Device Manager Family
Design Document

Macintosh System Software
Apple Computer, Inc.

Table of Contents

1. Related Documents

1. *Next Generation MacOS I/O Architecture*, 3rd Draft (really), Holly Knight, Wayne Meretsky, Alan Mimms, Carl Sutton
2. *Inside Macintosh - Devices*, Addison-Wesley, 1994
3. *Inside Macintosh - Designing PCI Cards and Drivers for Power Macintosh Computers*, Interim Draft, February 10, Apple Computer, 1995
4. *Copland Ngaio Device Manager*, Alan Mimms, 03/23/95
5. *NuKernel ERS*, March 25, 1995, David Harrison, Bill Kinkaid, Jeff Robins, Tom Saulpaugh

2. SUMMARY

This Document outlines the design proposal for the Device Manager for the Copland release of the MacOS. It adheres to the architectural guidelines described by *the Next Generation MacOS I/O Architecture* document.

The orientation of this design has two facets, namely compatibility with the two existing Device Manager implementations and the provision of a new implementation which is in line with the Copland I/O architecture and provides an environment which will service native drivers.

The Copland Device Manager supports both a subset of existing `‘DRVR’`s that adhere to the compatibility assertions described in this document, and `‘ndrv’` drivers that have been written according to the guidelines described in *Designing PCI Cards and Drivers for Power Macintosh Computers* . The Copland Device Manager offers both a migration path for existing device drivers and a home for device drivers that do not have their own Copland I/O family.

3. VISION

The goal of the Copland Device Manager is to provide some level of compatibility with the existing device drivers, both the System 7 style `'DRVR'`s and the PCI Power Macintosh native `'ndrv'`s, while providing a new Copland "family style" environment within which native device drivers can be implemented in the Copland world.

The key aspect of this design is separation of function into layers that provide the necessary level of support. Function that can be provided locally, is provided locally; function that can be provided in user mode, is provided in user mode. The layers exist in domains and kernel messages and queues are used to communicate between the domains. The Device Manager APIs provide the interface between these domains. The domain implementation is hidden from the Device Manager clients via the Device Manager APIs.

Most drivers that provide their services through the Device Manager API as documented in the "Device Manager" chapter of *Inside Macintosh: Devices* (i.e. `'DRVR'`s) and that do not touch real hardware and that do not require that they be operating in kernel mode are supported.[1] Drivers written according to the rules for drivers of family type `'ndrv'` described in *Designing PCI Cards and Drivers for Power Macintosh Computers* are supported in Copland as *plug-in*s to the Device Manager family.

The Copland Device Manager offers both a migration path and a home for device drivers that do not have their own Copland I/O family.

[1] `'DRVR'`s that do their own internal queueing and dispatching will be required to make some changes in order to work within the Copland emulation environment. `'DRVR'`s that insert themselves into the page fault path, disk drivers for example, will not work because they require kernel mode execution. See the System-7 Device Manager section for further discussion of these restrictions.

4. Design Goals

The following list describes the design goals for the Device Manager:

1. Provide compatibility for pre-Copland device drivers written to the System 7 model (*Inside Macintosh - Devices*) that do not “touch” real hardware and that observe the other restrictictions described for `DRVR`s in a Copland environment[2].
2. Provide compatibility for pre-Copland device drivers written to the Marconi model (*Inside Macintosh - Designing PCI Cards and Drivers for Power Macintosh Computers*).
3. Provide a design that will guarantee the correct data context for those `‘DRVR’` drivers that allow the Device Manager to perform the request queuing and dispatching and which make use of asynchronous callback routines.
4. Provide reasonable performance for a variety of I/O styles.
5. Provide a design that’s simple to implement using the Copland infrastructure to provide its services.
6. Provide a scaleable design.
7. Provide a design that provides the best overall system performance and parallelism; avoid the use of mechanisms that serialize the entire machine (e.g. secondary interrupts) wherever possible.
8. Provide a Power PC native implementation for the Device Manager.
9. Isolate the “plug-in” from task knowledge.
10. Provide compatible[3] Family Programming Interfaces (FPIs) and Plug-in Programming Interfaces.
11. Provide the Reliability and Availability aspects of RAS[4].

[2] See the section on “Compatibility Assertions” later in this document.
[3] APIs compatible with the current Device Manager APIs will be provided since the Device Manager is in itself a compatibility mechanism that is expected to go away in subsequent releases of the OS.
[4] Many of the Servicability functions for Copland are still being defined. These will be integrated when available; e.g. the logging facility.

5. OVERVIEW

The general structure of the Device Manager is shown below in Figure 8-1. It illustrates that there are three distinct parts to the Device Manager itself, one that operates in user mode in the Blue World and provides System 7 compatibility, another that provides the interface to the Device Manager Family Server from user-mode programs, and third, the Device Manager Family Server which operates in the kernel space.

