

Open Transport Module Developer Note

PRELIMINARY
Revision 1.5d2
06/18/96

Table of Contents

Revision History.....	4
Related Documents.....	4
Module Development for Open Transport.....	5
Dynamic Loading.....	6
Building Modules and Drivers.....	6
Module exports.....	6
GetOTInstallInfo.....	7
InitStreamModule.....	11
TerminateStreamModule.....	12
Building Modules with ASLM.....	13
68K ASLM Modules.....	13
PPC ASLM Modules.....	14
Building Modules with CFM.....	17
Working with Port Drivers.....	18
APIs for Port Drivers.....	19
OTRegisterPort.....	19
OTUnregisterPort.....	24
OTChangePortState.....	24
OTGetIndexedPort.....	25
OTFindPort.....	25
OTFindPortByRef.....	25
OTFindPortByDev.....	25
Registering Port Drivers.....	27
Port Driver Configuration Info.....	30
Module and Driver Operation.....	32
Interrupt-Safe functions.....	32
Secondary Interrupt Services.....	33
Timer Services.....	34
Atomic Services.....	35
Power services.....	37
Memory allocation functions.....	38
OTAllocMem.....	38
OTFreeMem.....	38
alloca.....	38
free.....	38
freemsg.....	38
dupb.....	38
dupmsg.....	38
copyb.....	39
copymsg.....	39
OTAllocMsg.....	39
OTAllocReadOnlyMsg.....	39
Open and Close support Code.....	40
mi_close_comm.....	40
mi_close_detached.....	41
mi_open_comm.....	42

mi_next_ptr.....	43
mi_open_detached.....	44
mi_bufcall.....	45
IOCTL Support functions.....	46
mi_copyin.....	47
mi_copyout.....	48
mi_copyout_alloc.....	48
MI_COPY_CASE.....	49
mi_copy_done.....	49
mi_copy_set_rval.....	50
mi_copy_state.....	50
TPI Support functions.....	51
mi_tpi_ack_alloc.....	51
mi_tpi_err_ack_alloc.....	51
mi_tpi_ok_ack_alloc.....	51
Other TPI prototypes.....	52
Synchronization support.....	53
IOCTL Messages.....	53
Appendix A- Synchronization.....	57
Appendix B- Performance hints.....	60
Appendix C - Random Notes/Warnings.....	62
Index.....	64

Revision History

05/30/95	Some updates for version 1.5 of Open Transport
01/18/95	Updated for 1.1b14
8/28/95	Updated for 1.1b2 (partially - still more to do)
11/28/94	Update for 1.0a2 Open Transport. Changes to ValidateHardware call and removed one timer call.
09/16/94	Update for 1.0d16 Open Transport and new Power calls
07/25/94	Merged with Open Transport Ethernet Developer Note, with additions for PCI Bus development
06/16/94	Minor corrections
05/04/94	Revised for new address formats
02/14/94	Creation

Related Documents

Data Link Provider Interface Specification Unix International, OSI Workgroup

Transport Provider Interface Specification Unix International, OSI Special Interest Group, Revision 1.5 (Date: 92/12/10)

Streams Modules and Drivers Unix® SVR4.2 UNIX Press

Apple Shared Library Manager Developer's Guide, by ESD Publications, October 4, 1993, Apple Computer, Inc.

Open Transport Client Developer Note

Designing PCI Cards and Drivers for Power Macintosh Computers, Apple Computer, Inc.

Module Development for Open Transport

Open Transport has chosen the STREAMS model for implementing protocols and drivers. This provides a large amount of flexibility for mixing and matching protocols. It also allows a wide range of third-party STREAMS modules and drivers to be easily ported to the Open Transport environment.

Part of the flexibility of the STREAMS environment comes from being a messaging interface with only a few well-defined messages. The most common types of messages are M_DATA (for sending raw data), M_PROTO (for sending normal commands), and M_PCPROTO (for sending high-priority commands). Since STREAMS does not define the content of M_PROTO or M_PCPROTO messages, it is necessary for modules to agree on a message format if they are to communicate.

Open Transport has standardized on the Transport Provider Interface (TPI) message format for most protocol modules, and the Data Link Provider Interface (DLPI) for most STREAMS hardware drivers.

This document describes what must be done to create STREAMS modules and drivers for Open Transport. It assumes that you are familiar with the material in the *Streams Modules and Drivers for Unix® SVR4.2* published by Unix Press, as well as the TPI and DLPI specifications (see the Related Documents section for references).

Open Transport classifies STREAMS modules into three different categories. These are modules, drivers, and port drivers.

A module is a STREAMS module that expects MODOPEN to be set in the sflags parameter of its open routine. It is always "pushed" onto other modules, and never "opened" as a driver.

A driver is a STREAMS module that expects 0 or CLONEOPEN to be set in the sflags parameter of its open routine. It is always "opened" as a driver, and never "pushed" onto other modules. It may be I_LINKed or I_PLINKed below other drivers

A port driver is a STREAMS module that acts exactly like the driver described above, but it is "registered" with Open Transport in Open Transport's port registry (see the Open Tpt Client Note for information on the port registry). This allows several things to happen. First, it is visible in the port registry for clients to browse. Second, multiple instances of the module are possible. You can register a single module as several different ports, which allows a single driver to support multiple hardware devices (one for each port registered). For PowerPC port drivers implemented using CFM, this allows a separate static data instance for each hardware device, which is very convenient. It also gives each instance of the driver a unique major device number (See the section on Port Drivers for more information).

Dynamic Loading

Open Transport supports two methods of dynamic loading for STREAMS modules. A STREAMS module may be written as an Apple Shared Library Manager (ASLM) shared library, or as a CFM shared library. For 68K STREAMS modules, you must use ASLM. For PowerPC, CFM is the preferred mechanism, but ASLM may also be used (Note: ASLM will not be available for module loading in Mac OS 8).

Whenever a STREAMS module or driver is described as exporting a function in this document, it means to export the function using the named export method of the appropriate DLL. For ASLM, this means using the "extern" keyword in front of the name of a function in the export file. For CFM, this means using the -export switch to export the functions when linking a shared library.

For hardware STREAMS drivers that are written on Power Macintosh systems with the Native driver architecture, the driver must be written to conform with that architecture. This means that the hardware driver must be written using CFM only. Open Transport will get all of the information it needs from the System Registry in this case.

Building Modules and Drivers

This section defines the actual steps necessary to build STREAMS modules and drivers for Open Transport. It will describe any code that needs to be supplied to include the module or driver into the Open Transport system, but it will not talk about how to write the operational portion of a STREAMS module or driver.

WARNING: When building PowerPC version of modules, NEVER set any compiler options that indicate that structure should be aligned in any way but PowerPC native. Open Transport includes #pragmas to align all structures that are shared between 68K and PowerPC code to 68K alignment. However, any structures that are not shared between 68K and PowerPC code (which includes most of the module-level headers) are aligned to PowerPC standard alignment. If you override this alignment, your code will not be looking at the fields you think you are. Many of Open Transport's data structures are common with the Unix world, where a 16-bit field is followed by a 32-bit field, causing non-optimal alignment on the 32-bit field unless PowerPC standard alignment is used.

Module exports

In order to use your STREAMS module or driver, Open Transport needs to be able to locate information about your module. This section will describe those exports.

GetOTInstallInfo

When a service requires the use of your driver, Open Transport will automatically load it and install it into the STREAMS module tables. In order to do this, your module must export a function named either `GetOTInstallInfo` or `GetOTxxxxxInstallInfo` (where `xxxxx` is the name of the module or driver).

```
install_info* GetOTInstallInfo(void);
```

NOTE: The reason for having two different possible names for most of the Open Transport interface functions is so that you can put more than one STREAMS module in a single shared library. This is often necessary when a STREAMS module is both a driver and a module.

This function returns the installation information that Open Transport needs to install the driver into the STREAMS tables.

```
struct install_info
{
    struct streamtab* install_str;
    UInt32           install_flags;
    UInt32           install_sqlvl;
    char*            install_buddy;
    void*            ref_load;
    UInt32           ref_count;
};
```

install_str	This is a pointer to the driver's <code>streamtab</code> structure.	
install_flags	This contains bits to inform Open Transport about some of your capabilities and needs (see below for the bit definitions).	
install_sqlvl	This flag is set to the type of reentrancy your driver can handle. <code>mps_become_writer</code> can be used to insure that only 1 instance of your module is running for those functions that require exclusivity. Legal values are:	
	SQLVL_QUEUE	Each stream containing your module can be entered once from the upper queue and once from the lower queue at the same time. Using <code>mps_become_writer</code> is very time-consuming with this sync level.
	SQLVL_QUEUEPAIR	Each stream containing your module can be entered from either the upper queue or the lower queue, but not at the same time. Using <code>mps_become_writer</code> is very time-consuming with this sync level.
	SQLVL_MODULE	Your module can only be entered once, no matter which instance of the module is entered. <code>mps_become_writer</code> is not needed with this sync level.
	SQLVL_GLOBAL	Only 1 STREAMS module or driver can be entered at any one time. Between all modules that use <code>SQLVL_GLOBAL</code> , only 1 will be entered at a time. <code>mps_become_writer</code> is not needed with this sync level.
	SQLVL_SPLITMODULE	Your module can be entered once from an upper queue and once from a lower queue. With this sync level, the <code>mps_become_writer</code> function is relatively cheap, and this is the recommended sync level for network and link-layer drivers. This sync level is available only in Open Transport 1.5 or later.

install_buddy	This field is set to the name of a module/driver that needs to be synchronized with this driver. For modules with SQLVL_MODULE synchronization, it means that both modules are considered a single module from the point of view of the synchronization. For other synchronization, it means that if you call <code>mps_become_writer</code> , you will be synchronized between both sets of modules. You can make more than 2 modules be install buddies, by creating a "ring" of buddies (i.e. A has B as a buddy, B has C as a buddy, and C has A as a buddy). NEVER set up install_buddies that are not in a "ring"-type configuration, or Open Transport may go into an infinite loop trying to find your "buddies". For drivers that are registered ports (see the section "Working with Registered Ports"), you must use the real module name of the port, not the name it is registered with. In addition, making a registered port driver a writer buddy means that all instances of the registered port driver are synchronized together, which can have a detrimental performance impact if the driver synchronization is set to SQLVL_MODULE.
ref_load	This field is used by Open Transport to keep a load reference to the module. It should be initialized to zero.
ref_count	This field is used by Open Transport to keep track of when a driver is first loaded, and when it is last unloaded. It should be initialized to zero.

The `install_flags` contain several bits that must be set in order for Open Transport to properly use your module:

```
//
//      Flags used in the install_flags field
//
enum
{
    kOTModIsDriver          = 0x00000001,
    kOTModIsModule          = 0x00000002,

    kOTModUpperIsTPI        = 0x00001000,
    kOTModUpperIsDLPI       = 0x00002000,
    kOTModLowerIsTPI        = 0x00004000,
    kOTModLowerIsDLPI       = 0x00008000,
    //
    // This flag says you don't want per-context globals
    //
    kOTModGlobalContext      = 0x00800000,
    //
    // These flags are only valid if kOTModIsDriver is set.
    //
    kOTModUsesInterrupts     = 0x08000000,
    kOTModIsComplexDriver    = 0x20000000,
    //
    // These flags are only valid if kOTModIsModule is set.
    //
    kOTModIsFilter           = 0x40000000
};
```

These flags have the following meaning:

kOTModIsDriver	Set this bit if your STREAMS module is a driver (i.e. expects <code>CLONEOPEN</code> or 0 in the <code>sflags</code> parameter of the <code>open</code> routine). This bit or the <code>kOTModIsModule</code> bit MUST be set for your STREAMS module to be valid.
kOTModIsModule	Set this bit if your STREAMS module is a module (i.e. expects <code>MODOPEN</code> in the <code>sflags</code> parameter of the <code>open</code> routine). This bit or the <code>kOTModIsDriver</code> bit MUST be set for your STREAMS module to be valid.
kOTModUpperIsTPI	Set this bit if your STREAMS module understands TPI commands on its upper queue.
kOTModUpperIsDLPI	Set this bit if your STREAMS module understands DLPI commands on its upper queue.
kOTModLowerIsTPI	Set this bit if your STREAMS module understands TPI commands on its lower queue.
kOTModLowerIsDLPI	Set this bit if your STREAMS module understands DLPI commands on its lower queue.
kOTModGlobalContext	Set this bit if your driver requires a single static data space for all instances of the driver. For CFM drivers, Open Transport normally creates a new static data area for each hardware device that the driver handles. However, ASLM does not support this, so drivers using ASLM are not easily ported to CFM unless this flag is set. This bit is only valid for STREAMS drivers, not modules (modules never get a second static data instance).
kOTModUsesInterrupts	Set this bit if your driver fields hardware interrupts. This bit is only valid for STREAMS drivers, not modules. Only set this bit if you are a driver for a PCI module. If you are writing a STREAMS driver

kOTModIsComplexDriver	Set this bit if your driver has "complex" plumbing needs and requires a "Configurator" (see the Open Tpt Protocol Dev. Note and sample code for more information). Basically, this bit says that just "opening" the driver is not enough for operation. This bit is only valid for STREAMS drivers, not modules.
kOTModIsFilter	Set this bit if your STREAMS module is a "filter"-type module (i.e. it does not affect the operation of modules above or below it, so it is in effect invisible to them). This bit is only valid for STREAMS modules, not drivers.

