K).

Change History

- First published in February 1998.
- Updated in February 1999 to add additional references.
- Updated in March 1999 to mention "Open Transport Advanced Client Programming" in the recommendation for DLPI device driver writers.

To contact us, please use the Contact Us page.
Updated: 15-February-98