# Native Driver White Paper

**version 1.0**
**May 19, 1994**

Holly Knight
John Fitzgerald
Mike Quinn
Wayne Meretsky

## Overview

This document proposes required interfaces and packaging for Communications drivers on PCI cards. At the time of this writing there are three different native device driver proposals for the Marconi project. Each proposal covers a separate but overlapping piece of the device driver puzzle for Marconi and future OS releases. The design presented here will cover two broad issues. The first goal is to regularize device driver writing so that PCI and non-PCI device drivers can be written to a single specification. The second equally important goal is to have device drivers written to this specification work for both the Marconi and Maxwell releases, as well as any future releases.

Documentation for PCI devices may be found in the following documents:
“Device Driver Development Kit” - ApplePC
“The Macintosh Expansion Manager” - ApplePC

Native Device Driver documents:
    “Device Manager Changes to Support Native Drivers.” - AppleSoft
    “Native Drivers for PCI and other Peripherals in Marconi” - AppleSoft
“AppleSoft PCI Implementation Plan” - AppleSoft

Open Transport/Streams documents:
“PCI/DLPI - Open Transport Integration” - Open Transport
“Open Transport Ethernet Developer Note” - Open Transport
“Streams Modules and Drivers” - Unix Press

In addition to providing general native device driver guidelines we use the native DLPI driver for Open Transport to provide implementation specifics. We hope to see this document become part of both a Device Driver Kit for PCI devices and an Open Transport Software Developers Kit.

## Terminology

In this document we use a set of possibly unfamiliar terms. This list is provided to clarify discussions in subsequent sections.
Family - This term refers to a collection of devices that provide the same kind of functionality. One example of a family is the set of Open Transport devices with their corresponding Open Transport DLPI drivers. A second example is the family of Display Devices.

Scanning - I use the word scanning or scanner to describe code that matches a device with its corresponding driver. The scanning portion of device location is one of the problem areas discussed in this paper. Scanning is probably a sub-function of the family Expert.

Expert - Expert is a term coined by Brenden Creane to describe family code that extracts appropriate information from the system (ROM, Slot Manager, Expansion Manager, etc.) and presents the device specific information to each family device in a format agreed upon (documented). A family Expert is the administration function for a family. Experts are important when devices are brought into the system (classically at boot time) but are not part of the primary data paths to a device.

## Driver Components

All native device drivers are CFM-loadable modules that export their API by using CFM's export-by-name feature. Native drivers are located in either System-ROM, PCI-card ROM, or a file. PCI-card ROM drivers are CFM modules in PCI ROM. PCI drivers may be replaced by newer versions of the driver located in System-ROM or in a file.

Device drivers require scanning and driver code. PCI drivers may optionally provide Open Boot code. The Open Boot PCI ROM code is responsible for installing PCI ROM-based device properties in the device tree in support of plug and play.
All PCI devices support configuration ROM that conform to the PCI Local Bus Specification. Apple does not require vendors to register their cards with DTS; the vendor ID of the PCI card is allocated by the PCI SIG.

Device scanning/Expert code is responsible for scanning the system resources to match device drivers to devices and supply system information to the device driver initialization code. The device driver itself provides the system entry points to control a device. A description of the standard device entry point(s) is detailed in later sections.

### Scanning details

Scanning for devices needs to be a system service. Such a service requires a central repository for device information for all system devices. Lacking a scanning service and central repository for device information, device driver writers must either hard code information (motherboard devices) or use the specific low level services associated with their device to glean required device configuration information.
Examples of the differences between drivers are:

- PCI devices use the Expansion Manager to scan the Device Tree for configuration information. PCI devices must also use the Expansion Manager for interrupt handler registration and possibly for some direct I/O memory accesses.

- Drivers for NuBus devices use the Slot Manager to acquire base addresses and device slot information. NuBus device drivers us the Slot Manager for interrupt handler registration.

- Motherboard devices traditionally have hard coded device base addresses, names, etc. within the driver initialization code. Motherboard devices use OS specific interrupt registration mechanisms.

Device scanning code must recognize the device based on property types and values such as:
Vendor-ID
Device-ID
Physical address
Device name
Device type
Interrupts
Device registers
Version

For PCI devices there must be an additional property:

drvr,Apple,MacOS *(for PCI devices see the Expansion Manager doc)*

This propery is used to locate a PCI driver for a given device.

With device scanning code and a device information repository all native device drivers can be written to use a single set of services. The system scanning system will provide configuration information to native device driver when device initialization driver code is called from the scanner. This will allow PCI, motherboard and NuBus device drivers to be written identically; that is to expect basic hardware information to be made available and not be required to locate or hard code this information within the driver itself. This also allows for a comprehensive driver replacement/overloading capability.