If you have a STREAMS module that is both a driver and a module, you must export two `GetOTInstallInfo` functions with two different names, using two different streamtabs. This is most commonly done by appending an "m" to the end of the driver name for the module version (e.g. "ip" is the driver version of the ip protocol, and "ipm" is the module version of the ip protocol).

InitStreamModule

Whenever Open Transport loads your module or driver for the first time, Open Transport will call an optional initialization function exported by the module.

NOTE: Instantiating a module for the first time means that the module is currently not loaded by Open Transport. This can include the module having been used earlier, and then unloaded because it was no longer in use.

For drivers that support multiple hardware devices, Open Transport treats each hardware device that the driver supports as though there were multiple drivers. For instance, if a driver supports both "enet1" and "enet2" devices, the first time "enet1" is used, `InitStreamModule` will be called for "enet1". If someone subsequently uses the "enet2" device, `InitStreamModule` will be called again for "enet2".

This function must be named either `InitStreamModule` or `InitxxxxStreamModule` (where `xxxx` is the name of the module or driver).

```
Boolean InitStreamModule(void* systemDependent);
```

If the `InitStreamModule` returns false to Open Transport, then the loading of the module will be aborted and a `kENXIOErr` error will be returned to the client. Otherwise, the module will be loaded, and installed into a stream.

The `systemDependent` parameter will be a pointer to a value that depends on the type of driver/module. For Native drivers that are in the System Registry, this will be a pointer to a `PCIInfo` structure defined in `OpenTptPCISupport.h`. For normal drivers and modules, this parameter will be `NULL`. For all registered port drivers it will be the value registered as the `contextPtr` when the port was registered.

If your device supports changing power usage, the `InitStreamModule` function should set the power level for normal operation.

TerminateStreamModule

Whenever Open Transport removes the last instance of a module or driver from the system, Open Transport will call an optional termination function exported by the module. This function must be named either `TerminateStreamModule` or `TerminatexxxxxStreamModule` (where `xxxxx` is the name of the module or driver).

```
void TerminateStreamModule(void);
```

If your device supports changing power usage, the `TerminateStreamModule` function should set the power level to low power or no power, as appropriate.

Of course, modules and drivers may also use the initialization and termination features of their DLL technology (both CFM and ASLM allow initialization and termination routines). However, Open Transport often loads a module or driver just to obtain information about the module. In this case, `InitStreamModule` and `TerminateStreamModule` are not called.

All memory allocations that do not use the Open Transport allocation routines (`OTAllocMem` and `OTFreeMem`) or any interrupt-safe allocators supplied by the interrupt sub-system must be done from within your initialization and termination routines (i.e. `NewPtr`, `NewHandle`, `DisposePtr`, `DisposeHandle`, `PoolAllocateResident`, and `PoolDeallocate` may only be called from your initialization and termination routines).

Once your module has been loaded, all communication with it will be through STREAMS messages, and the entry points in the `streamtab`.

Building Modules with ASLM

In order to build your modules with ASLM, you need to create a .exp file and invoke the BuildSharedLibrary script from the ASLM SDK.

68K ASLM Modules

For 68K code, your module may be built with any compiler that uses 4 byte integers and has "C" stack-based calling conventions (Of course, these restrictions only apply to those functions you are exporting either through the streamtab or to Open Transport). After linking your module into a single object file, a line similar to the following needs to be executed to create the shared library:

```
BuildSharedLibrary "MyModuleLib.o" 0
    -lib "MyModule.RSC" 0
    -symdir ":" -symfile OTLib$MyModule -sym on 0
    -clientFile "MyModule.cl.o" 0
    -exp "MyModule.exp" 0
    -restype cd02 -resid 02 0
    -i "{CIncludes}" -i "{OTIncludes}" 0
    -obj "MyModule" 0
    "{OTLibs}OpenTptModule.o" 0
    "{OTLibs}LibraryManager.o" 0
    "{Libraries}"Interface.o 0
    "{Libraries}"MacRunTime.o
```

This command will create a file called MyModule.RSC. ASLM places shared libraries into multiple resources.

- | | |
|-------------|---|
| -resid | ASLM creates several resources for each shared library it creates. This switch gives a resource ID to those resources. By using different resource IDs for different shared libraries, you can combine multiple ASLM shared libraries into a single shared library file. |
| -restype | This switch specifies the resource type that the code will be placed into. By using different resource types for different shared libraries, you can combine multiple ASLM shared libraries into a single library file. DO NOT USE 'code' or 'CODE' as the resource type. |
| -lib | This switch specifies where the output resource file is to go. |
| -clientFile | This switch specifies the client file. You must supply this parameter, but the client file is unused unless you are exporting other entry points into your module for your own use (see the ASLM documentation for more details) |
| -sym on | This optional switch specifies that symbols are desired. |
| -symDir | This optional switch specifies where a symbol file should be placed |

- symfile This optional switch specifies the name of the symbol file without the .SYM extension.
- exp This switch locates the export file that tells ASLM the particulars about your shared library.
- obj This switch tells ASLM where to put the intermediate object files it creates, and what file name root to use.

The first file on the command line is the object file to make into a shared library. All other files on the command line are libraries to link with.

PPC ASLM Modules

For PowerPC ASLM libraries, your module can be built with any compiler that outputs standard XCOFF files. After linking your module into a single object file, a line similar to the following needs to be executed to create the shared library:

```
BuildSharedLibrary "MyModuleLib.o" @
    -powerpc -xcoffSymfile -sym on @
    -symdir ":" -symfile OTLib$MyModule @
    -lib "MyModule.RSC" @
    -clientFile "MyModule.cl.o" @
    -exp "MyModule.exp" @
    -restype cd02 -resid 02 @
    -i {StdCIncludes} @
    -obj "MyModule" @
    "{OTLibs}OpenTptModuleLib" @
    "{OTLibs}LibraryManagerPPC.o" @
    "{Libraries}"RunTime.o
```

The switches mean the same thing as in the 68K example, except:

- powerpc Tells the tool that we are building a PowerPC shared library.
- sym on This optional switch specifies that symbols are desired.
- symDir This optional switch specifies where a symbol file should be placed
- symfile This optional switch specifies the name of the symbol file without the .xSYM (or .xc) extension.
- xcoffSymFile This switch specifies that an xcoff file is desired for symbols instead of a .xSYM file. ".xc" will be used as the extension for the file in this case.

The first file on the command line is the XCOFF file to make into a shared library. All other files on the command line are libraries to link with. Both XCOFF and PEF libraries are supported.

.exp files for ASLM Modules

An ASLM export file describes how the library is to be created. Below is a sample of what should be placed in the export file for your module. If you want more details, please refer to the ASLM documentation.

```
/*
 * Include OpenTptModule.h to get pertinent defines
 */
#include <OpenTptModule.h>

#define kMyModuleName      "MyModule"    /* "StreamTab" name of module */
#define kMyVersion         "1.0"         /* Version number of module */

/*
 * Just use this verbatim */
Library
{
    /*
     * Typically we load modules this way so there are no surprises.
     * This insures that the module, and all other shared libraries
     * that it depends on, are in memory when we load. It insures
     * that the module will not be loaded on a 68000 machine, and that
     * if it is a 68K module, it will not be loaded when running emulated
     * on a Power Macintosh.
     */
    flags = noSegUnload, forceDeps, !mc68000, !emulated;
    /*
     * Create the name of the library. We use this format,
     * but you can use anything that you want
     */
    id = kOTLibraryPrefix kMyModuleName "," kMyVersion;
    /*
     * Set up the version number of the library
     */
    version = kMyVersion;
    /*
     * Always use memory = local.
     */
    memory = local;
};
```

```

/*
 * Change the name of this function set by substituting the real name
 * of your module for "MyModule"
 */
FunctionSet Module_MyModule
{
    /*
     * The ID for your module MUST look like the following:
     */
    id          = kOTModulePrefix kMyModuleName "," kMyVersion;
    /*
     * The function set for your module can export many functions.
     * However, you must export GetOTInstallInfo
     * (or GetOTxxxxxInstallInfo) by name.
     * If you want an initialization function, it must also be
     * exported by the name InitStreamModule (or InitxxxxStreamModule),
     * and will be called the first time your module is loaded.
     *
     * If you also need a termination function, export a function
     * called TerminateStreamModule (or TerminatxxxxStreamModule),
     * and it will be called before Open Transport unloads your module.
     *
     * NOTE: Don't make these "static" functions in your file, or
     *       ASLM can't export them. For C++ clients, make sure
     *       that they are declared extern "C".
     */
    exports =    extern GetOTMyModuleInstallInfo,
                extern InitStreamModule,
                extern TerminateStreamModule;
};

```

If you are exporting more than one module from an ASLM library, create additional FunctionSet declarations (making sure the name following the work FunctionSet is unique) for each module that is exported. In addition, you need to have named your exports with the "xxxxx" version of the names, since you have to link your modules together, and each entry point needs a unique name for each module.

Building Modules with CFM

Building modules with CFM is simple. Create a CFM shared library and export the functions you need to support. The ONLY requirement is that the name of your CFM library MUST be "OTModl\$MyModule", where "MyModule" is replaced by the name of your module that is in the streamtab (it's in the st_rdinit->qinfo->mi_idname field).

If you export more than one module from the shared library, you must create a 'cfrg' resource that gives your library more than one name for the same library:

```
resource 'cfrg' (0) {
    { /* array memberArray: 2 elements */
        /* [1] */
        kPowerPC,
        kFullLib,
        kNoVersionNum,
        kNoVersionNum,
        kDefaultStackSize,
        kNoAppSubFolder,
        kIsLib,
        kOnDiskFlat,
        kZeroOffset,
        kWholeFork,
        "OTModl$MyModule",
        /* [2] */
        kPowerPC,
        kFullLib,
        kNoVersionNum,
        kNoVersionNum,
        kDefaultStackSize,
        kNoAppSubFolder,
        kIsLib,
        kOnDiskFlat,
        kZeroOffset,
        kWholeFork,
        "OTModl$MyDriver",
    }
};
```

In addition, when exporting multiple modules from the same library, they will all shared the same static data instance. However, registered port drivers will have separate data instances from all of the other modules, even if the kOTModGlobalContext flag is set in the install_info.

Working with Port Drivers

Open Transport maintains a registry of all port drivers in the system. This registry is accessed through a number of functions, including `OTGetIndexedPort`, `OTFindPort`, and `OTFindPortByRef`. If your driver is a hardware driver, it should be registered with Open Transport. If it is not registered with Open Transport, there are several issues that will arise. The first is that your driver will not know which hardware device it is supposed to be controlling. Even if your driver only controls a single known hardware device, if it is not in the port registry, it will not be visible to Control Panels and configuration applications for use by the various protocols in the system, including AppleTalk and TCP/IP.

As a registered port driver, your driver will be able to use the `OTFindPortByDev` API to discover what hardware device it is supposed to control when it is instantiated. In addition, depending on the `contextPtr` that was stored when the driver was registered, additional information may be available to the driver.

Open Transport treats each entry in the port registry as though it were a unique and separate streams driver. Unless the `kOTModGlobalContext` bit is set or ASLM is being used, an instantiation of a port is given a separate static data area from all other instantiations of the same stream driver in use by other ports.