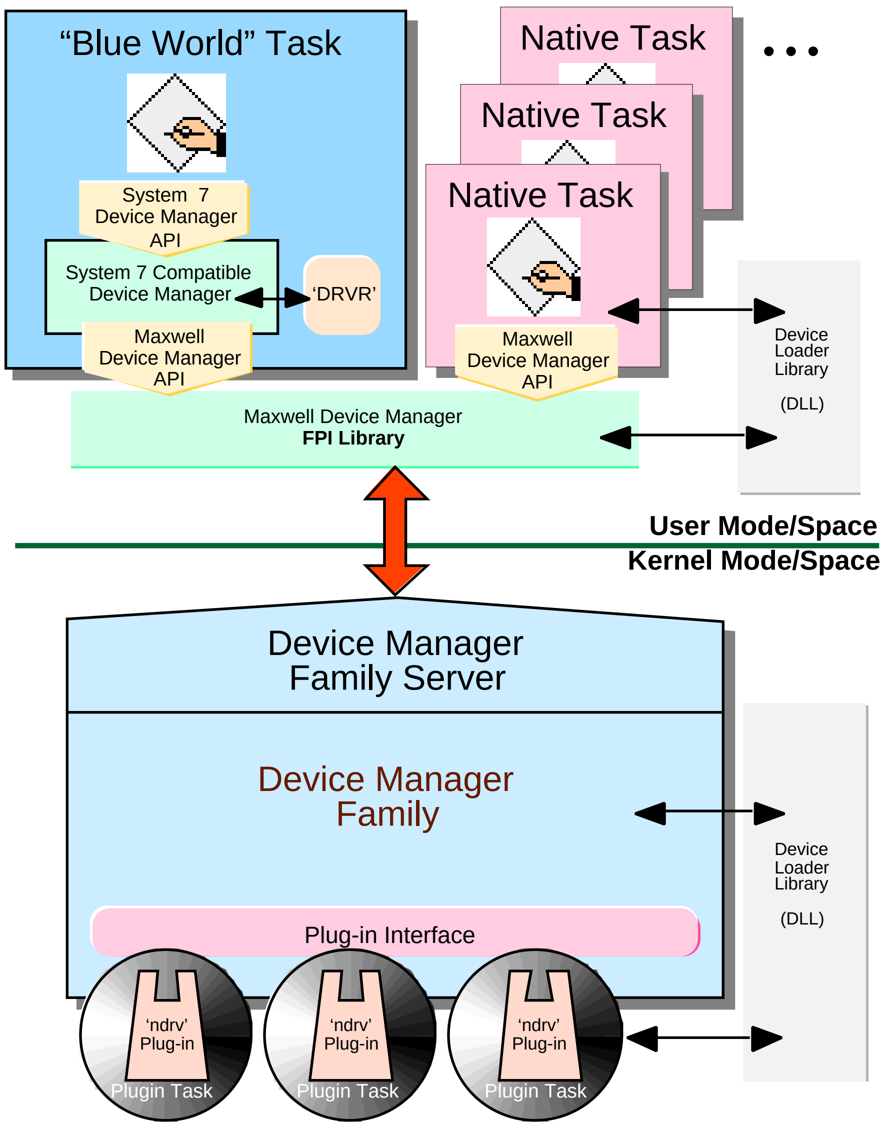
"Blue World" Task
System 7 Device Manager API
System 7 Compatible Device Manager
'DRVR'
Maxwell Device Manager API
Native Task
Native Task
Native Task
• • •
Maxwell Device Manager API
Device Loader Library (DLL)
Maxwell Device Manager FPI Library
User Mode/Space
Kernel Mode/Space
Device Manager Family Server
Device Manager Family
Device Loader Library (DLL)
Plug-in Interface
'ndrv' Plug-in
Plugin Task
'ndrv' Plug-in
Plugin Task
'ndrv' Plug-in
Plugin Task

Figure 8.1 General View Device Manager I/O Family

Each of these provides a programming interface as does the Device Manager Family. [The APIs have been seperated from the libraries in the picture in order to indicate that the libraries provide functions in addition to the direct support of the API requests.] The "Blue World" applications communicate to the upper half of the Device Manager using the System 7 Device Manager API (i.e. that documented in *Inside Macintosh: Device*

Manager). This component of the Device Manager operates in user mode and either directly drives compatible System 7 `'DRVR'`s in the “Blue” space or talks to the Device Manager Family Server using the Device Manager FPI Library calls. This compatibility layer converts old function calls to new ones. There is a performance penalty to be incurred using this path to a native driver.

Native applications use the Device Manager FPI calls directly to communicate to the Device Manager Family Server. The FPI calls result in kernel messages to the Device Manager Family Server which operates in kernel space.

The Device Loader Library (DLL) is used by the Device Manager and other Families to locate, match, install and remove native drivers. There are currently a number of DLL calls that refer to the Unit Table. These calls will be subsumed and implemented by the Device Manager[5]. See the DLL section for more information.

While the Device Manager has its own activation model and set of services, it is not tuned to the needs of any one particular type of driver. Although it’s APIs may be more restrictive than APIs designed specifically for some particular device, the Device Manager offers both a migration path for drivers that have been converted to run native on Copland but have not provided their own families, and a home to those devices that do not require or cannot justify their own family implementation.

The Device Manager family offers a compromise. As a result, the Device Manager plug-ins are likely to be quite different from one another rather than having monolithic characteristics like a video family might for example.

[5] The entry points will still be exported by the DLL but the Device Manager will actually implement the functions and the DLL will call it.

6. DEVICE MANAGER COMPATIBILITY

The Device Manager provides services for *generic drivers*, that is, drivers written to the specifications in *Inside Macintosh: Devices* (`'DRVR'`s) with the restrictions discussed below. Drivers written to the specifications in *Designing PCI Cards and Drivers for Power Macintosh Computers* (`'ndrv'`s) are also supported.

The Device Manager provides three different external programming interfaces:

1. The System-7 Device Manager API.
2. The Device Manager Family Programming Interface (FPI).
3. The Device Manager Family Plugin Programming Interface.

The System-7 Device Manager API is compatible with that defined in *Inside Macintosh: Devices* in the Device Manager chapter. The Plugin Programming API is the API described in *Designing PCI Cards and Drivers for Power Macintosh Computers* in the Writing Native Drivers section. The Device Manager Family API is a new API used by both the System-7 API and native applications to communicate with the Device Manager Family Server. It is not a compatibility API.

6.1. THE SYSTEM-7 DEVICE MANAGER

The System-7 Device Manager support is compatible with that defined in *Inside Macintosh: Devices* in the Device Manager chapter. These "older" style operations are only supported for 68k code running in emulation mode in the "Blue" world. The following sections describe the externals of that interface which are supported by the Copland Device Manager.