## Native Driver APIs

Within the MacOS today there is only one flavor of programming interface, an application interface or API. What this means is that all services defined within the MacOS are available to all Macintosh Applications. Within the new native Device Manager documentation there is a description of the new API. Application calls to raw devices go through this API.

With the advent of a new set of device driver services and families, I would like to introduce the idea of a new class of programming interfaces. This new class of interfaces is the Device Driver Programming Interface or DPI, that is the set of services and mechanisms defined by and used by a family or class of service. Open Transport, for example, represents a family of services, networking, that uses a well defined set of family specific APIs that are not available to Macintosh Applications.

### Native Device Manager Driver API

The ‘Native Drivers for PCI and other Peripherals’ document specifies a single code entry point will be exported by all native drivers. The high-level semantics of this entry point are described in 'Native Drivers for PCI and other Peripherals in Marconi'. The entry point is:

```
OSErr (*NuDriverEntryPoint)
            (IOCommandID            theID,
            IOCommandContentsPtr    theContents,
            IOCommandCode           theCode,
            IOCommandKind           theKind);
```

In addition to the  entry point, a data symbol that characterizes the driver's functionality and origin must be exported through CFM's symbol mechanism.  The data symbol's name (TheDriverDescription) and structure is as follows:

```
typedef long      DriverDescVersion;
typedef long      DriverVersion;
typedef OSType    DriverDescSignature;

enum
{
theDescriptionSignature = 'mtej'    // must appear in first long
};                                  // of DriverDescription

typedef struct DriverDescription {
DriverDescSignature     Signature;        // Signature of this structure
DriverDescVersion       Version;          // Version
OSType            DeviceFamilyType;       // Family of devices
OSType            driverType;             // Driver type within family
OSType            driverSubType;          // SubType of driver in family
OSType            driverManufacturer;     // Maker of this driver
DriverVersion     driverVersion;          // Driver's version
char              driverNameLength;       // Length of driver's name
char              driverName[32];         // Driver's name
char              driverDescReserved[128];// Reserved for the future
} DriverDescription, *DriverDescriptionPtr;

DriverDescription TheDriverDescription =
{
// Initialize Description Info Here
};
```

See the “Device Manager Changes to Support Native Drivers” document for more information about the fields of the DriverDescription structure.

Documented device families include:

| | |
|---|---|
| 'DISP' | Display Devices |
| 'BLCK' | Block Storage Devices |

A possible device family name for Open Transport:
'OPTP' Open Transport

Family names, such as DISP, BLCK, OPTP are administered by Apple.  Each device family defines driver types and subtypes that are only unique within that family.

**** Do we want to have all networking devices belong to the Open Transport  family and  Apple maintains types and subtyes?  Do we want each manufacturer to identify  their own family types and subtypes?  See the issues section!*

Device driver are created as CFM modules.  CFM modules are always allowed to have persistent specific global data storage. Each instance of a single driver has private static data, and shared code with every other instance of that driver. This is one of the changes from classic MacOS device drivers. CFM is responsible for maintaining the driver context (the driver "a5" world). Device drivers no longer need to hang their private data from a field in their Device Unit Table Entry.

**Native Device Driver API**

All conforming drivers to "Device Manager Changes to Support Native Drivers" provide a single driver entry point NuDriverEntryPoint that switches on the following set of commands:

OpenCmd
CloseCmd
ReadCmd
WriteCmd
ControlCmd
StatusCmd
KillIOCmd
InitializeCmd
FinalizeCmd

These commands, available through the NDRV Device Manager are an Applications API, in the sense that all Toolbox services are available to all Macintosh applications. In addition to providing the NDRV Device Manager interfaces to a driver, certain families of devices require a private interface to their device drivers that does not go through the Device Manager at all. This class of interface is a Driver Programming Interface (DPI). The DPIs are defined per family, and are not available to Macintosh Applications. Should an application discover these DPIs and attempt to make a DPI call, the application will fail. In the V1 world, the application will probably crash with an access violation because the device driver services are in a different address space than the Macintosh Application.

I will list three possible approaches to allow DPI clients to access the private family APIs.

**New Property**
All drivers with family private APIs must provide a property within the System Registry that contains a list of names of exported API structures and functions. The family client searches for the family-API property with the System Registry for a given device. If the property is located, the client does a CFM look-up by name on all the names within the list, and plugs in the returned addresses within a client private structure. Calls and data accesses to the family API would be made via the client private structure.

**Additional CFM exports**
All drivers with family private APIs must export well defined family API names for both API data and API functions. Clients loads the CFM driver resource, and calls the well defined names found within the CFM driver. CFM does the magic to connect the client to the CFM device driver exports.