While this feature is valuable for hardware device driver writers, it is potentially useful for protocol writers as well. For instance, the Apple implementation of the DDP protocol registers a private "pseudo-port" in the port registry for each different driver that DDP is instantiated on. This gives a separate instantiation of DDP over each driver, allowing DDP to multihome easily. These pseudo-ports are registered and unregistered on an as-needed basis, and have the `kOTPortIsPrivate` bit set so that Control Panels, etc. are not tempted to display them as legitimate ports.

AppleTalk creates the stream by registering the port, and then opening the stream using the port name returned in the `fPortName` field of the `OTPortRecord`. It then links the appropriate driver underneath the DDP stream. From that point on, until the port is unregistered, opening a DDP stream using that port name will result in opening a clone of this DDP stream (as opposed to DDP over some other hardware device).

APIs for Port Drivers

Open Transport has several APIs for dealing with port drivers. They are described in the following sections.

OTRegisterPort

Open Transport provides a function, `OTRegisterPort`, to allow port drivers to be registered with Open Transport. This function makes the port visible to clients through the various port APIs defined in `OpenTransport.h`. It also allows a STREAMS driver to be instantiated multiple times for different hardware devices.

```
enum
{
    kMaxModuleNameLength      = 31,
    kMaxModuleNameSize        = kMaxModuleNameLength + 1,

    kMaxProviderNameLength    = kMaxModuleNameLength + 4,
    kMaxProviderNameSize      = kMaxProviderNameLength + 1,

    kMaxSlotIDLength          = 7,
    kMaxSlotIDSize            = 8,

    kMaxResourceInfoLength    = 31,
    kMaxResourceInfoSize      = 32
};

/*
 * Values for the fInfoFlags field of OTPortRecord
 */
enum
{
    kOTPortIsDLPI              = 0x00000001,
    kOTPortIsTPI               = 0x00000002,
    kOTPortCanYield            = 0x00000004,
    kOTPortCanArbitrate        = 0x00000008,
    kOTPortIsTransitory        = 0x00000010,
    kOTPortAutoConnects        = 0x00000020,
    kOTPortIsSystemRegistered  = 0x00004000,
    kOTPortIsPrivate           = 0x00008000,
    kOTPortIsAlias             = 0x80000000
};

struct OTPortRecord
{
    OTPortRef          fRef;
    UInt32             fPortFlags;
    UInt32             fInfoFlags;
    UInt32             fCapabilities;
    size_t             fNumChildPorts;
    OTPortRef*         fChildPorts;
    char               fPortName[kMaxProviderNameSize];
    char               fModuleName[kMaxModuleNameSize];
    char               fSlotID[kMaxSlotIDSize];
    char               fResourceInfo[kMaxResourceInfoSize];
    char               fReserved[164];
};

OSStatus OTRegisterPort(OTPortRecord* portInfo, void* contextPtr);
```

A port is registered by filling out the `OTPortRecord`, and passing it, along with a `contextPtr` to the `OTRegisterPort` function. If you need persistent memory for the `contextPtr`, use the `OTAllocPortMem` and `OTFreePortMem` functions to allocate and free the memory.

The `contextPtr` that is registered is the value that is passed to the `InitStreamModule` function when the port driver is loaded. It may also be retrieved by the driver by making the call `OTFindPortByDev` and retrieving the value from the `TPortRecord`.

<code>fRef</code>	This field must be filled in with the <code>OTPortRef</code> of the port. <code>OTPortRefs</code> must be unique in the system. If another port is already registered with the same <code>OTPortRef</code> , Open Transport will assume that this port is an alias for the real port. This is especially convenient for registering "default" ports (e.g., "ltlkB", and "ltlk" are both registered to the same <code>OTPortRef</code> on most machines, but on PowerBooks, quite often "ltlk" is registered to the "ltlkA" <code>OTPortRef</code> , since there is no PortB LocalTalk. This allows clients to use "ltlk" and get whatever the default LocalTalk port is). When registering "pseudo-ports", it is permissible to use <code>OTCreatePortRef(0, kOTPseudoDevice, 0, 0)</code> , and Open Transport will assign a unique <code>OTPortRef</code> to the device (the value will be in this same field when the <code>OTRegisterPort</code> function returns).
<code>fPortFlags</code>	Unused - set to 0

InfoFlags Set the appropriate bits.

kOTPortIsDLPI

Set this bit if the upper interface to your driver is DLPI.

kOTPortIsTPI

Set this bit if the upper interface to your driver is TPI.

kOTPortCanYield

Set this bit if your driver supports the `I_YIELDPORT` and `I_PROVIDERTYPE` IOCTL messages (if you are a serial port or some other device type that cannot demultiplex incoming data, you should support these messages and set this bit. See the section on IOCTL support for a description of the IOCTL messages).

kOTPortCanArbitrate

Set this bit if your driver makes the `OTRequestHardwareAccess` and `OTReleaseHardwareAccess` APIs (available only in versions 1.5 and later). If you are a serial port or some other device type that cannot demultiplex incoming data, you should support these APIs. Open Transport will automatically request and release the hardware access for you if this bit is not set (Ethernet drivers, for instance). You should normally call `OTRequestHardwareAccess` at the point in time where your driver is going to assert ownership of the hardware (for Serial drivers, this is normally at Bind (`qlen > 0`) or Connect time.

kOTPortIsTransitory

Set this bit if your driver has offline/online status (PPP over a Modem, for instance). Drivers that can do this should make the calls:

```
OTChangePortState(myRef, kOTPortOffline, myReason)
OTChangePortState(myRef, kOTPortOnline, myReason)
```

to inform Open Transport of the change of state.

kOTPortAutoConnects

Set this bit if your driver automatically goes offline and online “on demand”. ISDN drivers quite often do this.

kOTPortIsSystemRegistered

This should only be set by Open Transport. It is set if Open Transport registered the port from the information in the Name Registry.

kOTPortIsPrivate

This bit should be set if you are registering “pseudo-ports” for your own private purposes. It is informational, and tells Control Panels and other control programs that they should not display the port. For instance, AppleTalk registers a DDP “pseudo-port” for every device that AppleTalk is using.

kOTPortIsAlias

This bit is used internally by Open Transport, and should not be set when making the OTRegisterPort call. It tells Open Transport that this port is an alias record for another port with the same OTPortRef. This is how Open Transport allows “serial” to be the name of the “default” serial port - “serial” is registered as the name of a port with the same OTPortRef as “serialA” or “serialB”, depending on which port is “default”.

fCapabilities

fCapabilities is a bitmap that is defined on a device-by-device basis. Typically, they define framing options for a protocol or device. If you do not use zero for the framing flags, then you **MUST** support the I_OTSetFramingType IOCTL, which will pass you a 32-bit value with a single bit set, indicating the framing option desired. If this IOCTL call is made, then your DLPI driver should fill in the dl_mac_type field of a dl_info_ack_t with a value consistent with the requested framing type.

For instance, ethernet supports four framing options:

kOTFramingEthernet	= 0x01
kOTFramingEthernetIPX	= 0x02
kOTFraming8023	= 0x04
kOTFraming8022	= 0x08

Most Ethernet drivers support all but kOTFraming8023 (typically, Ethernet drivers support kOTFraming8022, which indicates that they can handle full SAP/SNAP demultiplexing, whereas kOTFraming8023 indicates that they will deliver all 802.3 frames to a single client). If a client requests kOTFraming8022 using the IOCTL, then DL_CSMACD should be returned in the dl_mac_type field. If a client requests kOTFramingEthernet or kOTFramingEthernetIPX, then DL_ETHER should be returned instead. **NOTE:** Currently, Open Transport does not support the kOTFraming8023 framing type, so Ethernet drivers must handle full SAP/SNAP demultiplexing and Test/XID frames in order to work properly with AppleTalk and TCP/IP.

fNumChildPorts

This field contains the number of "child" ports that this port uses. See the description of the fChildPorts field for more information.

fChildPorts

This field is a pointer to an array of OTPortRefs that are "child" ports for the port being registered. "Child" ports occur when one port driver depends on another. Typically, a port has 0 or 1 child port. Some examples of "child" ports would be: 1) a modem device almost always has a "child" port that is a serial device; 2) A pseudo-port (like ddp0, ddp1, etc..) almost always has a "child" port that is either another pseudo-port, or is a real hardware device.

fPortName

This field contains the name of the port. Typically, just leave the first byte of this field set to '\0', and Open Transport will make up a unique port name. However, feel free to supply the name you want your port to be known by (but if another port is already registered with that name, an error will occur and your port will not be registered). This name is the name that is used to open endpoints and streams with. If the OTRegisterPort call succeeds, this field will be filled in with the port name that was assigned.

fModuleName

This field contains the name of the stream module (Of course, stream modules which use OTRegisterPort are really stream drivers). This should be the "MyModule" part of the shared library ID "OTModl\$MyModule", which should also be the name of the module stored in the streamtab (st_rdinit->qinfo->mi_idname field).

fSlotID

This field is a 0-terminated string that contains a slot identifier. If this string is a null-string, Control Panels will use the information from the OTPortRef to attempt to create a slot identifier string.

fResourceInfo

This field is a 0-terminated string that contains an identifier that will allow Open Transport to access auxiliary information about your driver (Open Transport creates shared library ids from this string to be able to find these extra shared libraries). This string should either be unique to your driver, or should be set to a null string. See the section on Port Driver Configuration Info for more detail.

fReserved

This field should be set to all zeros.

Note: OTRegisterPort is available as both a kernel and a client API. If the contextPtr is actually a pointer to memory, under Mac OS 8 it must have been allocated with PoolAllocateGlobal if you are using the client API and wish the stream module to be able to address the memory. Also, OTRegisterPort copies all of the information in the OTPortRecord, so the OTPortRecord may be allocated on the stack

OTUnregisterPort

OTUnregisterPort allows a client to unregister a port by name. Unregistering by name is necessary because it is allowed to register an OTPortRef by several names, creating alias records. The "contextPtrPtr" parameter will return the value of the contextPtr that was used when registering. Unregistering a port is normally a permanent affair. If it is possible that the port will be back on-line (as with PCMCIA Ethernet cards), it is better to use the OTChangePortState API, since while the port was unregistered, another port may have grabbed the name that was being used.

```
OSStatus OTUnregisterPort(const char* portName, void** contextPtrPtr);
```

Note: OTUnregisterPort is available as both a kernel and a client API.

OTChangePortState

OTChangePortState allows you to change the state of a port without unregistering it. You can disable a port, which causes all streams using the port to be closed down, and causes kENXIOErr errors for anyone attempting to open the port while it is disabled. You can enable the port, which will cause the port to be reenabled. For devices that come and go (hot-docking and PCMCIA cards are examples), it is much better to use OTChangePortState than to Register and Unregister the port, since there is no chance of the port's name being taken in this case.

Typically, the "why" parameter is kOTPortHasDiedErr, kOTPortWasEjectedErr, kOTPortLostConnection, or kOTUserRequestedErr. All Open Transport clients are notified when this call is made, and the "why" parameter is one of the pieces of information given to the clients.

```
enum
{
    kOTPortDisabled = (OTEventCode)0x25000001,
    kOTPortEnabled  = (OTEventCode)0x25000002,
    kOTPortOffline   = (OTEventCode)0x25000003, /* OT 1.5 and later */
    kOTPortOnline    = (OTEventCode)0x25000004, /* OT 1.5 and later */
};

OSStatus OTChangePortState(OTPortRef ref, OTEventCode theChange, OSStatus why);
```


OTGetIndexedPort

OTGetIndexedPort returns the TPortRecord* corresponding to the index parameter (index = 0.....n). A NULL is returned if the index is too high. This API is only valid in the kernel (There is a separate and different API for the client). NOTE: This API returns the actual port record used by OpenTransport. Do NOT change any information in the TPortRecord.

```
TPortRecord* OTGetIndexedPort(size_t index);
```

OTFindPort

OTFindPort returns the TPortRecord* that has the requested portName. This API is only valid in the kernel (There is a separate and different API for the client). NOTE: This API returns the actual port record used by OpenTransport. Do NOT change any information in the TPortRecord.

```
TPortRecord* OTFindPort(const char* portName);
```

OTFindPortByRef

OTFindPortByRef returns the TPortRecord* corresponding to the specified OTPortRef. This API is only valid in the kernel (There is a separate and different API for the client). NOTE: This API returns the actual port record used by OpenTransport. Do NOT change any information in the TPortRecord.

```
TPortRecord* OTFindPortByRef(OTPortRef)
```

OTFindPortByDev

OTFindPortByDev returns the TPortRecord* corresponding to the dev_t specified. Only the major number of the dev_t is used to find the port. This API is only valid in the kernel. NOTE: This API returns the actual port record used by OpenTransport. Do NOT change any information in the TPortRecord.

```
TPortRecord* OTFindPortByDev(dev_t dev)
```

This function can be used in your module's open routine to obtain the `TPortRecord` related to your port. From this record, you can retrieve the "contextPtr" that was stored when the port was registered, as well as other useful information:

```
struct TPortRecord
{
    OTLink          fLink;
    char*           fPortName;
    char*           fModuleName;
    char*           fResourceInfo;
    char*           fSlotID;
    struct TPortRecord* fAlias;
    size_t          fNumChildren;
    OTPortRef*      fChildPorts;
    UInt32          fPortFlags;
    UInt32          fInfoFlags;
    UInt32          fCapabilities;
    OTPortRef       fRef;
    struct streamtab* fStreamtab;
    void*           fContext;
    void*           fExtra;
};
```

This function returns the actual `TPortRecord` used by the system, so don't modify it. The only really useful fields are the `fRef` field, which contains the `OTPortRef` for your module, and the `fContext` field, which contains the "cookie" that was saved for the driver when the port was registered. For systems using the Native driver architecture, this "cookie" will be a pointer to a structure whose first element is the `RegEntryID` for the driver (i.e. the "cookie" can be interpreted as a `RegEntryIDPtr`).