This driver interface, (not the API), is only supported from within the "Blue" world and only for device drivers of type `'DRVR'` that are compatible. Such device drivers are not plug-ins; they run in user mode outside the Copland I/O system and can exist only within a world which supports full ToolBox and `WaitNextEvent` access (i.e. the "Blue" world). Any drivers which require execution in kernel mode (because they touch real hardware, are in the page-fault path, &etc) will not work in this environment and will need to be converted to an `'ndrv'` or other type of native driver. In addition, `'DRVR'`s which perform their own queuing and dispatching will need to provide their own version of "thunk" support (described in a later section of this document).

Examples of drivers that fit in the supported category include:

- RAM disks
- desk accessories
- print drivers
- the Open Transport backwardly compatible protocol modules

6.1.1. THE SYSTEM-7 DEVICE MANAGER API

The System-7 Device Manager API is shown in Table 9-1 below which lists the System 7 high-level, low-level and Driver routines that are supported. Both the "High Level" and "Low Level" APIs will provide an access path to both System 7 style `'DRVR'`s and native `'ndrv'`s.

High-Level	Low Level	Driver Routines
`OpenDriver`	`PBOpen`	`Open`
`CloseDriver`	`PBClose`	`Close`
`FSRead`	`PBRead`	`Prime`
`FSWrite`	`PBWrite`	`Prime`
`Control`	`PBControl`	`Control`
`Status`	`PBStatus`	`Status`
`KillIO`	`PBKillIO`	`Control`

Table 9-1 System-7 Device Manager I/O functions and Driver Routines

The prototypes are documented in *Inside Macintosh: Devices* and have not changed:

```
pascal OSErr OpenDriver (ConstStr255Param name, short *drvrRefNum);
pascal OSErr CloseDriver (short refNum);
pascal OSErr FSRead (short refNum, long *count, void *buffPtr);
pascal OSErr FSWrite (short refNum, long *count,const void *buffPtr);
pascal OSErr Control (short refNum, short csCode,
                     const void *csParamPtr);
pascal OSErr Status (short refNum,short csCode,void *csParamPtr);
pascal OSErr KillIO (short refNum);

pascal OSErr PBOpen (ParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBClose (ParmBlkPtr paramBlock,Boolean async);
pascal OSErr PBRead(ParmBlkPtr paramBlock,Boolean async);
pascal OSErr PBWrite(ParmBlkPtr paramBlock,Boolean async);
pascal OSErr PBControl (ParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBStatus (ParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBKillIO (ParmBlkPtr paramBlock, Boolean async);

pascal DCtlHandle GetDCtlEntry (short refNum);
pascal OSErr DriverInstall (DRVRHeaderPtr drvrPtr, short refNum);
pascal OSErr DriverInstallReserveMem (DRVRHeaderPtr drvrPtr,
                                 short refNum);
pascal OSErr DriverRemove (short refNum);
```

The `DrvrInstall()` call is no longer supported (it never worked correctly); `DrvrRemove` has been renamed `DriverRemove` and will be changed to a `DriverRemove` call via a macro.

6.1.2. The System-7 Device Manager Data Structures

The Device Manager maintains a data structure called a device control entry (DCE) for each open driver. Each open driver may be referred to by a single DCE or by many DCEs. Amongst other information, the DCE contains a handle or pointer to the device driver code and a pointer to the I/O queue. The `AuxDCE` supersedes the original `DCtlEntry` data type and provides additional fields for the slot manager.

```
typedef struct AuxDCE
{
   Ptr          dCtlDriver;       // ptr/handle to driver
   short        dCtlFlags;        // flags: ability/state of driver
   QHdr         dCtlQHdr;         // I/O queue header
   long         dCtlPosition;     // current rd/write byte posn
   Handle       dCtlStorage;      // driver global data ptr (if req'd)
   short        dCtlRefNum;       // driver reference number
   long         dCtlCurTicks;     // for internal use only
   WindowPtr    dCtlWindow;       // -> driver's window (DAs)
   short        dCtlDelay;        // ticks between periodic actions
   short        dCtlEMask;        // DA event mask
   short        dCtlMenu;         // DA menu ID
   Sint8        dCtlSlot;         // slot number for a slot device
   Sint8        dCtlSlotId;       // sResource directory ID
   long         dCtlDevBase;      // slot device base address
   Ptr          dCtlOwner;        // Reserved - must be 0
   Sint8        dCtlExtDev;       // slot device external device ID
   Sint8        fillByte;         // Reserved
   UInt32       dCtlNodeID;
};
```

The Unit Table is used to organize and keep track of DCEs. It is an array of handles that point to the DCEs of installed device drivers. The location of the DCE in the Unit Table is called the driver's *unit number*. If the handle at a given number is `nil`, no DCE is installed in that position.

When a device driver is opened, the Device Manager returns a *driver reference number* for the driver which is the one's complement of the unit number (~unit number).

Two global variables are used in conjunction with this scheme. `UTableBase` points to the Unit Table while `UnitNtryCnt` describes the size of the table (i.e. the number of entries in the table). The first 48 entries (entry number 0 through 47 / reference number -1 through -48) are reserved by Apple. The remaining items are documented as available for slot devices and other drivers.

The Device Manager maintains an I/O queue for each open device driver. It is rooted in the drivers DCE. The I/O queue element at the head of the queue is the one currently being processed. The other elements on the queue (if any) are the "pending" I/O requests - those that the Device Manager has received but not yet passed on to the driver.

The Device Manager supports three types of I/O requests: asynchronous, synchronous and immediate. (The terms synchronous and asynchronous as used by the System 7 Device Manager refer to how the Device Manager queues I/O requests. How a device driver processes a request, synchronously or asynchronously, depends on the design of the driver. When making a synchronous request to a device driver, the Device Manager waits for the driver to complete the request regardless of whether the driver handles the request synchronously or asynchronously.)

· **asynchronous requests**: the Device Manager places the request at the end of the driver request queue, call the driver to process the request if it is not busy, and returns control to the application. The application can then continue its processing as it wishes. The Device Manager provides mechanisms to determine when the driver has completed the request.