**DriverGestalt selector**
All new drivers are required to support the `driverGestaltCode` as a subCommand (csCode) to the StatusCmd request. The `driverGestaltSelector` controls the type of information the DriverGestalt call returns. Currently the control selectors may be `'node'` or `'vers'`. To support family APIs, new selector `'fapi'` is defined to request a family API structure. The family API structure contains the private API data and function pointers. The device family API structure is returned in the status call `driverGestaltResponse` field.

****The driver gestalt call with driverGestaltSelector set to 'node' expects a PCI compliant driver to return a pointer the device node within the device tree to which this driver corresponds. What happens when a single device driver controls multiple devices? This also means that one of the bits of information the expert provides to the driver init code is the device node associated with this particular init call. I am assuming one init call per device.*

Open Transport requires a family data structure called `install_info,` and three OS specific driver routines, one device initialization routine, one primary interrupt handler, and finally, a secondary interrupt handler. The `install_info` structure is used by Open Transport to link streams modules to streams device drivers.

Open Transport is not alone in its requirement of three OS specific driver routines. All device drivers will need at least, device initialization, a primary interrupt handler, and a secondary interrupt handler. The device initialization routine is called with the device is located. The primary interrupt handler must be registered for the driver in an OS specific way. The secondary interrupt handler is scheduled in response to a device request. The device request for secondary interrupt scheduling will usually be called from the device driver primary interrupt routine to defer expensive data processing to non-primary interrupt time.

****Where does this get discussed? I think these OS specific driver routines need to be generalized. Should these routines be separate properties? 2nd, 3rd, and 4th CFM exports or what?*

Device drivers do not necessarily need to provide both types of interfaces. New device drivers would use their family API interfaces to the clients but would appear as standard devices to the system. The Device Manager IOCommands would be primarily stubs. New device drivers with no need for a family API would be directly callable through the standard IOCommands interface and would fail to deliver the non-existent family API in some “friendly” way.

**Native Driver Initialization**

In an ideal world, device drivers would be written in a location independent manner. In other words the device driver would not know whether its device was PCI, motherboard, or NuBus based. This is not how the current set of documents describes the device driver world. Within the current documentation there is no clear distinction between device initialization and device open. A device open is (or should be) a connection oriented response to client requests. Initialization can and will occur out-of-band to client requests, for example at boot time, or in the PCMCIA case, when a device is hot swapped into and out of the system.

I present three separate scenarios for device driver initialization. Plan A describes something like the perfect world scenario. Plan B details how an initialization sequence will occur if there is no coordination between the PCI/NuBus and motherboard devices. Plan C describes an attempt to use the Expansion Manager as the device repository for all device information. All three plans cover device initialization, this must be distinct from device open requests.

**Plan A**

In the PCI documentation the statement is made "When MacOS is launched on a Power Macintosh computer with a PCI bus, it acquires most run-time drivers during the initialization of the Expansion Manager. " The steps to initialize the driver by the Expansion Manager are then detailed. The final steps in the process are to call the device driver Open routine. These steps are very MacOS specific.

***This seems wrong. I need to look further but why OPEN the driver at boot time?*

****For PCI drivers (NuBus too?) the driver needs to remain available so the Open call must succeed. If the open fails, the driver is removed from the unit table and its space is released. Again, is this right? I just want the device driver to appear in the device tree from whatever its source and then all the device init steps become the same (following this point).*

For NuBus devices, the Slot Manager performs a parallel set of functions. Again the steps are very MacOS specific. For Motherboard devices, the usual path is an explicit PBOpen trap to locate and install a device driver for the device.

Plan A requires us to create a set of services that can walk the Expansion Manager device tree, explore all Slot Resources, find drivers in the System Folder and build up a System Registry from all this data. Then from whatever source, this System registry is populated with device nodes and their properties. One property is a CFM based device driver. The initialization steps for the new Device Manager are as follows:

- The device initialization manager is called.
- The family expert code is called (if there is only one device in a family a single driver may be the "expert")
- The expert code locates all of its devices and reads the required family properties for each device from the System Registry The expert builds up a family specific initialization parameter block for each device from its property lists and calls the device initialization routines with init parameter block.

****TomS. sez we can change init to do this. This init routine follows the NDRV documentation in that is is an NuDriverEntryPoint call with IOCOmmandID,ContentsPtr,Code and Kind parameters. The change is to allow passing in the init parameter block.*

****Here there be dragons. The NDRV documentation about loading/starting a driver specifies that an Open trap starts the process off. The PCI documentation sezs we are doing this at boot time (no user open trap that I can see). If they are both right then BOTH mechanisms need to follow the above steps, providing the init parameters to the driver. OR PCI and non-PCI drivers do need to* *be written differently (see **Plan B**).*

****The NDRV documentation has two INITs one is for CFM and one is the driver init. This is discussed in the document, but an incautious reader may miss this bit of information. How do we clarify these issues. The Init I am talking about is the driver init where we CAN do allocations, and NOT the CFM init.*

With Plan A, device family experts are initialized first, and device initialization occurs only after the family is ready to control devices.