This function is most useful to drivers that handle multiple hardware devices. By using the `fRef` or `fContext` value, the driver can determine which hardware device the `Open` call is referring to. **WARNING:** This function may not be called at interrupt time.

Do not be confused by the similarity between the `TPortRecord` and the `OTPortRecord`. An `OTPortRecord` is a copy of the `TPortRecord` specially formatted for client needs. The `TPortRecord` is the structure that Open Transport actually keeps in the port registry to keep track of information on registered ports. Only stream modules and kernel infrastructure have access to the actual `TPortRecord`.

Registering Port Drivers

Since it is necessary to have port drivers registered early in the boot process so that the protocol suites can be up and running at boot time, and there are very few mechanisms on the Macintosh for running early in the boot process, Open Transport provides an automated way to run your code to register your port drivers. These code modules are called "port scanners".

Open Transport provides a few default port scanners. On Macintosh systems, Open Transport will automatically register LocalTalk and serial ports on SCCA and SCCB, as well as any serial ports that are registered in the Communications Toolbox Resource Manager. On Nubus system, it will also automatically register all .ENET, .TOKEN, and .FDDI drivers (Open Transport has a DLPI "shim" to interface to these various drivers). On PCI machines with the System Registry, Open Transport automatically registers all Open Transport drivers that are present in the System Registry (see the "Designing PCI Cards and Driver for Power Macintosh Computers" for more information on this feature).

All other drivers must supply a port scanner to register the driver. The port scanner must currently be built as an ASLM library, because CFM does not provide a mechanism for finding a related group of shared libraries, like scanners.

Your port scanner will export a function called `OTScanPorts` by name from an ASLM function set with an `InterfaceID` of either `kOTPortScannerInterfaceID` or `kOTPseudoPortScannerInterfaceID` (more on these two in a moment). A sample .exp file for the ASLM build is:

```

/*
    Sample .exp file for creating an ASLM
    shared library to export an Open Transport
    port scanner.
*/

#include "OpenTptModule.h"

#define MyScannerName      "myScanner"

Library
{
    /*
     * 1) Segments won't be loaded or unloaded.
     * 2) We want any libraries we depend on to be
     *     forced to load prior to us loading.
     * 3) We don't run on 68000 processors
     */
    flags = noSegUnload, forceDeps, !mc68000;
    /*
     * This id can be anything you want
     */
    id = kOTLibraryPrefix MyScannerName;
    /*
     * Put the appropriate version number here.
     */
    version = 1.0;
    memory = client;
};

FunctionSet MyScannerFunctionSet
{
    /*
     * You must use this InterfaceID
     */
    InterfaceID = kOTPortScannerInterfaceID;
    /*
     * Your ID can be anything as long as it starts
     * with kOTPortScannerPrefix.
     */
    id          = kOTPortScannerPrefix MyScannerName ",1.0";
    /*
     * You can export other functions, but you must
     * have "extern OTScanPorts" somewhere in the list.
     */
    exports      = extern OTScanPorts;
};

```

It is also possible to combine your driver and port scanner into a single ASLM shared library. ASLM allows multiple FunctionSets to be exported from the same library, so you can export your driver entry points as one function set, and your scanner as another.

The `OTScanPorts` function is a C function with the following prototype:

```
void OTScanPorts(void)
```

When the `OTScanPorts` function is called, the scanner should search for the hardware and drivers that it is responsible for, and call the `OTRegisterPort` to register each hardware device and associate it with a driver (see the description for `OTRegisterPort` earlier in this document for more information).

For some devices, it is necessary to have all of the hardware ports available in the port registry before registering themselves. Examples of this are the ATMSNAP (ATM Classical IP) or ATMLANE (ATM LAN emulation) drivers. These drivers are software drivers that talk to ATM hardware drivers. Without these two drivers, AppleTalk or TCP/IP would be unable to use ATM. These drivers provide a translation function that allow AppleTalk or TCP/IP to believe that they are operating on a Local Area Network instead of over a point-to-point link. These drivers need to be registered as ports so that they show up in the appropriate Control Panels, and also so that they can have separate instances for each ATM hardware device present.

The problem, of course, is that if they have to register as a driver for each of the hardware devices that they support, some of the devices might be missed, since the order of running port scanners is unspecified. To alleviate this problem, you can use the `kOTPpseudoPortScannerInterfaceID` constant. All port scanners with this interfaceID are guaranteed to run after all the port scanners with the `kOTPortScannerInterfaceID` have run.

Port Driver Configuration Info

Once your port driver is registered with the system, you may want control panels and other programs which display port information to be able to get port information for your ports, such as icons and name strings.

When the driver was registered, an `fResourceInfo` field was supplied. Open Transport will create a library name from this field by prepending the constant `kPortConfigLibPrefix` to it (see `OpenTptConfig.h` in the Open Transport Protocol Developer SDK).

NOTE: For developers of PCI drivers, the value from the `DriverOSRuntimeInfo.driverName` field is moved into the `fResourceInfo` field (minus any leading "." in the name). In the original documentation, it said that this field should contain one of the device names found in `OpenTptLinks.h`. This is not required. The field can be 0-length (although if it is 0-length, your driver will not be compatible with 1.0.x version of Open Transport). If you want to be compatible with Open Transport 1.0.x, but don't have a configuration library for your driver, just continue putting the generic device name in this field.

Clients who want to get configuration information from your driver can use the API calls in `OpenTptConfig.h`:

```
void OTGetUserPortNameFromPortRef(OTPortRef ref, Str255 friendlyName)
```

This function calls the configuration library of the port specified, and returns a name in the `friendlyName` field. If the `friendlyName` field returns a 0-length string, then the port does not provide the functionality.

```
Boolean OTGetPortIconFromPortRef(OTPortRef, OTResourceLocator* iconLocation)
```

This function calls the configuration library of the port specified, and returns a true if the port supplied an `iconLocation`. If the function returns true, `iconLocation` can be used to look for the usual icon-type resources in the specified `FSSpec` with the specified resource ID.

```
typedef struct
{
    FSSpec fFile
    UInt16 fResID
} OTResourceLocator;
```

In order for your driver to benefit from these API calls, you must export the function the following functions from a shared library (either ASLM or CFM - Open Transport will find either one):

```
void OTGetUserPortName(OTPortRecord* port, boolean_p includeSlot, boolean_p include
Port, Str255 name);
```

This function should return a pascal string that is the name of the port. If `includSlot` is true, you should also include information about the slot location. If `includePort` is true, you should also include information about which port if your hardware device supports multiple ports.

```
Boolean OTGetPortIcon(OTPortRecord* port, OTResourceLocator* location)
```

This function should return an `FSSpec` and a resource number in the structure pointed to by `location`. Clients can use this to open the file with the icons and read them. Return false if no icons are available for the specified port.

Of course, you only need to export those functions that you care about. Currently, the Open Transport control panels do not need icons for the port, but that may change in the future.

Module and Driver Operation

Once your module or driver is installed in a stream and opened, it is ready for action. From that point on, the driver will respond to messages according to the interface specification(s) (TPI or DLPI) that it supports.

Drivers have one additional proviso that they must observe. If they are running as a primary interrupt, they must call the `OTEnterInterrupt` function prior to making any Open Transport calls, and must call `OTLeaveInterrupt` prior to exiting their current interrupt level, and after they have made their final call to any Open Transport routines.

```
void OTEnterInterrupt(void);
void OTLeaveInterrupt(void);
```

It is strongly suggested that for timing services and secondary interrupt services that the appropriate Open Transport functions be used, since they will adapt to the underlying system. In addition, the Open Transport secondary interrupt services do not have the restrictions present that some of the equivalent system services have, since any memory allocations needed are handled up front, keeping this function from failing at inconvenient times.

Interrupt-Safe functions

Open Transport provides many STREAMS services for module and driver writers. However, not all of these services may be used at interrupt time.

The following STREAMS functions may be safely called at interrupt time:

<code>allocb</code>	<code>adjmsg</code>	<code>copyb</code>	<code>copymsg</code>	<code>dupb</code>
<code>dupmsg</code>	<code>esballocc</code>	<code>freeb</code>	<code>freemsg</code>	<code>linkb</code>
<code>msgdsize</code>	<code>msgpullup</code>	<code>pullupmsg</code>	<code>putq</code>	<code>rmvb</code>
<code>testb</code>	<code>unlinkb</code>	<code>datamsg</code>	<code>OTHERQ</code>	<code>RD</code>
<code>WR</code>	<code>bzero</code>	<code>bcopy</code>	<code>bcmp</code>	<code>unlinkb</code>
<code>qenable</code>				

Note in particular that the `canput` function and its variants are not allowed to be called at interrupt time. In addition, `putq` may only be called to place an `mblock` on the lower stream queue. The most common use for this is to put incoming packets onto the lower stream queue, and then handle the data in the read service routine. Using the `qenable` function is a convenient way of scheduling yourself time outside of your interrupt. The service procedure of the specified queue will be called back at non-interrupt time, allowing you to process your incoming data. Of course, you can also use `OTScheduleDriverDeferredTask` for this purpose.

The following Open Transport functions may be safely called at interrupt time:

OTCreateDeferredTask	OTDestroyDeferredTask	OTScheduleDriverDeferredTask
OTGetClockTimeInSecs	OTGetTimeStamp	OTSubtractTimeStamps
OTTimeStampInMilliseconds	OTTimeStampInMicroseconds	OTElapsedMilliseconds
OTElapsedMicroseconds	OTAllocMsg	OTAllocReadOnlyMsg
OTAllocMem	OTFreeMem	mi_timer_alloc
mi_timer_free	mi_timer	mi_timer_cancel

In addition, all of the functions described under "Atomic Services" below may be called at interrupt time.

Secondary Interrupt Services

There are three functions associated with Open Transport's secondary interrupt services.

```
typedef void (*OTProcessProcPtr)(void* contextInfo);
```

This typedef defines the deferred task callback function.

```
long OTCreateDeferredTask(OTProcessProcPtr proc, long contextInfo)
```

This function creates a "cookie" (the returned long value) that can be used at a later time to schedule the function "proc". At the time that "proc" is invoked, it will be passed the same `contextInfo` parameter that was passed to the `OTCreateDeferredTask` procedure.

```
void OTScheduleDriverDeferredTask(long dtCookie);
```

This function is used to schedule the deferred procedure corresponding to the `dtCookie` value. It may be called multiple times prior to the deferred procedure actually being executed, but the deferred procedure will only be run once. Once the deferred procedure has run, subsequent calls to `OTScheduleDriverDeferredTask` will cause it to be scheduled to run again.