- **synchronous requests**: the Device Manager places the request at the end of the driver request queue and waits until all requests in the queue, including this one, have been completed by the driver before returning control to the application.
- **immediate request**: the Device Manager sends immediate requests directly to the device driver, bypassing the queue, and returns control back to the application when the request is complete.

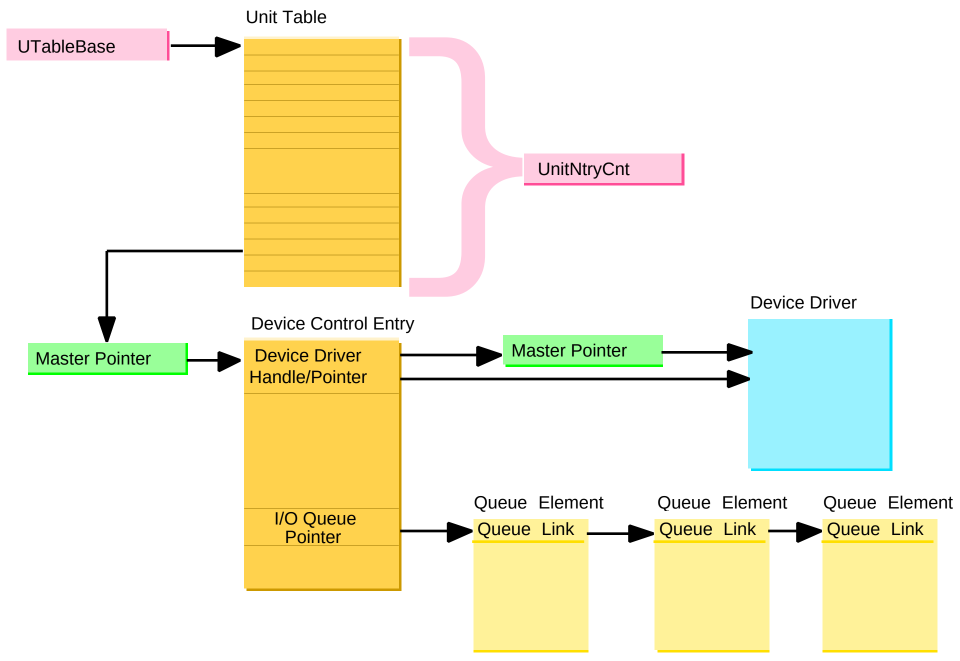

Figure 9-1 System 7 Device Manager Data Structures

The parameter blocks used by the System 7 Device Manager API are `IOParam` and `CntrlParam` as shown below. `IOParam` is used by `PBOpen`, `PBClose`, `PBRead` and `PBWrite` while `CntlParam` is used by `PBControl`, `PBStatus` and `PBKillIO`.

The `ioResult` field is used to communicate the success or failure of the request to the application. For synchronous requests, the value of `ioResult` is set when the Device Manager returns to the application. For asynchronous requests, the value of `ioResult` is set to `ioInProgress` (1) when the request is queued by the Device Manager and is set to the actual result when the driver indicates that it has completed the request (`noErr` (0) if successful or a negative value if the request failed).

Asynchronous callers can also provide a pointer to a completion routine in the `ioCompletion` field of the parameter block which the Device Manager will call when the driver indicates that the requested operation is complete.

```
struct IOParam
{
   QElemPtr          qLink;            // -> next queue entry
   short             qType;            // queue type
   short             ioTrap;           // routine trap
   Ptr               ioCmdAddr;        // function ptr
   IOCompletionUPP   ioCompletion;     // -> completion routine
   OSErr             ioResult;         // result code
   StringPtr         ioNamePtr;        // -> driver name
   short             ioVRefNum;        // vol ref/drive number
   short             ioRefNum;         // driver reference number
   SInt8             ioVersNum;        // not used by Device Manager
   SInt8             ioPermssn;        // read/write permission
   Ptr               ioMisc;           // not used by Device Manager
   Ptr               ioBuffer;         // -> data buffer
   long              ioReqCount;       // # of bytes requested
   long              ioActCount;       // actual # of bytes completed
   short             ioPosMode;        // positioning mode
   long              ioPosOffset;      // positioning offset
};

struct CntrlParam
{
   QElemPtr          qLink;            // -> next queue entry
   short             qType;            // queue type
   short             ioTrap;           // routine trap
   Ptr               ioCmdAddr;        // function ptr
   IOCompletionUPP   ioCompletion;     // -> completion routine
   OSErr             ioResult;         // result code
   StringPtr         ioNamePtr;        // -> driver name
   short             ioVRefNum;        // vol ref/drive number
   short             ioCRefNum;        // driver reference number
   short             csCode;           // control/status request type
   short             csParam[11];      // control/status information
};
```

6.2. COMPATIBILITY IN THE BLUE WORLD

The Copland “Blue” address space provides some new compatibility challenges for the Device Manager. Each process in the “Blue” address space has its own emulator data context. When the Device Manager receives a request from a client, it is operating in the context of that client. However, there are two conditions under which the Device Manager can get control in an arbitrary context:

1. When the driver returns control through IODone after completing an asynchronous request. The ioCompletion routine must be run by the Device Manager at this point in time. The context in which the I/O completed may not be the same context that originally made the request.

2. For synchronous I/O completion, the old Device Manager implementation used busy waiting to await the completion of a synchronous request. This is not an acceptable solution in the Copland world.

3. When the Device Manager has completed IODone processing for the current request, it looks to see if any pending requests are enqueued for this driver. If there are, it begins processing the next request. The context for this request is not necessarily the same as that for the previous request (i.e. different "Blue" processes may have queued requests for this driver). Note that the queued request can be either synchronous or asynchronous at this point (although if it's synchronous, it will be the last one on the queue).