**Plan B**

With Plan B there is no common initialization path for differing types of devices. The plan here is to work with devices once they appear as an entry in the Device Unit Table. The initialization steps in this case are detailed separately within the PCI and NDRV documentation. The only common feature between the initialization sequence for the two kinds of native device drivers is that the device driver ends up within the MacOS Device Unit Table.

For example PCI devices appear in the Expansion Manager device tree and are used and maintained by the Expansion Manager. PCI initialization is documented and the device driver writer must adhere to the PCI documentation to have the device driver appear within the Device Unit Table.

Motherboard devices do not appear in the device tree and are not available through the Expansion Manager, but must by some "standard" MacOS means end up in the Device Unit Table.

The implications of the above architecture is that motherboard devices have hard coded address/register/name/version information within each device driver. Each motherboard device driver uses this information at initialization (or Open) to initialize their device. PCI device drivers now need to match this logic with calls to the Expansion Manager. No expert capability is possible because device drivers are required to contain or obtain what a family expert would provide.

With this architecture, device location remains identical to existing MacOS services. Device clients make PBOpen/Close/Control/Status traps through the Device Manager to request device services.

As an example of device family initialization, the Open Transport device scanning code with this plan would search Device Unit Table Entries for devices with the family name `'OPTP'`. For each located device the Open Transport scanner calls a second? initialize or open routine and registers device driver information within the family data structures.

**Plan C**

Plan C requires that all devices within a system appear in the Expansion Manager device tree. With this model Motherboard and NuBus device drivers must mirror the OpenFirmware PCI driver logic and install device dependent information in the Expansion Manager device tree.

With Plan C, new native non-PCI device drivers must make Expansion Manager calls to *install* information into the device tree. The set of properties installed for each device family must match the properties required of PCI devices within the family. Clearly existing NuBus and motherboard device drivers will not do this work, so a new class of manager needs to be developed to extract Slot Manager available or hard coded system information within individual device drivers and populate the Expansion Manager device tree with these device properties and a driver property. At this point Plan C can match Plan A above.

Plan C means a new backwards compatibility manager to install device information for old devices. In addition to this backwards compatibility manager, Plan C requires all the work of Plan A. In short, Plan C means the Expansion Manager becomes the System Registry for Marconi.

In addition to the additional work required by Plan C, I have to ask: Does the Expansion Manager provide OS neutral non-device specific registry capabilities? This is a requirement for a maintainable/expandable/portable System Registry service.

## System Programming Interfaces (SPIs)

Every device driver writer on the planet needs a set of system services. Most MacOS driver writers have a standard way of handling scheduling, memory management, interrupts and configuration. This section covers changes to the existing mechanisms, and gives the replacement calls. Please note that these are guidelines, for exact calling sequences you will need to refer to the documents listed in the first section of this paper.

### Scheduling routines

The Deferred Task Manager calls:

```
DeferredTaskRef CreateDeferredTask(DeferredTaskPtr,  long contextPtr)
DestroyDeferredTask(DeferredTaskRef)
ScheduleDeferredTask(DeferredTaskRef)
```

are replaced by:

```
QueueSecondaryInterruptHandler(theHandler,theExceptionHandler, p1, p2)
CallSecondaryInterruptHandler(theHandler)
CallSecondaryInterruptHandler2(theHandler, p1, p2)
CallSecondaryInterruptHandler3(theHandler, p1,p2,p3)
CallSecondaryInterruptHandler(theHandler, p1, p2, p3, p4)
```

The Deferred Task Manager maintains a queue of deferred tasks that run when they are enabled by a call to ScheduleDeferredTask. The new mechanisms allow a “deferred task” now known as a “Secondary Interrupt Handler” to be queued or run on the fly. The OS mechanisms used to manage Secondary Interrupts are no longer visible to clients of the scheduling routines.

***** *Should the InterruptHandler routines be wrapped up in Open Transport wrappers? For example timeout and untimeout are exported APIs that use some system mechanism. Should we have Schedule and UnSchedule routines?*

### Interrupt mechanisms

To install interrupt handlers there are a pair of replacement routines.   The new routines:
```
      InstallInterruptHandler(theVector, theHandler)
      RemoveInterruptHandler(theVector, theHandler)
```
replace:
```
      SIntInstall(SQElemPtr, theSlot)
      SIntRemove(SQElemPtr, theSlot)
```

****Documentation Gotcha... What are the real calls? xxxInterruptFunctions ala the "NuKernel ERS" and "Device Manager Changes" or xxxInterruptHandler ala "Integration of Conventional and NuKernel Arch"*

These new routines register or remove interrupt handlers.  See "Integration of Conventional and NuKernel Interrupt Architectures"  and "Device Manager Changes to Support Native Drivers" for more details.