NOTE: `OTScheduleDriverDeferredTask` was introduced in version 1.1. Prior to version 1.5, it schedules exactly the same as `OTScheduleDeferredTask`. Starting with version 1.5, it will insure that deferred tasks scheduled with `OTScheduleDriverDeferredTask` run before all other deferred tasks, and run in a timely manner.

```
void OTDestroyDeferredTask(long dtCookie);
```

This function is used to destroy any resources associated with the deferred procedure. It should be called when you no longer require the deferred procedure.

Timer Services

Open Transport supplies robust timer services that are synchronized with the STREAMS environment. Timer services are supported by using special STREAMS messages. The function `mi_timer_alloc` is used to create one of these special STREAMS messages:

```
mblk_t* mi_timer_alloc(queue_t* targetQueue, size_t size);
```

Calling this function will create a STREAMS timer message of the requested size, that is targeted to the specified STREAMS queue.

```
void mi_timer(mblk_t* timerMsg, unsigned long milliSeconds);
```

This function will schedule the `timerMsg` (which must be created using `mi_timer_alloc`) to be placed on the target STREAMS queue at the specified future time.

```
void mi_timer_cancel(mblk_t* timerMsg)
```

This function will cancel an outstanding timer message. The `timerMsg` will not be destroyed, but will no longer be delivered to the target queue. It may be rescheduled by using `mi_timer` at a later time.

```
void mi_timer_free(mblk_t* timerMsg)
```

This function cancels and frees the specified timer message (remember, `mi_timer_cancel` does not free the message).

```
Boolean mi_timer_valid(mblk_t* timerMsg)
```

Timer messages enter the target queue as `M_PCSIG` messages. Whenever a queue that can receive a timer message receives an `M_PCSIG` message, it should call `mi_timer_valid`, passing the `M_PCSIG` message as a parameter. If the function returns true, then the timer message is valid and should be processed. If the function returns false, then the timer message was either deleted or canceled. In this case, the correct course of action is to ignore the message (i.e. **DON'T** free it). **WARNING:** This function may not be called at interrupt time, and it may not be called twice on the same timer.

Atomic Services

Open Transport supplies some atomic services to help reduce the need to disable and enable interrupts.

The first set of services allow atomically setting, clearing and testing of a single bit in a byte. The first parameter is always a pointer to a single byte, and the second is always a bit number from 0 to 7. The functions always return the previous value of the bit. Bit zero (0) always corresponds to a mask of 0x01, and bit seven (7) always corresponds to a mask of 0x80.

```
Boolean OTAtomicSetBit(UInt8* theByte, size_t theBitNo)
```

```
Boolean OTAtomicClearBit(UInt8* theByte, size_t theBitNo)
```

```
Boolean OTAtomicTestBit(UInt8* theByte, size_t theBitNo)
```

The second set of services allow atomically adding (or subtracting by using a negative number) from a 32, 16, or 8 bit variable. The return value is the new value of the variable (at least what it was when the atomic add was done). Note that an SInt32 is always used as the first parameter. This is to properly handle different 68K "C" compiler calling conventions. Only the first 8 or 16 bits will be used for the 8 or 16 bit atomic add calls.

```
SInt32 OTAtomicAdd32(SInt32, SInt32* varToBeAddedTo)
```

```
SInt16 OTAtomicAdd16(SInt32, SInt16* varToBeAddedTo)
```

```
SInt8 OTAtomicAdd8(SInt32, SInt8* varToBeAddedTo)
```

The third service is a general compare and swap. It insures that the value at `where` still contains the value `oVal`, and if so, the value `nVal` is substituted. If the compare and swap succeeds, the function returns true. Otherwise false is returned.

```
Boolean OTCompareAndSwapPtr(void* oVal, void* nVal, void** where)
```

```
Boolean OTCompareAndSwap32(UInt32 oVal, UInt32* nVal, UInt32** where)
```

```
Boolean OTCompareAndSwap16(UInt16 oVal, UInt16* nVal, UInt16** where)
```

```
Boolean OTCompareAndSwap8(UInt8 oVal, UInt8* nVal, UInt8** where)
```

The fourth set of services is an atomic LIFO list. `OTLIFOEnqueue` and `OTLIFODequeue` are self-explanatory. `OTLIFOStealList` allows you to remove all of the elements from the LIFO list atomically, so that the elements in the list can be iterated at your leisure by traditional means. `OTLIFOReverseList` is for those of us who find that LIFO lists are next-to-useless in networking. Once the `OTLIFOStealList` function has been executed, the result can be passed to `OTLIFOReverseList` to flip the list into a FIFO configuration. Be aware that `OTLIFOReverseList` is NOT atomic.

```

struct OTLink
{
    void*    fNext;
};

struct OTLIFO
{
    void* fHead;
};

void OTLIFOEnqueue(OTLIFO* list, OTLink* toAdd)

OTLink* OTLIFODequeue(OTLIFO* list)

OTLink* OTLIFOStealList(OTLIFO* list)

OTLink* OTReverseList(OTLink* firstInList)

```

The last set of services is for enqueueing and dequeueing from a LIFO list. It is used internally in the STREAMS implementation, so we exported it so that you can use it if it proves useful. If you look at the OTLIFO implementation, it assumes that the structures being linked have their links pointing at the next link, and so on. Unfortunately, STREAMS messages (`mblk_t` structures) are not linked this way internally (the `b_cont` field does not point to the `b_cont` field of the next message block, but instead points to the actual message block itself). These two functions allow creating a LIFO list where the head pointer of the list points to the actual object, but the "next" pointer in the object is at some arbitrary offset in the object .

```

void* OTEnqueue(void** list, void* newListHead, size_t offsetOfNextPtr);

void* OTDequeue(void** theList, size_t offsetOfNextPtr)

```

Power services

For those devices which can change their power usage, the following assumptions are made:

- 1) The call to `ValidateHardware` will set the device to either low power or power off, as appropriate for the device. This is only applicable to drivers using the Native driver architecture.
- 2) The call to `InitStreamModule` will set the device to the power level appropriate to normal operations.
- 3) The call to `TerminateStreamModule` will set the device to either low power or power off, as appropriate for the device.

In addition, devices which can change their power usage should support the `I_OTSetPowerLevel` IOCTL message.

The following describes the four-byte selectors that can be passed in the IOCTL message, and what the return value should be in the IOCTL ack message:

'psup' Return a value of 1 if the card supports power control, 0 if it does not.

'ptog' Return a value of 1 if the card supports switch between high and low power after initialization, 0 if it does not.

'psta' Return a value of 1 if the card is in high power mode

'pmx5' Returns the card's maximum power consumption in microwatts from the 5 Volt supply while in high-power mode

'pmn5' Returns the cards maximum power consumption in microwatts from the 5 Volt supply while in low power mode.

'pmx3' Returns the card's maximum power consumption in microwatts from the 3.3 Volt supply while in high-power mode

'pmn3' Returns the cards maximum power consumption in microwatts from the 3.3 Volt supply while in low power mode.

'splo' Set the card into low power mode. Return a value of 0 if completed successfully, an OSStatus code if not.

'sphi' Set the card into high power mode. Return a value of 0 if completed successfully, an OSStatus code if not.

Memory allocation functions

Open Transport provides several functions for performing memory allocation in STREAMS modules and drivers. All of these memory allocation functions may safely be called at interrupt time.

OTAllocMem

This function allocates a variable-sized chunk of memory

```
void* OTAllocMem(size_t sizeToAlloc)
```

OTFreeMem

This function frees a chunk of memory that was allocated by OTAllocMem.

```
void OTFreeMem(void* memToFree)
```

allocb

This function allocates a message block of the requested size, with a requested priority (BPRI_LO, BPRI_MED, BPRI_HI). STREAMS currently defines that allocb will ignore the priority.

```
mblk_t* allocb(size_t sizeToAlloc, int pri)
```

freeb

This function frees a single message block.

```
void freeb(mblk_t* msgToFree)
```

freemsg

This function frees the message block passed in as a parameter, as well as all of the message blocks linked to it through the b_cont field.

```
void freemsg(mblk_t* msgToFree)
```

dupb

This function returns a new message block which references the data in the original message block. It DOES NOT copy the data, and it only duplicates the original message block.

```
mblk_t* dupb(mblk_t* msgToDup)
```

dupmsg

This function returns a new message block which references the data in the original message block. It DOES NOT copy the data. It also duplicates all of the data chained by the b_cont pointer of the original message block.

```
mblk_t* dupmsg(mblk_t* msgToDup)
```

copyb

This function returns a new message block which is an exact copy of the original block. It DOES copy the data, and it only duplicates the original message block.

```
mbblk_t* copyb(mbbk_t* msgToDup)
```

copymsg

This function returns a new message block which is an exact copy of the original block. It DOES copy the data. It also duplicates all of the data chained by the `b_cont` pointer of the original message block.

```
mbblk_t* copymsg(mbbk_t* msgToDup)
```

OTAllocMsg

This function allocates a message block that is a reference to a buffer, and will call you back when the message block is freed:

```
typedef void (*EsbFreeProcPtr)(char* arg);  
mbblk_t* OTAllocMsg(void* buf, size_t size, EsbFreeProcPtr, void* arg)
```

The callback procedure will be called back with the value “arg” as it’s parameter when all references to the specified buffer have been freed.

This function works exactly like the STREAMS function `esballoc`. NOTE: using this function, other modules are allowed to write into your buffer.

OTAllocReadOnlyMsg

This function allocates a message block that is a reference to a buffer, and will call you back when the message block is freed:

```
typedef void (*EsbFreeProcPtr)(char* arg);  
mbblk_t* OTAllocReadOnlyMsg(void* buf, size_t size, EsbFreeProcPtr, void* arg)
```

The callback procedure will be called back with the value “arg” as it’s parameter when all references to the specified buffer have been freed.

This function works exactly like the `OTAllocMsg`, except that the `datab->db_ref` field of the `mbblk_t` is set to 2, which tells other modules that they cannot modify the buffer..

Open and Close support Code

Open Transport provides functions for performing several of the common actions required by essentially all module open and close procedures.

These routines rely on a global variable declared by the module which serves as the head of the list of all module instances. The typical C declaration is:

```
static char* gModuleListHead;
```

The address of this variable is an argument passed to the various routines. The routines assume this variable is NULL before the first call to `mi_open_comm`, that this list is manipulated only by the module declaring the list head, and only by the functions described in this section.

mi_close_comm

This function must be used in a module's close procedure if `mi_open_comm` was used in the open procedure. It frees the structures allocated by `mi_open_comm` and removes the current instance from the linked list of the module's instances.

```
int mi_close_comm(char** list_head, queue_t* q);
```

`list_head` Address of the global variable that is the module instance list head passed to `mi_open_comm`.

`q` The queue argument passed to the close procedure, i.e., this module instances' read-side queue.

This function always returns 0.

Notes:

- 1) Each `mi_open_comm` needs a matching `mi_close_comm` (or `mi_close_detached`). Using one without the other has unpredictable results.
- 2) If this was the last instance of this module, `list_head` will be NULL after `mi_close_comm` returns.

mi_close_detached and mi_detach

These functions permit a module to preserve its instance data structure after the associated queue has been closed (deallocated). Typically, the module is a protocol module which must complete an orderly termination of a remote connection, even though the stream it's in is about to be closed.

```
void mi_detach(queue_t* q, char* ptr)
```

```
void mi_close_detached(char** list_head, char* ptr)
```

list_head	Address of the global variable that is the module instance list head passed to <code>mi_open_comm</code> .
ptr	Pointer to the instance data (<code>q_ptr</code> field) that needs to be kept after the queue is closed.
q	The queue argument passed to the close procedure, i.e., this module instance's read-side queue.

Notes:

- 1) `mi_detach` is called from a module to disassociate the instance data from the queue and to remove the module from the global list of open modules. This is normally done from the module close procedure.
- 2) `mi_close_detached` is called to release the instance data and perform other internal cleanup. Note that `mi_close_detached` must be called; it is not sufficient simply to call `freeb` to release the instance data.
- 3) Your `TerminateStreamModule` entry point will not be called until all detached modules have had `mi_close_detached` called.

mi_open_comm

This function performs a number of common initialization actions which are part of most module open procedures.

```
int mi_open_comm(char** list_head, size_t size, queue_t* q, dev_t* devp, int flag,
                 int sflag, cred_t* credp);
```

list_head	Address of the global variable that is the module instance list head. This variable must be initialized to NULL at compile or load time, and then never changed directly, only by passing it as an argument to mi_open_comm functions.
size	The number of bytes to allocate for the queue's instance data
q	The queue argument passed to the open procedure, i.e., this module instance's read-side queue.
devp	Pointer to the device number. For clone opens, the device number is returned; for non-clone opens, the device is passed in. This value was passed to you in your open procedure.
flag	Flag from the open system call. This flag is passed to you in your open procedure.
sflag	0 for a normal device open; CLONEOPEN for a clone open of a device, MODOPEN for a module open. This flag is passed to you in your open procedure.
credp	Pointer to the credentials structure for the process issuing the open. This parameter is passed to you in your open procedure. The credp pointer is not currently used in Open Transport, but it may be at some time. This parameter describes the privileges of the client that is opening the stream. If you just pass the value to mi_open_comm, the right things will happen even if Open Transport starts using the parameter in the future.