6.2.1. "THUNKING"[6] THE CORRECT DATA CONTEXT

Problems 1 and 3 above can be corrected by introducing a function into the execution path that guarantees that the correct data context (i.e. TOC) will be associated with the current execution just by calling the function. This "thunk" then just calls through to the code that was called in the old execution path. When the thunk returns, the previous TOC is restored; no context related work should be done under this TOC.

In order to have access to the appropriate function/TOC when it is required, at the time the request is queued a `ThunkTableEntry` is allocated from a table that is maintained in globally accessible and system owned memory by the Device Manager (the `thunkTable).` For all requests, the key by which the table entry can be found later is the `IOParam`/`CntrlParam` parameter block address. This must be unique as long as the request is outstanding[7]. In addition, in order to solve problem 3 above, a `ProcPtr` to the thunk routine to call the driver when dispatching a new request is also stored (this causes the compiler and linker to conspire to point the code at glue code which will cause the correct TOC to be loaded when the routine is called). The remaining information stored in the table entries varies depending on whether the request is synchronous or asynchronous.

For asynchronous requests, the `ioCompletion` routine specified by the caller in the `IOParam`/`CntrlParam` parameter block is also stored. The `ioCompletion` routine pointer is then replaced by a pointer to a UPP for the I/O completion thunk. For synchronous calls, the Device Manager still substitutes its own I/O completion routine. The callers ioCompletion specification is ignored and forced to zero before returning in the synchronous case. Figute 9.2 illustrates this process when the Device Manager receives a non-immediate I/O request. (The two numbered dark circles represent the two mixed-mode switches that occur here, one when the application calls the Device Manager and a second when the Device Manager returns to the application; the Device Manager is native code.)

[6] Alan Mimms first "coined" the term "thunk" with reference to the mechanism being used by the Device Manager to switch execution into the correct data context. The term "thunk" is traditionally used by compiler writers to describe a routine that calculates and returns the address of an actual parameter corresponding to a formal parameter called by name. Our thunk causes the address of the correct TOC context to be associated with a particular execution of a function. The term was originally coined by P. Z. Ingerman in *Thunks*, CACM, Jan 1961.

[7] The Device Manager could check for this uniqueness when it queues the new request, but there are too many other ways for the programmer to potentially misuse the parameter block while it is queued. It is not wothwhile to perform such a check given the overhead and the potential for other for programmer errors which cannot be detected early.

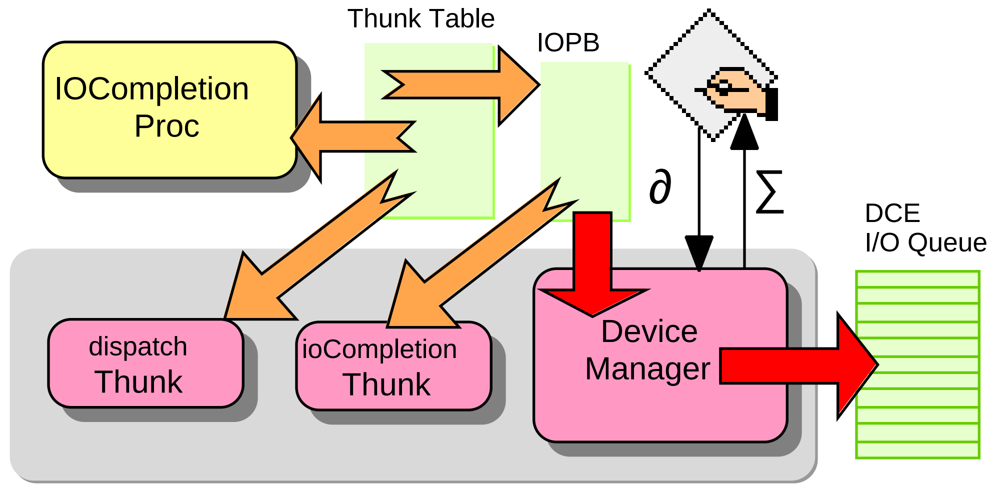

Figure 9.2 Processing Synchronous/Asynchronous I/O Requests

When the [next] entry enqueued to the `dCtlQHdr` queue in the `DCE` is to be dispatched, the Device Manager calls the dispatch thunk stored in the thunk table for that parameter block. This causes the correct data context to be loaded for that request.

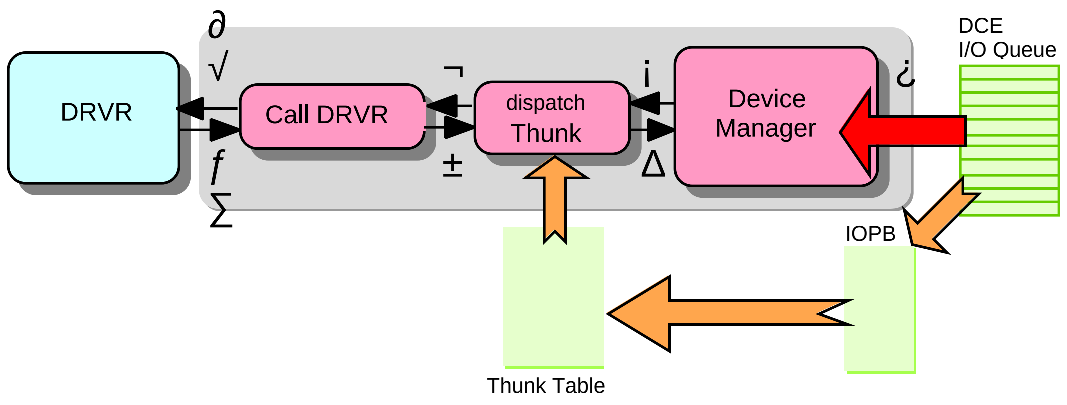

Figure 9.3 Thunking the Dispatch Queue

Figure 9.3 illustrates this. (The transparent circled numbers represent the call sequence; the dark numbered circles represent the mixed-mode switches.) The queue entry is sent to the `DRVR` via the dispatch thunk (which was created when the request was originally queued). This causes a mixed-mode switch. When the `DRVR` returns to the Device Manager through the thunk, it causes another mixed-mode switch.