**** See the Issues section under Interrupt Management for Expansion Manager vs. NDRV discussion.*

**** What about direct manipulation of GrandCentral Interrupt vector tables?  Do we say do not do this?  Do we allow "special" device interrupt management.  This type of access is likely to conflict with the NDRV device manager/interrupt manager.  Each device using GC will be affecting this table directly.  What does the PDM class machine have  that corresponds to this?  The right answer is to treat these interrupt vectors the same as IO Register Sets passed from the System Registry into the device init routine and manipulated through SPIs.*

**Resource Location**

Resource location is part of what we are describing in the native device driver scanning and initialization descriptions above.  To remove OS dependencies from within driver code, all resources must be provided to device drivers in a family or globally defined manner.  What this means for device driver writer is:
• Do not use the Resource Manager
• Do not use the file system

Support for both of these mechanisms is not available to drivers after Marconi.
In short and in general:
• Do not use the toolbox

If we do not define and document a mechanism (API) to provide the resources required by drivers, driver writers for new devices will be forced to use mechanisms that are in place today; the Resource Manager and the file system.

****This is  related to the Expert code in the initialization discussion above .  The Expert is responsible for  acquiring the information a driver needs from either the Slot Manager, the Expansion Manager, from Gestalt calls,.  The Expert presents it in some family specific way to the devices.  This needs an **API***

**Memory Management Services**

Do not call:
```
        NewPtr
        NewPtrSys
```

Memory allocation requests should use either a device family specific allocation mechanism or the new Memory Management Services:
```
        PoolAllocateResident
        PoolDeallocate
```

An example of a family specific allocation mechanism is 'allocb' for Streams drivers. Allocb is an exported allocation mechanism provided to all Streams drivers and protocol modules. Allocb uses the appropriate memory management services to its underlying OS.

The new NDRV Memory Management Services are listed in the "Device Manager Changes to Support Native Drivers" and described in detail in the "NuKernel ERS".

**** Do the Pool allocation routines provide the kind of granularity as that of the NewPtr calls? I think the answer here is yes.*

## Issues

### Device Location

A major issue with respect to the PCI/NDRV is the location of devices. Open Transport currently has a requirement that drivers provide their own scanners. One proposal called for an Open Transport standard scanner that would locate all network devices, and load the correct CFM driver modules for each networking device located in the device tree. This scheme is very much in line with the "Expert" notion proposed by Brendan Creane (see Terminology section). This convention needs to be defined for each Device Family. Forcing a "family" property for each driver, and maintaining a list of families would allow a Family Expert to locate all drivers and load them if needed.

To make this work, the device name, vendor, etc, stored in a System Registry as items within a set of properties must exactly match some set of fields within the DriverDescription structure required by NDRV.

### Device Initialization

A major issue with respect to the PCI/NDRV is the initialization of devices. In what order are devices initialized. Who is responsible for device Open calls. How do we clearly separate device initialization and device open calls. Where and when are each appropriate. What services may be invoked at each, and what are the device obligations for close and finalize calls. What is the difference between CFM init calls, and the DoDriverIO command Initialize call?

When is Open called? What services are available at Open? When are resources allocated and deallocated. I would like to have these questions answered so that the example that I am creating at the end of this document will be an example not only for Open Transport, but for every Native Device Driver.

**Expansion Manager is not a System Registry**

"PCI/DLPI Open Transport Integration" lists a set of requirements for the Expansion Manager. For the most part the Expansion manager fulfills those needs for PCI devices. The larger question is what role does the Expansion Manager play within the system. The Expansion Manager is not a System Registry because:

1) Not all devices and services are located within the Expansion Manager device tree. Where is information for non-PCI devices to be located?

2) Human interface issues are not addressed within the Expansion manager. Localizable information must be obtainable given a pointer to a PCI device within the device tree to find an optional family of icons for the module, as well as a user-friendly description of the location of the PCI card, so that users can distinguish between multiple cards of the same type. This issue needs to be resolved before an Open Transport DDK is delivered to developers.

3) The Expansion Manager does device Opens. The Expansion Manager seems to be the SlotManager for PCI devices, not a device Registry. That leaves me with two questions, should the Expansion Manager do device Opens? and why?

4) The Expansion Manager does not provide a notification mechanism to flag changes to the device tree.

5) The Expansion Manager does not handle virtual devices such as an AppleShare volume.

6) The Expansion Manager does not provide a mechanism for catagory wide searches of the device tree.