This function returns 0 on success. It returns ENXIO on failure.

Notes:

- 1) The last five arguments are the same as the arguments to the open procedure itself.
- 2) `mi_open_comm` assumes that `list_head` is NULL if this is the first open instance of this module.
- 3) `mi_open_comm` assigns a minor device number to the new stream. if `sflag` is 0, the minor number specified by the `*devp` argument is used. Otherwise, for `MODOPEN` and `CLONEOPEN`, a unique minor number ≥ 10 is assigned (Device numbers 0 through 9 are reserved for the module writer as special access codes).

If a given minor number is requested, and another stream already has it open, then an `ENXIO` error will be returned.
- 4) The instance data is allocated to be size bytes (plus an amount for internal structures. Each queue's `q_ptr` field is set to point to this same structure; the internal fields are "hidden" from the module, being located at negative offsets from `q->q_ptr`.
- 5) If a module requires separate instance data for the read and write queues, it must do this indirectly by allocating its own instance data, and storing a pointer to each in the shared instance data.
- 6) A module cannot simply call `freeb` or `OTFreeMem` on the instance data created by `mi_open_comm`; `mi_close_comm` or `mi_close_detached` must be used to free the instance data and remove the queue from the list of module instances.
- 7) The device number of the stream is stored into `*devp`. For `MODOPEN`, the original value of `*devp` is ignored and left unchanged.

mi_next_ptr

This function is used to traverse the linked list of open module instances.

```
char* mi_next_ptr(char* ptr)
```

`ptr` Pointer to the instance data (`q_ptr`) field for which the "next" instance is desired.

This function returns a pointer to the instance data of the next module instance in the linked list. It returns NULL if `ptr` is from the last instance in the list.

If the instance data is of type `xx_t`, the global list head variable is `xx_list_head`, and `xyp` is of type `xx_t*`, then list traversal using this function takes the form:

```
for (xyp = (xx_t*)xx_list_head; xyp != NULL; xyp = (xx_t*)mi_next_ptr((char*)xyp))
```

mi_open_detached

This function creates an instance data structure and chains it into the list of module instances, without requiring an existing stream or module instance (queue) with which to associate the instance data.

It is normally called by modules which need to access a module's instance data before an actual stream has been created. Later, when the module's open routine is called, this floating instance can be used as the new queue's instance data (`mi_open_comm` would not be called in this case; the module's open routine would need to locate the floating instance using its own mechanisms).

```
char* mi_open_detached(char** list_head, size_t size, dev_t* devp)
```

This function returns a pointer to the newly created module instance data, or NULL if memory is not available.

mi_bufcall

This function provides a reliable wrapper for the standard `STREAMS bufcall` function.

```
void mi_bufcall(queue_t* q, size_t size, int pri)
```

q	The queue to enable when memory is available
size	Number of bytes needed (as passed to the failed <code>allocb</code> call)
pri	(unused)

Notes:

- 1) This function may only be used when `mi_open_comm` and `mi_close_comm` are being used in the module open and close procedures.
- 2) `mi_bufcall` fixes two problems that exist with the standard `bufcall` function:
 - 1) `bufcall` has no provision to ensure that the stream which makes the call has not closed. If the `bufcall` callback function is `qenable`, which is often the case, the resulting callback will attempt to reference a nonexistent queue. `mi_bufcall` ensures that the queue is still valid before performing the callback to `qenable`.
 - 2) The `bufcall` call itself may fail because of lack of resources. When `mi_bufcall` detects such a failure, it sets a timer and tries again when the timer fires. This process is repeated until the `bufcall` succeeds.
- 3) `mi_bufcall` uses the module instance list maintained by `mi_open_comm` and `mi_close_comm` to determine if the stream is still open before attempting the callback. The standard `bufcall` function is used to schedule the callback; the ultimate callback function is always `qenable` on the queue passed to `mi_bufcall`. The module writer is responsible for setting appropriate flags in the queue's instance data, so that the service routine will be able to determine that it has been called as part of a `bufcall` callback, if necessary.

IOCTL Support functions

The `mi_copy` facility is a collection of functions which simplifies ioctl processing. These functions arose because of the need to process both `I_STR` and `TRANSPARENT` ioctl forms of the same command. These function permit both type of ioctl to be processed with the same logic. In this section, we focus on how the `mi_copy` facility is used.

To use the `mi_copy` facility, a module's write-side put procedure calls `mi_copyin` in response to an `M_IOCTL` message. `mi_copyin` will then perform the necessary processing depending upon the type of the ioctl. As a result of this processing, regardless of the original ioctl type, an `M_IOCDATA` message will be passed to the module's write-side put procedure.

When an `M_IOCDATA` message arrives in the write-side put procedure, the module **must** call `mi_copy_state` to determine which message is arriving. `mi_copy_state` returns a value which may be used in a switch statement whose case labels are defined by the macro `MI_COPY_CASE`. The sample below shows a simplified example from a typical module put procedure:

```
mblk_t* mp1;

switch (mp->b_datap->db_type) {
case M_DATA:
    ...
case M_IOCTL:
    /* Set copyin_size = 1st buffer size per ioc_cmd */
    mi_copyin(q, mp, NULL, copyin_size);
    return 0;
case M_IOCDATA:
    switch (mi_copy_state(q, mp, &mp1) ) {
    case -1:
        return 0;
    case MI_COPY_CASE(MI_COPY_IN, 1):
        /* process copied-in data in mp1
        mp2 = mi_copyout_alloc(q, mp, uaddr, ubuflen);
        // fill in mp2 with data to copy out to uaddr
        mi_copyout(q, mp); // mp is correct here
        return 0;
    case MI_COPY_CASE(MI_COPY_OUT, 1):
        mi_copyout(q, mp); // Copy out the netbuf
        return 0;
    case MI_COPY_CASE(MI_COPY_OUT, 2):
        mi_copy_done(q, mp, 0); // All done
        return 0;
    }
}
```

Each M_IOCADATA message has associated with it a "direction" and a state. The state is a simple count of the number of M_IOCADATA messages already processed in the direction indicated ("in" or "out"). For example, MI_COPY_CASE(MI_COPY_IN, 1) corresponds to the first message being copied in. If this is an I_STR ioctl, this is the data buffer from the original M_IOCTL; If a TRANSPARENT ioctl, this is the data from an M_COPYIN request. The module needn't care (and can't really tell at this point) which type of ioctl was issued. Again referring to the previous example, it knows that after calling mi_copy_state, the message pointed at by mp is the M_IOCTL being processed, and mp1 is the data buffer.

In a more complex case, there may be multiple buffers to copy in. This is done by calling mi_copyin and adding additional MI_COPY_IN cases to the mi_copy_state switch.

For each buffer to be copied out, mi_copyout_alloc is called to allocate the buffer before calling mi_copyout to copy contents of the buffer. If more than one copy out operation is needed, mi_copy_state and MI_COPY_CASE are used to control what is copied. The example shows two buffers being copied out: a netbuf structure and the buffer it points to.

Additional details are provided in the prototype descriptions which follow.

mi_copyin

This function is called to copy data from a user buffer into the kernel.

```
void mi_copyin(queue_t* q, mblk_t* mp, char* uaddr, size_t len);
```

q	The queue argument to the current write-side put procedure from which mi_copyin is being called.
mp	The M_IOCTL or M_IOCADATA message being processed. This message must not be modified by the module except by calling mi_copy routines.
uaddr	The user-space buffer address from which data will be copied. This argument must be NULL when mi_copyin is called for the original M_IOCTL message; mi_copyin determines the buffer address from the M_IOCTL message. For subsequent calls for M_IOCADATA messages, this address must be extracted from data structures being passed in by the ioctl itself, e.g., buffer addresses from a netbuf structure.
len	The number of bytes to copy-in

This function may be called multiple times to copy-in multiple user buffers. mp is always the message passed to the put procedure.

mi_copyout

This function is called to copy data to a user buffer. Data to be copied out must be stored in message blocks allocated by mi_copyout_alloc.

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

- | | |
|----|--|
| q | The queue argument to the current write-side put procedure from which mi_copyout is being called. |
| mp | The M_IOCTL or M_IOCDSA message being processed. This message must not be modified by the module except by calling mi_copy routines. |

mi_copyout_alloc must be used to allocate the message block into which the data to be copied-out will be placed. Note, however, that the message pointer returned from mi_copyout_alloc is not passed as an argument.

mi_copyout_alloc

This function allocates and returns a pointer to a buffer to be copied out by mi_copyout.

```
mblk_t* mi_copyout_alloc(queue_t* q, mblk_t* mp, char* uaddr, size_t len);
```

- | | |
|-------|--|
| q | The queue argument to the current write-side put procedure from which mi_copyout_alloc is being called. |
| mp | The M_IOCTL or M_IOCDSA message being processed. This message must not be modified by the module except by calling mi_copy routines. |
| uaddr | The user-space buffer address to which data will be copied. |
| len | The number of bytes to copy-out. |

The return value is a pointer to a message block of size len, into which the caller can place whatever data is to be copied out. NULL is returned if memory cannot be allocated.

Notes:

- 1) If multiple copy-out operations and buffers are required, they must be allocated in order from last out to first out. That is, the last buffer allocated will be the first copied out.
- 2) You may allocate all copy out buffers at one time, or you may alternate mi_copyout and mi_copyout_alloc calls
- 3) Note that mp is the message pointer passed to a subsequent mi_copyout function call, but the caller puts the data to be copied out into mp1. Internally, mp points to the first message block in a chain of mi_copyout_alloc'd message blocks.

- 4) Do not free or otherwise manipulate the mblk fields of the message returned by `mi_copyout_alloc`.

MI_COPY_CASE

This macro returns a constant which can be used as a case label for a switch statement that switches on the return value of `mi_copy_state`.

```
#define MI_COPY_CASE(dir, count)
```

dir Direction of current operation. Sepcified as one of the two symbolic constants `MI_COPY_IN` or `MI_COPY_OUT`.

count The number of copy operation s already completed, or equivalently, the number of the current `M_IOCADATA` message being processed.

mi_copy_done

This function is called to complete an ioctl which copies nothing out, or as the last case after multiple copy outs.

```
void mi_copy_done(queue_t* q, mblk_t* mp, int err);
```

q The queue argument to the current write-side put procedure from which `mi_copy_done` is being called

mp The `M_IOCTL` or `M_IOCADATA` message being processed. This message must not be modified by the module except by calling `mi_copy` routines.

err The ioctl return value to set into the `ioc_error` field of the `iocblk` structure

Notes:

- 1) If necessary, call `mi_copy_done` to complete the ioctl. This step is only required after the last copy-out of a transparent ioctl or for either type of ioctl when nothing is being copied out to the caller
- 2) If the ioctl neither copies in nor out any data, only `mi_copy_done` and optionally `mi_copy_set_rval` are required.
- 3) Your code must provide for calling `mi_copy_done` for any `M_IOCADATA` message that it doesn't expect.

mi_copy_set_rval

If the ioctl has a non-zero return value, that return value must be set by this function before the final call to `mi_copyout` or `mi_copy_done`. This function must be called before the last `mi_copyout` or `mi_copy_done` call. If the ioctl neither copies in nor out any data, only `mi_copy_done` and optionally `mi_copy_set_rval` are required.

```
void mi_copy_set_rval(mblk_t* mp, int rval);
```

mi_copy_state

This function returns the current internal state and optionally the next message block to process. The values match those returned by the `MI_COPY_CASE` macro

```
int mi_copy_state(queue_t* q, mblk_t* mp, mblk_t* mpp);
```

<code>q</code>	The queue argument to the current write-side put procedure from which <code>mi_copyxxx</code> is being called
<code>mp</code>	The <code>M_IOCTL</code> message being processed
<code>mpp</code>	Pointer to <code>mblok_t</code> pointer, into which the pointer to the just-copied data is placed.