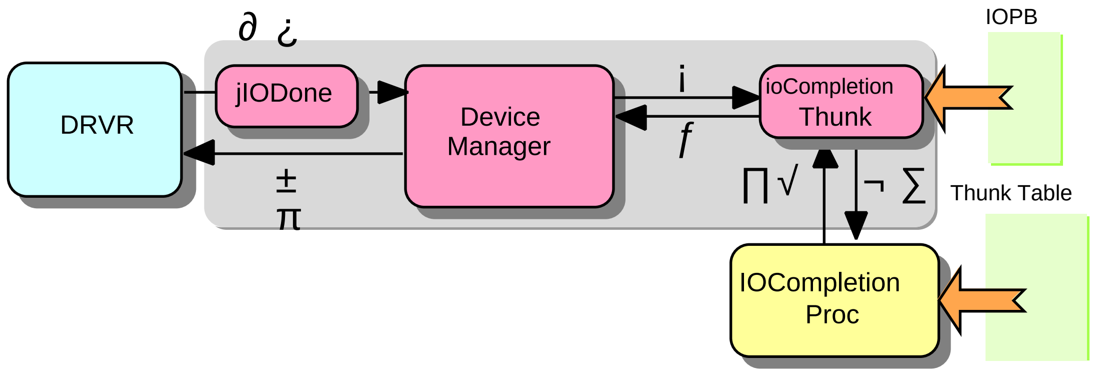

Figure 9.4 Thunking an I/O Completion

The `DRVR` indicates that it has completed an I/O request by calling back to the Device Manager through `jIODone` as shown in Figure 9.4. The IODone vector points to the native Device Manager code. This causes the first mixed-mode switch. The Device Manager then calls the ioCompletion thunk which it created when it queued the request (this is the one pointed to by the I/O parameter block that is being completed). This causes the correct data context (TOC) to be loaded. The thunk then calls the "real" `ioCompletion` routine (saved in the thunk table entry) if any. This causes the second mixed-mode switch. The user-specified `ioCompletion` routine eventually returns to the thunk (causing the third mixed-mode switch) which in turn returns to the Device Manager. The Device Manager then examines the DCE queue. If it's empty, the Device Manager returns to the `DRVR` causing a fourth mixed-mode switch. If it is not empty, the process continues as shown in Figure 9.3 with the next I/O request to be processed.

Drivers that perform their own queueing and do not call through `IODone` (i.e. they perform their own IODone processing) must still call the I/O completion routine specified by the parameter block. The `ioCompletion` thunking will function properly in this case as well. Such drivers will be responsible for their own dispatch thunking however.

The operation and further use of the table for synchronous requests is discussed in the next section.

6.2.2. Handling Synchronous Wait

In order to avoid busy waiting, the Device Manager uses Event Flags and `WaitForEvents` to cause the requester's thread to wait until the request is signaled complete by the driver. The event mask and event group ID are saved in the thunk table entry associated with the current parameter block. In order to wake the thread, an `ioCompletion` routine is specified in the parameter block by the Device Manager before the driver is called so that the Device Manager is assured of getting control back when the I/O is completed. (As a side effect, this I/O completion thunk ensures that the Device Manager code will always run in the correct data context for the request regardless of the context in which the driver returns control to us.) The synchronous I/O completion routine performs a `SetEvents` on the event flag being waited on by the main thread and everything eventually unfolds as it should.

6.2.3. Compatibility Assertions

The following is a list of assertions that define the level of compatibility provided by this implementation of the Device Manager.

- `DRVR`s that provide their own queueing and dispatching must also provide their own dispatch thunking.
- `DRVR`s that insert themselves in the page fault path are not supported.
- `DRVR`s that require kernel mode for any reason are not supported.
- `DRVR`s must either call the device manager back via `jIODone` or call the I/O completion routine directly irrespective of whether the request is synchronous or asynchronous.
- Client specified I/O completion routines are not supported except for those requests that can be and are issued asynchronously; i.e. the device manager will ignore any completion routine specified in the parameter block for either an immediate or a synchronous request.
- Busy waiting on the I/O parameter block by the application is not supported; the application must use an I/O completion routine to find out about the completion of asynchronous requests.
- The Device Manager does not support patches to the `jSyncWait` vector and does not run any routine specified there.
- The client application cannot free, reuse or otherwise modify the I/O parameter block used to make a Device Manager requests until the Device Manager has indicated that it has finished processing the request.

`Data`

6.3. Compatibility Issues With “Hidden” Pointers

Currently some users of the System-7 Device Manager API provide themselves with data pointers to client data and/or callback pointers to private routines inside the control block passed to the driver on control and/or status calls. This may also occur for non-Blue clients of `ndrv`s.The data areas are no longer available to native drivers when they execute because their context is that of the kernel rather than the client. In addition, client callback routines have the same problems as Device Manager callback routines do: they must be “thunked” in order to ensure execution in the correct data context. However, since the Device Manager knows nothing about these pointers, it can do nothing to assist to client.

The solution is to allow a third party to register interest in a particular driver and insert itself in the Device Manager processing path for all requests. The filter proc receives control both when the original request is queued by the Device Manager and again when the ioCompletion thunk is called but before the asynchronous completion routine is called. The specified callback routine is responsible for doing the “right thing”.

```
OSStatus
RegisterDriverFilterProc (DriverRefNum      refNum,
                          CFragConnectionID fragConnID);
OSStatus
UnRegisterDriverFilterProc (DriverRefNum      refNum,
                            CFragConnectionID fragConnID);

typedef enum DMFilterCallType
{
  DMFilterBegin,
  DMFilterComplete
} DMFilterCallType;
```

The filter proc is defined as:

```
OSStatus
DMDriverFilterProc (DriverRefNum     refNum,
                    DMFilterCallType callType);
                    ParmBlkPtr       paramBlock);
```

The registered code fragment will be loaded into the kernel context by the Device Manager. It must export the DMDriverFilterProc entry point[8].