7) The Expansion Manager uses PCI defined property names. These property names/types are not administered by Apple. How does Apple resolve name conflicts between PCI properties and potential expanded property types.

I claim that a Registry mechanism should have nothing to do with device driver functions. A Registry holds device information and provides device tree APIs. One simple example Registry is MinIO, developed as an interim solution to the full System Registry being developed for Maxwell. We need to keep device location mechanisms separate from the mechanism for adding and removing devices to the System.

**Interrupt Registration SPIs**

In "Designing PCI Cards and Drivers for Power Macintosh" new interrupt mechanisms based on the Expansion Manager are detailed with the following four exposed interface routines:

ExpMgrInstallISR
ExpMgrRemoveISR
ExpMgrInstallVBL

ExpMgrRemoveVBL

How are these routines integrated with the NDRV:

InstallInterruptFunctions
RemoveInterruptFunctions

It appears that the Expansion manager  routines above replace the Slot Manager/VBL Manager routines:

SIntInstall
SIntRemove
SlotVInstall
SlotVRemove

What replaces the VBL functionality within NDRV, or is it even needed? Shouldn't VBL interfaces be confined to the `'DISP'` family of devices?  We need to talk about the way things were, and the way things are, and then the way things are going to be with respect to interrupts.  Having devices fielding interrupts directly could circumvent the NDRV I/O support infrastructure.

**Interrupt Manager**

With the development of NDRV,  a new manager is being defined,  the Interrupt Manager. The list of Interrupt Manager related calls is:

InstallDriver
RemoveDriver
LookupDrivers
Create InterruptSourceTree
(De)ActivateInterruptSourceTree
GetInterruptSourceTreeVector
CreateInterruptSet
DeleteInterruptSet
LookupRootSet
LookupInterruptMemebers
InstallInterruptFunctions
RemoveInterruptFunctions
GetInterruptFunctions

How are these routines used?  Who is  responsible for installing or removing Interrupt Sets?  How are old 68K drivers to appear in the Interrupt Tree?   What role does the Interrupt Manager play within the system?  Where is the Interrupt Manager in the boot sequence, and within the device initalization sequence?  What is the expected relationship between the Interrupt Manager, the Device Manager, the Expansion Manager and a System Registry?

These questions must be answered before a clear story can be presented to device driver writers.

**Interrupt Mechanisms**

The document “Integration of Conventional and NuKernel Interrupt Architectures” provides an overview of the roles of Task/Primary/Secondary levels of interrupts and covers some of the performance issues. This overview is one part of the interrupt mechanism story for device driver writers, but there is no consistent set of interrupt services or mechanisms that apply to all devices. There is no clear picture of how the NDRV interrupt mechanisms relate to the other subsystems/managers available within Marconi.

Given a clear story of what service each manager provides, a native device driver developer needs to know:

- what services are available at which interrupt levels (Primary/Secondary/Task)
- what are the programming guidelines for the interrupt levels
- what is set of standard programming interfaces
- what special programming interfaces are available and why
- interrupt mechanism costs and trade-offs
- what interfaces are prohibitively expensive but may provide backwards compatibility

Device driver writers also need guidelines for interrupt protection mechanisms. We need specific examples for what to do if your code sez:

```
s = splimp();              // Raise interrupt protection mask
do queue stuff
splx(s);                   // Restore interrupt protection mask
```

In the example device driver I am writing, I directly disable my device interrupts to protect my per device queues. I am not running any timers, and all my clients run at task or secondary interrupt time so I think this will work.

An example of how interrupts are used within the new system component Open Transport:

> One of the goals of using the STREAMS architecture is that developers of STREAMS modules can just "drop" their module into the STREAMS environment.
>
> Since STREAMS drivers can call many of the STREAMS functions at interrupt level, it is vital that the STREAMS environment be able to protect itself from data corruption due to reentrancy (the ability to allocate memory at interrupt time is one very critical example). The OpenTransport STREAMS environment also protects itself from data corruption due to multiple processor access to STREAMS data. In order to accomplish this in a portable way, our STREAMS vendor (Mentat supplies us with our STREAMS implementation), shuts off interrupts for short periods of time (typically 6 to 7 68K-equivalent instructions).
>
> It is vital that the device architecture for Marconi and Maxwell provide this ability for the Open Transport implementation. Since on a typical packet delivery path, interrupts are turned off and on once for each module entered, it is very important for performance that this operation be quick (a mixed-mode switch will probably

cause performance to drop below Mac II levels).

There is no intent to export this capability outside the Open Transport/DLPI environment.