TPI Support functions

The functions in this section isolate the construction of TPI messages. The functions handle the details of allocating and formatting messages.

Arguments to all the functions follow the same pattern. "ACK" functions accept a pointer to the TPI message being ACK's. Message which permit data to be appended through the b_cont field are passed a trailer_mp argument which is the message block(s) to append. Other arguments correspond to field in the TPI message being created. The following table summarizes:

mp	Functions which return an acknowledgement, some form of T_XXX_ACK message, are passed the message to be ACK'd in this argument. IMPORTANT: The message pointed at by this argument is either reused or freed by the function. Therefore it must not be referenced by the calling code after the function returns.
trailer_mp	If the TPI message supports optional data in attached M_DATA message blocks, the optional data is passed to the function in this argument.
Other	All other arguments are values to be copied into the corresponding field of the TPI message.

Arguments not following this pattern are described in the individual function descriptions which follow. On success each function returns a pointer to the completed message; on memory allocation failure, it returns NULL.

mi_tpi_ack_alloc

```
mblk_t* mi_tpi_ack_alloc(mblk_t* mp, size_t size, long type)
```

size	Length of ACK message to allocate, typically specified as sizeof(struct T_XXX_ack).
type	The PRIM_type of the ACK message being allocated.

This function fills in only the primitive type; the caller must fill in all other fields.

mi_tpi_err_ack_alloc

```
mblk_t* mi_tpi_err_ack_alloc(mblk_t* mp, int tlierr, int unixerr)
```

This function creates a T_ERROR_ACK for the TPI message contained in mp.

mi_tpi_ok_ack_alloc

```
mblk_t* mi_tpi_ok_ack_alloc(mblk_t* mp)
```

This function creates a T_OK_ACK for the TPI message contained in mp.

Other TPI prototypes

These are the prototypes for functions whose workings should be obvious:

```
mbblk_t* mi_tpi_conn_con(mbbk_t* trailer_mp, char* src, size_t srcLength, char* opt,
                        size_t optLength);

mbblk_t* mi_tpi_conn_ind(mbbk_t* trailer_mp, char* src, size_t srcLength, char* opt,
                        size_t optLength, int seqnum);

mbblk_t* mi_tpi_conn_req(mbbk_t* trailer_mp, char* dest, size_t destLength, char* opt,
                        size_t optLength);

mbblk_t* mi_tpi_discon_ind(mbbk_t* trailer_mp, int reason, int seqnum);

mbblk_t* mi_tpi_discon_req(mbbk_t* trailer_mp, int seqnum);

mbblk_t* mi_tpi_info_req(void);

mbblk_t* mi_tpi_ordrel_ind(void);

mbblk_t* mi_tpi_ordrel_req(void);

mbblk_t* mi_tpi_uderror_ind(char* dest, size_t destLength, char* opt, size_t optLength,
                        int error);

mbblk_t* mi_tpi_unitdata_ind(mbbk_t* trailer_mp, char* src, size_t srcLength,
                        char* opt, size_t optLength);

mbblk_t* mi_tpi_unitdata_req(mbbk_t* trailer_mp, char* src, size_t srcLength,
                        char* opt, size_t optLength);
```

For these next four functions, the `type` parameter is reserved and should be set to 0.

```
mbblk_t* mi_tpi_data_ind(mbbk_t* trailer_mp, int flags, int type);

mbblk_t* mi_tpi_data_req(mbbk_t* trailer_mp, int flags, int type);

mbblk_t* mi_tpi_exdata_ind(mbbk_t* trailer_mp, int flags, int type);

mbblk_t* mi_tpi_exdata_req(mbbk_t* trailer_mp, int flags, int type);
```

Synchronization support

The `mps_become_writer` function is provided to support modules which have `SQLVL_QUEUE`, `SQLVL_QUEUEPAIR`, or `SQLVL_SPLITMODULE` (version 1.5 or later) set. At some time during operation, it may become necessary to access a resource that is shared between all instances of the module. This access must be coordinated so that an instance updating the resource does not collide with an instance reading the resource (ARP tables are one such example). In order to support this, Open Transport supplies the `mps_become_writer` function:

```
typedef void (*OTWriterProcPtr)(queue_t*, mblk_t*);

void mps_become_writer(queue_t* q, mblk_t* mp, OTWriterProcPtr proc);
```

This function will lock out all instances of the module owning the queue "q" (and any "writer buddies" specified in the `install_info`), and when that is done, call back the function specified by the `OTWriterProcPtr`. For the duration of the call, the function has sole access to the variables of the module.

You should be aware, however, that this function is expensive in terms of time. It requires that each queue or queue pair in existence be acquired. A new synchronization level, `SQLVL_SPLITMODULE`, has been introduced that works almost as well as `SQLVL_QUEUE` or `SQLVL_QUEUEPAIR`, but calling `mps_become_writer` is much quicker. `SQLVL_SPLITMODULE` allows you to be inside your module once from an upper queue and once from a lower queue. If you are writing a module, we highly suggest that you use this synchronization level once it becomes available (in Open Transport 1.5 and later). This level allows an upper level protocol to receive a packet, and send a reply without the reply getting hung up on a queue until the stack unwinds, which reduces gaps on the wire.

IOCTL Messages

This section describes some IOCTL message that Open Transport has defined that you might want to consider supporting.

These first two IOCTLs are part of Open Transport's arbitration mechanism for ports that cannot demultiplex incoming data (like serial ports). In order to support these IOCTLs, your driver must be written with a few things in mind:

- 1) It should allow any client to open a STREAMS, whether or not there is already a client using the driver.
- 2) It should classify STREAMS in one of 3 states - nonused, listening, or connected. The "connected" state implies that a connection is in progress and data is actively being transferred, while "listening" implies that the stream is waiting for an incoming connection request. Only one stream may be in a state other than nonused.

The following IOCTL message complete the support for the arbitration mechanism for non-shareable ports:

I_OTYieldPort This IOCTL is only issued by Open Transport. It passes a 4-byte value that is either a 0 or a 1. A "1" tells your driver that it should place the current "listening" client in a backup position (if there is one), and accept the next Bind or Connect as the new active client. The driver should return an ENXIO error if the current client is in the "connected" state, and an ENOENT error if there is no current client. Otherwise, a 0 value should be returned. A `T_event_ind` message should be sent to the current "listening" client with an `EVENT_code` of `kOTProviderIsDisconnected` and an `EVENT_cookie` of `NULL` (see `tihdr.h` for a definition of the `T_event_ind` message). When the new active client unbinds (if `qlen <> 0`) or disconnects (if `qlen == 0`), the previous listening client should be restored as the listener and a `T_event_ind` message should be sent to the client with an `EVENT_code` of `kOTProviderIsReconnected` and an `EVENT_cookie` of `NULL`. This IOCTL is required to be supported by those drivers that have set the `kOTPortCanYield` bit in the `install_flags`. It should be supported by all devices which cannot demultiplex incoming data to multiple clients (like serial ports).

A "0" value indicates that the yield was canceled, and you should restore the previous client to ownership of the port.

In any case, after receiving a yield request, your driver should set about a 10 second timer. If no one else grabs ownership of the port in that time, ownership should automatically revert to the previous client. Remember, you should always send the `kOTProviderIsReconnected` event indication whenever you revert back to the previous client.

This IOCTL should be accepted on any queue on the port, since Open Transport will use a provider supplied by the client that is requesting the yield.

The next set of IOCTL messages can be supported by any stream module or driver if it make's sense to support it:

I_OTGetMiscellaneousEvent

This IOCTL tells a driver or protocol that the client is interested in any miscellaneous events (using the T_event_ind message defined by Open Transport) that the protocol or driver wishes to send up. AppleTalk uses this IOCTL and the T_event_ind messages to inform clients when routers go up and down, as well as other kind of incidental messages. The value accompanying the IOCTL data should be a 4-byte value that is 0, 1, or -1. Any other value is an error. A value of 0 requests that miscellaneous events no longer be delivered. A value of 1 requests that miscellaneous events be delivered. A value of -1 does not change the mode, and is just used where the client wants to read the current state.. The return value of the IOCTL should be a 0 or a 1 reflecting the current or new state of miscellaneous event delivery. If you don't support the IOCTL, pass it on or NAK it, as appropriate.

I_OTSetFramingType

This IOCTL is used by registered port drivers that have specified a set of capability bits in the fCapabilities field of the OTPortRecord. The only time this IOCTL is needed is if the information in a DL_INFO_REQ or T_INFO_REQ message is different depending on what capability (most often used to specify a framing type) is used. For instance, Ethernet can return a dl_mac_type of DL_CSMACD or DL_ETHER depending on the type of framing chosen. Since the TCP/IP stack determines whether it should use straight ethernet headers or 802.2 headers based on the dl_mac_type returned from a DL_INFO_REQ, the code that creates a TCP/IP stack opens the Ethernet driver and sends an I_OTSetFramingType IOCTL to tell the Ethernet driver which value to put in the dl_mac_type field (it specifies the bit kOTFraming8022 to get DL_CSMACD, and kOTFramingEthernet to get DL_ETHER. The return value of this IOCTL is the current "capability" in effect, (or 0, if no "capability" is in effect.). See the *Open Tpt Ethernet Dev. Note* for more information on this IOCTL and it's use.

This IOCTL is also used by Serial and ISDN drivers to select the appropriate framing types for the link.

The value accompanying the IOCTL data should be a 4-byte value that is either a single bit, or a -1. -1 is used to read the current "capability" that is in effect, and any other single-bit value is a request to set the corresponding "capability".

I_OTSetRawMode

This IOCTL is used to deal with "raw" mode packets for DLPI drivers. See the *Open Tpt Ethernet Dev. Note* for more information on this IOCTL and its use.

I_OTNotifyAllClients

This IOCTL requests a module or driver to send a `T_event_ind` message to all of its clients. This IOCTL is sent as an `I_STR` IOCTL, with 12 bytes of data corresponding to the structure:

```
struct OTIOctlNotifyInfo
{
    OTEventCode    fCode;
    void*          fCookie;
    UInt32         fNotifyType;
}

enum
{
    kOTNotifyAllModules = 0, kOTNotifyInterestedModules = 1,
    kOTNotifyControlModules = 2
};
```

The `fNotifyType` field tells the module which group of modules should be notified.

Appendix A- Synchronization

Synchronization is one of the biggest problems when writing complex protocols and drivers. While the synch queue levels go a long way to controlling synchronization, performance of a system is enhanced if you can use less restrictive synchronization mechanisms than SQLVL_MODULE. However, with this comes the headaches of controlling access to critical resources. The API `mps_become_writer` helps with this synchronization, but it has a severe shortcoming - the less restrictive your synchronization, the longer it takes.

Open Transport has defined a very simple data structure and a couple of routines to go with it, called an OTGate. An OTGate is associated with a procedure (the OTGateProcPtr), a list of things to process, and a lock. When you want to process something that deals with a critical resource, call `OTEnterGate` with some kind of structure that will tell the gate procedure what to do.

WARNING: This structure MUST be allocated on a 4-byte boundary, or unpredictable results will occur.

```
typedef Boolean      (*OTGateProcPtr) (OTLink*);

struct OTGate
{
    OTLIFO          fLIFO;
    OTList          fList;
    OTGateProcPtr fProc;
    SInt32          fNumQueued;
    OTLock          fInside;
    UInt8           fFiller[3];
};

typedef struct OTGate      OTGate;

extern void          OTInitGate(OTGate*, OTGateProcPtr proc);
extern Boolean       OTEnterGate(OTGate*, OTLink*);
extern void          OTLeaveGate(OTGate*);
```

The gate operates by keeping a LIFO and a simple list, as well as an atomic lock. When you enter the gate, the link is thrown atomically on the LIFO (unless, of course, the link is NULL) and the atomic lock is acquired. If it cannot be acquired a false is returned immediately from the `OTEnterGate` function. If the lock can be acquired, if the incoming link was NULL, true is returned from the `OTEnterGate` function. This allows "locking out" the gate until you call `OTLeaveGate`.

If the incoming link was not NULL, then the first item on the list is run. If the list is empty, the LIFO is atomically stolen, reversed, and stored on the list and we loop back around to run the first item on the list.