Since this facility needs to be available to both Blue and non-Blue clients of ‘ndrv’s, the function must be implemented in the server side of the Device Manager. In order to allow access to data areas in the current address space, the filter will need to be called from the accept function when a request is sent. This means that the filter function must be native PPC code and will run in supervisor mode which will allow it to perform any data mapping or copying that is required and to save away any other information that is required. On the return trip, the code will run in user mode.

This requires changes, but not to the driver or the client and the software could be provided by anyone to do this job.

6.4. IOCommandIsComplete and the DSL

The `IOCommandIsComplete` call has been implemented in the Driver Services Library (DSL). This net effect of this is that all existing `ndrv`’s link with the DSL in order to service the Device Manager specific I/O request completion. This means that clients like the Video Family that want to continue using existing `ndrv`’s written to handle video will break when the `ndrv`’s call `IOCommandIsComplete` since the code has no context to know which family to call back (and assumes that this is a Device Manager I/O completion).

The `IOCommandIsComplete` call is defined as follows:

```
OSErr
IOCommandIsComplete (IOCommandID  theID,      // completing command
                     OSErr        theResult); // status for IOPB
```

[8] The exact details of this are still being worked out and the interfaces are not shown in the implementation sections which follow. This section needs further investigation and definition.

The ID passed back by `IOCommandIsComplete` is the value originally passed in to DoDriverIO both via the `IOCommandID` parameter and in the `ioCmdAddr` field of the IOPB.

The `DoDriverIO` call is defined as follows:

```
OSErr
DoDriverIO (AddressSpaceID    spaceID,
            IOCommandID       ID,
            IOCommandContents contents,
            IOCommandCode     command,
            IOCommandKind     kind);
```

In order to fix this problem in such a way that existing ndrv’s (especially those that exist on video boards) do not have to be changed, a new call to a new DSL routine will be required by any family which wants to use ndrv’s (and the defined interface) including the Device Manager. The rationale is that since the ID returned by the driver is the only context available to `IOCommandIsComplete`, it must be used to associate the call with the originator. This is accomplished by the originator “registering” each request to the driver `DoDriverIO` entry point first with the `RegisterDoDriverIO` call and using the `ID` that it returns as the `ID` specified in the `DoDriverIO` call.

The `RegisterDoDriverIO` call is defined as follows:

```
OSErr
RegisterDoDriverIO (IOCommandID           *ID,
                    DriverEntryPointPtr   returnAddr);

ID             specifies the ID to be used on the associated
               DoDriverIO call.

returnAddr     specifies the return address to the family code
               to be called from IOCommandIsComplete. The specified
               address will be called as a function defined exactly
               like the IOCommandIsComplete function definition.
```

The family function that calls `RegisterDoDriverIO` is required to save any information that it needs to in order to associate the ID returned by `RegisterDoDriverIO` with the I/O request. It must also place the ID in the `ioCmdAddr` field of the IOPB and save the previous contents if required.

Figure 9.5 illustrates the revised call sequence. First the new call to `RegisterDoDriverIO` is made specifying the return address at which the family code is to be called when `IOCommandIsComplete` is called back by the driver for this request. `RegisterDoDriverIO` creates a new unique id and hashs the information about the id and the return address into its internal hash table and returns the id to be used when the `DoDriverIO` call is made. `DoDriverIO` is then called to request that the driver perform the I/O requests specified in the parameter block. When the driver has completed the non-immediate request, it calls the DSL function `IOCommandIsComplete` which then looks up the id in its hash table and calls back the family code at the specified address exactly as if it were the `IOCommandIsComplete` function.

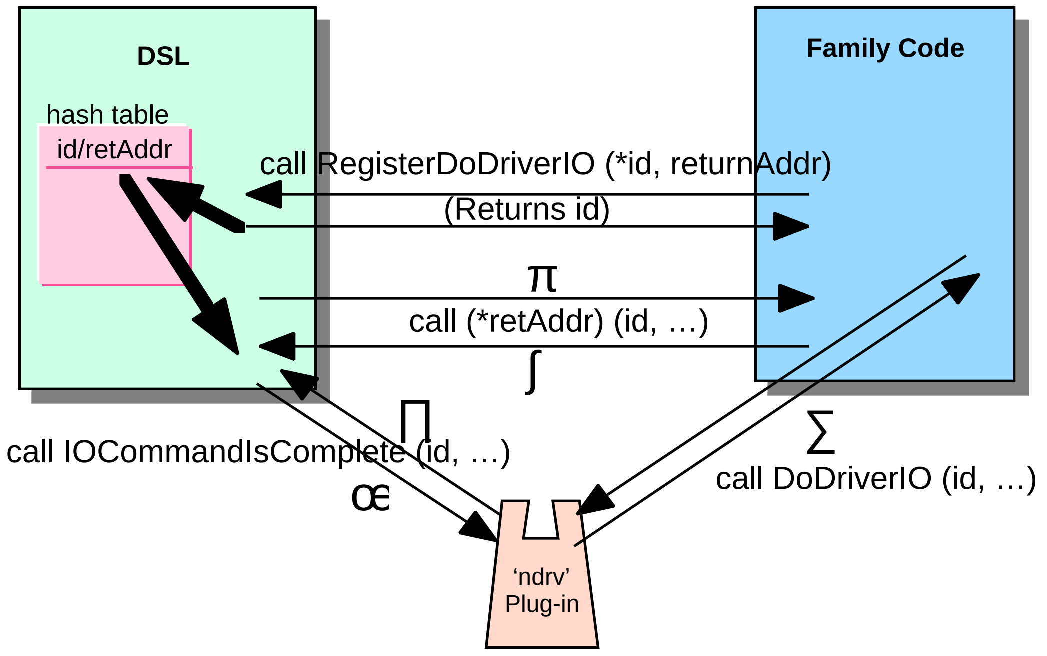

Figure 9.5 RegisterDoDriverIO, DoDriverIO and IOCommandIsComplete Call Flows

6.5. Coherency and the System-7 Unit Table

The Unit Table as described for the System-7 world is required for compatibility. However, it is not the same information that is required by the Copland implementation. Therefore two different representations of "a table that references drivers" exist, one for each implementation. Since native device drivers must be available to clients in the "Blue" world, the "Blue" world Unit Table must be kept in synch with the native Device Manager plug-in table (see the Native Activation Model data structures section) so that a Unit Table reference number can be used to access a driver in the kernel space..[9]