I give this example to illustrate how Classic MacOS mechanisms, shutting off interrupts around critical sections, create a problem with the Native Driver model we are creating. We need to be able to address these issues for internal and external developers. We need to create a set of services or architecture that allows development of device drivers without tying the device drivers to specific implementation details. I think driver writers need to have a very clear picture of how the whole interrupt structure plays together in order to create drivers for Marconi, Maxwell and future releases.

**ASLM Interrupt Routines**

The following APIs are required by ASLM:

```
void EnterNetworkInterrupt(NetworkState*)
```

This function must be called by your interrupt routine BEFORE calling any STREAMS or Open Transport function. The NetworkState* parameter is normally a stack variable. This function call must be paired with a LeaveNetworkInterrupt call that is in the same function at the same block level.

```
void LeaveNetworkInterrupt(NetworkState*)
```

This function must be called by your interrupt routine before you exit it. It must be called at the same block level as the EnterNetworkInterrupt function. These two functions tell the Open Transport infrastructure that we are at interrupt time, and allows it to do any setup required to support calls at interrupt time.

It seems to me that the operating system or Open Transport should provide these services. Drivers are CFM based and should not be exposed to kernel internal ASLM issues.

## Creating Marconi Device Drivers

The native I/O architecture that is defined for Marconi will set a long-lived standard for writing device drivers that MUST be supported into V1. Marconi device drivers are destined to become the standard for new native Macintosh device drivers, because no other viable solution will exist within the Marconi timeframe. Because the definition of the Marconi native I/O architecture has such an impact on V1 and beyond, it is imperative that it's architecture allow Marconi native device drivers to work unmodified, and efficiently under V1.

Successful execution of strategy that allows native device drivers to work portably and effectively across Marconi and V1 depends upon the successful resolution of these following issues:

**Structure your driver into two parts**

Your device driver is a low level piece of operating system software. Part of the task of allowing your driver to be portable to a modern operating system like NuKernel, is to determine which portions of the driver belong in the OS itself and which do not. Remember that code which resides in the OS gains the advantage of fine level control over system facilities like paging or interrupts. But it does so by giving up access to the rich set of high level APIs available to Applications. Your driver should be divided into at least two parts:

• **The main driver** is the body of code which reside in the OS and does all the work of repsonding to the I/O command set– Open(), Close(), Control(), Prime(), etc. This code should be passive in the sense that it makes no assumptions about any particular harware settings or configuration. Any device configuration information– like baud rate settings, your ethernet address, the bit depth of your screen, etc.– should be obtained from the I/O command set as parameters passed to Init(), Open(), or Control(). The main driver should not attempt to actively obtain device configuration information on its own. Typical Toolbox APIs (ResourceMgr, FileMgr, SlotMgr, PRAM utils, etc.) required to obtain such information, will not be available to drivers under V1. Use of these APIs will prevent the driver from being portable to V1.

• **The configuration section** is a supporting piece of software for your driver that doesn't necessarily reside in the OS. The configuration section can take many forms, it can be an INIT or a piece of UI like a CDEV or RDEV, etc. Its primary function is to communicate device configuration information to the device driver using Init(), Open(), or Control(). It has access to APIs and system information that the main driver can't/shouldn't try to access itself. If your need to do something high level – like put up a dialog box, or read resources from a file – you should do it in the configuration section.

**Use the System Programming Interfaces**

The use of the System Programming Interfaces are essential to the portability of your device driver to V1. These are the set of Programming Interfaces for device drivers that are guaranteed to be common across operating system releases. When writing the main driver section of your device driver, NEVER make Toolbox API calls. Instead, use corresponding System Programming Interfaces. You will find that these sets of calls allows you to more naturally deal with device driver issues than the Toolbox API, which was intended for Applications. If you find that functionality you depended on in the Toolbox has been removed from the set of SPIs provided, either perfrom that function in the configuration section of your driver, or provide Bill Bruffy feedback... maybe this function was mistakenly omitted

**Use the System Registry**
The system registry provides a unified way of identifying or obtaining information on many system resources – not just devices. The system registry will be key to implementing several important features necessary for the native I/O architecture:

• Effective driver replacement / overloading capability (that allows you, or 3rd parties, to release updates to drivers that shipped with bugs).

• Dynamic driver loading / unloading. A System Registry provides a dynamic and flexible environment for identifing devices. This type of capability will be necessary for supporting hot swappable PCMCIA cards.

• Simplify your driver writing. You won't have different rules for writing device drivers just because the device is located on the MotherBoard vs. NuBus vs. PCIBus vs. PCMCIA bus, etc.

• Makes System Software more EOM'able. The System Registry will provide the layer of abstration necessary for us to remove incestuous device identification / device information callouts (like the SlotMgr) that prevent our software from being portable to hardware other than our own.