If the gate procedure returns `true`, it means it is finished with the link and the lock can be relinquished. In this case, we loop back around and run the next item on the list. If it returns `false`, then the gate procedure wants to manually leave the gate, so we just exit the `OTEnterGate` function. Then sometime in the future, a call to `OTLeaveGate` will be made to relinquish the lock. The call to `OTLeaveGate` will once again attempt to run things on the LIFO and the list until they are both empty, at which point, the lock will be dropped.

Since someone might have queued something up just prior to us dropping the lock, we have to check the LIFO one last time, and if it is not empty, attempt to acquire the lock and loop back around. If we can't acquire the lock, then some other thread is taking care of it and we are free to exit the function.

In Apple's AppleTalk implementation, `mblk_t`'s are used to queue up on the gate.

For instance, whenever we want to add or remove something from a critical list (such as our socket list), we call:

```
OTEnterGate(&theListGate, (OTLink*)&mp->b_next);
```

This calls the gate procedure, which gets passed the `mblk_t`:

```
Boolean MyGateProc(OTLink* link)
{
    mblk_t* mp = OTGetLinkObject(link, mblk_t, b_next);
    /*
     * Here, we look at the "mp" and do whatever we're supposed to do
     */
    return true;
}
```

The gate procedure returns `true` if it has completed processing the "`link`". It returns `false` if it has not, and at some later time, the code must call `OTLeaveGate` when done processing "`link`". While you are executing in the gate procedure, you are guaranteed that no one else can be executing the gate procedure (at least, not for the specified `OTGate` object).

Different `OTGate` objects can be set up to protect different resources.

Of course, most lists in networking are searched often, and modified infrequently. Having to go through the overhead of the gate procedure to look things up is not ideal, and in addition, the flow of control has to move from the point where you want to search the list to somewhere in the gate procedure (or called by it).

As long as you remove and add entries to the list in your gate procedure properly (by insuring that you don't invalidate pointers in the list so that they could point to deleted memory), you can have multiple readers of the list interrupting a single writer with no problem.

The way to do this is simple:

```
Boolean inGate = OTEnterGate(&theListGate, NULL);  
/* Search the List here */  
if ( inGate )  
    OTLeaveGate(&theListGate)
```

Note that calling OTEnterGate with a NULL link pointer attempts to acquire the gate. It returns true if it did, and false if it did not. This operation is very fast.

If it did acquire the gate, we have exclusive access to the list (except for possibly other readers). If it did not, then there is at most 1 writer that we have interrupted, and we know it's safe to scan the list (assuming you don't invalidate pointers in the list as a writer).

After searching the list, if we "entered" the gate, we need to call OTLeaveGate so that any writers that are queued up get run.

Appendix B- Performance hints

- 1) Don't keep resetting timers for high-frequency events. Let your timer free-run and modify your timeout algorithms accordingly.
- 2) Support and use FastPath if at all possible to avoid the allocation of M_PROTOS.
- 3) Keep your receive path and transmit path as short as possible.
- 4) Handle your own flow control. This is easily done by always handling data in a "put" routine, setting your queues high and low water marks to 31 and 16 respectively. To flow control, do a `putq` of a 32-byte message onto your queue. To lift flow control, do a `rmvq` on your flow control message.
- 5) If you are a TPI module or driver, support the `XTI_SNDBUF`, `XTI_RCVBUF`, `XTI_SNDLOWAT`, and `XTI_RCVLOWAT` options. Use the `XTI_SNDBUF` and `XTI_SNDLOWAT` to set the stream-head high and low water marks, as well as to regulate your own send window. Use the `XTI_RCVBUF` option to set your lower queue's high water mark and the `XTI_RCVLOWAT` option to set your lower queue's low water mark.
- 6) For drivers that can selectively disable their own interrupts, spend as long as you can in your interrupt service routine. This includes allocating mblks for messages, finding the lower queues to put them on, and then doing a `putq` on the correct lower queues. Then check your hardware or transmit queue again. Maybe more message have arrived or DMA buffers have freed up, allowing you to do more processing without taking another interrupt.
- 7) Use large packet sizes, if possible. The larger the packet size, the higher the throughput. The time it takes to transmit a single packet is all the time there is to go up and down the stack once if you want to saturate the link.

10 MB - 600 byte packet = 480 μ Sec
100MB - 600 byte packet = 48 μ Sec
100MB - 1500 byte packet = 120 μ Sec
100MB - 8192 byte packet = 655 μ Sec
- 8) When writing client applications and test programs, for best performance, modify the default buffer sizes. The defaults are set for a compromise between speed and memory usage.
- 9) Avoid mixed-mode switches like the plague in your send and receive paths. Each mixed-mode switch costs 25-40 μ Sec.

- 10) Don't use `BCOPY/bcopy` for small copies that are of known size. Do the copy inline.
- 11) Code any operations that have to be done on every byte in the data in assembly language.
- 12) If you run out of memory, and aren't in a position to "toss" the data, be sure to disable the appropriate queue until memory becomes available. You don't want to keep making the situation worse. Better yet, use the `mi_open_comm` and `mi_bufcall` facility to handle it all automatically for you.
- 13) If you have your own DMA buffers for incoming data, and aren't restricted by a ring-type architecture where you can't use free buffers that aren't contiguous in the ring, consider using `esballoc` for about 1/2 your DMA buffers. If the client is a high-speed client, you may get them back before you have to begin copying mblks. This also helps reduce footprint, by avoiding the duplication of data.
- 14) As a client, use `AckSends` and no-copy receives to read the data off of the streamhead. Of course, on no-copy receives it is crucial that you do something with the data quickly and release the buffer if you want maximum throughput.
- 15) Open Transport needs to define a way to appropriately lock down mblks for DMA. Possibilities: 1) A new "allocbDMA" call that keeps a separate set of mblks that are already locked down (downside is footprint); 2) A `LockMblk` and `UnlockMblk` set of APIs; 3) An Open Transport memory allocator for DMA-able memory that can be called at interrupt time, making it feasible to do `esballocs` on the data. 4) Any suggestions?

Appendix C - Random Notes/Warnings

- 1) The SVR4.2 STREAMS guide states that modules may sleep in their `Open` and `Close` routines (for the MacOS, this means making synchronous I/O calls). In the Open Transport version of STREAMS, this is not true due to the nature of the Macintosh Operating System. `Open` and `Close` routines may not make synchronous calls. Use the `InitStreamModule` and `TerminateStreamModule` are always called at System Task time. You need to structure your module or driver so that any synchronous operations or calls to the Macintosh toolbox are done in these routines. Remember that for port drivers, these entry points will be called for any instance of your driver that is being instantiated or destroyed, whether or not your shared library is being unloaded.
- 2) Drivers for hardware that can be ejected or removed (e.g. PCMCIA cards), should call

```
OTChangePortState(myPortRef, kOTPortDisabled, kOTPortEjected)
```

so that Open Transport can close down all of the providers that are using the port, and notify clients that the port has gone away. If the card is reinserted, call

```
OTChangePortState(myPortRef, kOTPortEnabled, kOTNoError)
```

to notify all clients that the port is back on-line.
- 3) Drivers that sit atop connection-oriented links (e.g. PPP sitting atop a modem device), should also call `OTChangePortState` if they receive a `T_DISCONNECT` from the modem device. In the `Close()` routine of their driver, or in response to a final `Unbind()` call, they should reenable themselves. This allows the upper-level protocols to know that the link has died,. Steps can then be taking to unbind or close all providers using the link, and then bringing the link back up, presumably causing an attempt to reestablish the connect(e.g. redial the phone).
- 4) Drivers that use `OTChangePortState` still need to protect themselves from being used. This can be done by using the `M_ERROR` message to send an error up to the streamhead, or it can be done by local state and dealing with the fact that the link below has disconnected on a message-by-message basis. (If you use `M_ERROR`, remember to send another `M_ERROR` message to clear the error when you are again open for business!)

- 5) Use care when calling the `rmvq` function. Unix documentation states that calling `rmvq` for a message that is not on the queue is a system panic. Under Open Transport, the `rmvq` function will remove the message from whatever queue it is on (or trash location 0 if it's not on a queue), and subtract the size of the message from the number of bytes on the queue. This will cause "canput" to fail when called on your queue, quite probably causing Macintosh networking to hang.
- 6) The `bcopy` routine supplied by Open Transport eventually calls the `BlockMoveData` trap supplied by the system (on 68K, this only occurs for transfers > 68 bytes - for less than 68 bytes, it's faster to do it ourselves in-line). The `BlockMoveData` routine is not suitable for moving data into and out of uncached memory (like DMA buffers on some systems). Use the `BlockMoveUncached` routine instead.

Index

allocb 38
 arbitration 53
 ASLM 6
 bufcall 45
 CFM 6
 Close 62
 copyb 39
 copymsg 39
 DLPI 5
 dupb 38
 dupmsg 38
 freeb 38
 freemsg 38
 GetOTInstallInfo 7
 Index 64
 InitStreamModule 11, 37, 62
 install_flags 9
 install_info 7
 I_OTGetMiscellaneousEvent 55
 I_OTNotifyAllClients 56
 I_OTSetFramingType 55
 I_OTSetPowerLevel 37
 I_OTSetRawMode 56
 I_OTYieldPort 54
 kOTPortCanYield 54
 kOTPortScannerInterfaceID 27, 29
 kOTProviderIsDisconnected 54
 kOTProviderIsReconnected 54
 kOTPseudoPortScannerInterfaceID 27, 29
 memory allocations 12
 mi_bufcall 45
 mi_close_comm 40
 mi_close_detached 41
 mi_copyin 47
 mi_copyout 48
 mi_copyout_alloc 48
 MI_COPY_CASE 49
 mi_copy_done 49
 mi_copy_set_rval 50
 mi_copy_state 50
 mi_detach 41
 mi_next_ptr 43
 mi_open_comm 42
 mi_open_detached 44
 mi_timer 34
 mi_timer_alloc 34
 mi_timer_cancel 34
 mi_timer_free 34
 mi_timer_valid 34
 mi_tpi_ack_alloc 51
 mi_tpi_conn_con 52
 mi_tpi_conn_ind 52
 mi_tpi_conn_req 52
 mi_tpi_data_ind 52
 mi_tpi_data_req 52
 mi_tpi_discon_ind 52
 mi_tpi_discon_req 52
 mi_tpi_err_ack_alloc 51
 mi_tpi_exdata_ind 52
 mi_tpi_exdata_req 52
 mi_tpi_info_req 52
 mi_tpi_ok_ack_alloc 51
 mi_tpi_ordrel_ind 52
 mi_tpi_ordrel_req 52
 mi_tpi_uderror_ind 52
 mi_tpi_unitdata_ind 52
 mi_tpi_unitdata_req 52
 M_DATA 5
 M_PCPROTO 5
 M_PCSIG 34
 M_PROTO 5
 Open 62
 OTAllocMem 38
 OTAllocMsg 39
 OTAllocPortMem 20
 OTAllocReadOnlyMsg 39
 OTAtomicAdd16 35
 OTAtomicAdd32 35
 OTAtomicAdd8 35
 OTAtomicClearBit 35
 OTAtomicSetBit 35
 OTAtomicTestBit 35
 OTChangePortState 62
 OTCompareAndSwap16 35
 OTCompareAndSwap32 35
 OTCompareAndSwap8 35
 OTCompareAndSwapPtr 35
 OTCreateDeferredTask 33
 OTDequeue 36
 OTDestroyDeferredTask 33
 OTEnqueue 36
 OTEnterInterrupt 32
 OTFindPortByDev 25
 OTFreeMem 38
 OTFreePortMem 20
 OTLeaveInterrupt 32
 OTLIFO 36
 OTLIFODequeue 36
 OTLIFOEnqueue 36
 OTLIFOStealList 36
 OTLink 36
 OTPortRecord 19
 OTRegisterPort 19, 28
 OTReverseList 36
 OTScanPorts 27, 28
 OTScheduleDriverDeferredTask 33
 PCMCIA 62
 port scanners 27
 sleep 62
 SQLVL_GLOBAL 8
 SQLVL_MODULE 8
 SQLVL_QUEUE 8
 SQLVL_QUEUEPAIR 8
 SQLVL_SPLITMODULE 8
 streamtab 12
 TerminateStreamModule 12, 37, 62
 TPI 5