## Open Transport Example

This section is not written. A few outstanding questions:

• Open Transport defines a Port Scanner. This Port Scanner is an "expert" for only one type of device, for example, there is a scanner for the GC/Mace. To be compatible with either Plan A or Plan B above there should be a single scanner that locates all networking devices.

• Right now the device Port Scanner uses kernelCommSys->Register() to place device information into the Open Transport registry. The information installed in the registry is device specific information needed to initialize/open this particular device. This information needs to be passed to the init routine. What mechanism do we have for this within OT or within the general system?
(See the Open Transport API documents for more information about Port Scanners and the "Register" function.)

• Should Open Transport maintain their separate database of device information or use the "standard" registry when/if it is defined.

• Open Transport is ASLM based for streams modules and streams drivers. Devices will be CFM based. What changes are required to the Open Transport device location scheme to move from ASLM to CFM?

• Where do we start to provide PCI Open Transport Open Boot example code?

• Does Open Transport provide Init/Primary/Secondary routines as part of its private API. Do we provide a standard way for all devices to export these routines?

### Open Transport API

In the "PCI/DLPI/NDRV - Open Transport Integration" paper an API is described that calls out two initialization routines, `GetInstallInfo` and `Init`. The routines must be exported by name. The `GetInstallInfo` returns an `install_info` structure pointer that is used to link the device driver to the streams module immediately above the driver.

```
struct install_info* GetInstallInfo()

typedef struct install_info
{
struct streamtab* install_str;      // Streamtab pointer.
UInt32            install_flags;    // StayLoaded & ???
UInt32            install_sqlvl;    // Synchronization level.
char*             install_buddy;    // Shared writer list buddy
UInt32            ref_count;        // set to 0
} install_info;
```

This is an example of the family API described in previous sections. We can export a routine called `GetInstallInfo` or we could export `install_info` directly.

To initialize an Open Transport device driver we need to pass information to the device initialization routine. Using the proposed ParamBlock/ParamSize idea to the device init call (see TomS. note) here is a possible PCIInfo structure for the Open Transport family of devices:

```
typedef struct PCIInfo
{
char              *deviceName;      // PCI ROM device name
char              *deviceAddress;   // N bytes of physical address
unsigned char     *base_address     // device base address
dev_t             deviceID;         // dev_t passed to device open
unsigned short    slotID;           // Backward compatible pseudo slot #
} driver_info;
```

Open Transport maintains module information for all device/streams modules loaded into the system. I do not know what is in the Module information structure. I need to fill this out and explain how it is used. This module information is definitely Open Transport internal. Example Modulecookie structure for the Open Transport family of devices:

```
typedef struct Modulecookie
{
unsigned char     *something;       // Who knows... Get this info.
unsigned char     somethingElse;
} module_info;
```

The next bit of information is out of "PCI/DLPI/NDRV - Open Transport Integration." This section demonstrates how Open Transport thinks about initializing devices. This will change to match the Native Device Driver design, when complete.

```
OTError Init(OTPortRef ref, PCICookie sysCookie, long* moduleCookie,
       Boolean* stayLoaded)
```

> This function is called by the Open Transport driver scanners when the driver is first found. It is NOT required that a driver export this call. If it is not exported, then the driver's "Open" entry point will be called when it is installed into a STREAM and opened.

The “ref” parameter is a unique identifier for the port. It is used by Open Transport as a convenient value to distinguish ports in internal tables. You may or may not need this value.

The “sysCookie” parameter is whatever information the Expansion Manager returns pertaining to your device.

The “stayLoaded” parameter tells Open Transport whether your driver needs to stay loaded at all times, or whether it can be unloaded until needed. It is initialized to false. Set it to true if your driver needs to saty loaded at all times.

The “moduleCookie” parameter is saved by Open Transport, and is retrievable via the *FindPort* APIs when your module is opened (by using the major device number).

The return value tells Open Transport whether an error occurred. If an error is returned, Open Transport will not install your device into the STREAMS device table. However, it will still honor the return value of the “stayLoaded” parameter

**Open Transport Driver Initialization**

We need this section. I can not write this until the larger questions have been answered.

**Open Transport Support Programming Interfaces**
The following is a list of standard streams facilities provided as imports to device drivers by Open Transport. These facilities are in addition to the new standard SPIs.

```
allocb()
bcopy()
bzero()
bufcall()
canput()
dupmsg()
enableok()
esballoc()
esbbcall()
flushq()
freemsg()
noenable()
putbq()
putnext()
putq()
qenable()
qreply()
timeout()
unbufcall()
untimeout()
```

For more details about these routines, their parameters and their return values see

“Streams Modules and Drivers” or the Open Transport documentation.

****Functions missing from the above list are more OS specific. I cover them in the replaced APIs section. Should we standardize these accesses into the Open Transport family of device routines?*