6.5.1. System-7 `'DRVR'` Unit Table Update Methods

There are currently two ways that the System-7 Unit Table is updated when installing `'DRVR'` drivers. This information is documented in *Inside Macintosh: Devices*. The first is through the driver install and remove routines provided by the API[10]:

[9] It is assumed throughout that `'DRVR'`s will not be accessible outside of the Blue address space..

[10] `DRVRInstall` and `DRVRRemove` have been replaced by `DriverInstall` and `DriverRemove` respectively.

```
pascal OSErr DriverInstall (DRVRHeaderPtr drvrPtr, short refNum);
pascal OSErr DriverInstallReserveMem (DRVRHeaderPtr drvrPtr,
                                      short refNum);
pascal OSErr DriverRemove (short refNum);
```

These provide a contained means of capturing table updates. Unfortunately, there are several drivers/applications around which do not use the provided API to update the Unit Table but do it directly themselves by scanning the table for a free entry and storing a DCE handle in it or simply clearing an entry already in use. This makes using a common reference number between the two spaces very difficult.

6.5.2. The ‘ndrv’ Driver Unit Table Updates

The Driver Loader Library (DLL) has 12 routines defined in its API that reference or update the System-7 Unit Table (see the Driver Loader Library section below). This interface is documented in *Inside Macintosh - Designing PCI Cards and Drivers for Power Macintosh Computers.* These routines also provide a contained means of updating the Unit Table. However, as explained in the Driver Loader Library section, the Device Manager will have to provide a new user-mode library implementation to allow the updates to be made in both the System-7 Unit Table in “Blue” space and the new plug-in table in kernel space. The updates need to be coordinated as explained above.

6.5.3. ‘ndrv’s Installed by Family Experts

As it “discovers” them, the Device Manager Family Expert will install new ‘ndrv’s into the Device Manager plugin table. The plugin table is not visible to the “Blue” world so all updates must be announced to the Device Manager running in user-mode in the Blue address space making native drivers available to applications running there. In many ways this is the reverse of 9.4.2 above.

6.5.4. Unit Table Updates Using Alias Reference Numbers

In order to allow each environment (i.e. Device Manager Server and “Blue” world Device Manager) to update their tables independently, the “Blue” world implementation will use “alias” driver reference numbers when referencing an ‘ndrv’. This requires explicit communication between the two worlds whenever any of the ‘ndrv’ updates described above occur.

When the “Blue” world Device Manager installs a new driver, if it was a ‘DRVR’, the Unit Table update can proceed and no notification must take place for the Device Manager Server since it will never need to reference a ‘DRVR’. (Direct Unit Table updates by System-7 applications code will only be supported for ‘DRVR’s and no special support needs to be supplied for such actions.)

If it is an ‘ndrv’ being installed, the Unit Table entry and its associated data structures must be allocated and a “private” call must be made to notify the Device Manager Family Server that it needs to install the new ‘ndrv’. This request is synchronous and returns an indication of success. Upon success, a driver reference number to be used by the FPI (i.e., the alias) is also returned. The System-7 Device Manager will then save the Device Manager Server device reference number away (in the AuxDCE since it is not really be used by a driver), and use the “alias” whenever it makes requests of the FPI, reporting its device reference number whenever communicating with the System-7 API client. If the Device Manager Server fails the request, so does the System-7 Device Manager interface (and it removes its Unit Table updates). When an ‘`ndrv`’ is deleted from the Unit Table by a client, an analogous operation takes place.

[Describe API here.]

An update by Device Manager Server must be communicated to the System-7 API Library. This requires that the Library register interest during initialization. When an ‘`ndrv`’ is added or deleted from the Device Manager Server’s Plug-in Table (and it is not as a result of an operation described above), the System-7 Library will be notified to update its Unit Table by scheduling a cooperative interrupt to run. An ID is associated with the driver `refnum` to guarantee that reassignment of the Plugin Table entry by the Device Manager Family Server before the cooperative interrupt is run to delete the corresponding Unit Table entry does not cause a subsequent request to be made to an incorrect plugin.

[Describe API here.]

6.5.5. Request Flows

The following sections describe the request flows for the two installations described above.

6.5.5.1. ‘`ndrv`’ Installed by a Blue World Client

[include flows diagram here]

6.5.5.2. ‘`ndrv`’ Installed by the Family Expert

[include flows diagram here]

6.6. The Native Driver Interface

Native drivers that follow the rules and use the interfaces described in *Designing PCI Cards and Drivers for Power Macintosh Computers: Writing Native Drivers* are supported by the Device manager and are called “generic device drivers”. All native drivers are PowerPC native code in Code Fragment Manager (CFM) container format and must run without access to the ToolBox. For generic drivers on PowerPC platforms, the Device Manager has changed to support PowerPC driver code and to allow drivers to operate concurrently. Generic drivers have a family type of ‘`ndrv`’.

6.6.1. Generic Drivers Imports & Exports

Native drivers must use CFM’s import and export library mechanisms to share code and/or data. The following exports and imports are defined for generic native drivers.

6.6.1.1. Native Driver Data Exports

All native drivers must export a single data symbol that characterizes the driver's functionality and origin: `TheDriverDescription`. Driver description information helps match drivers and devices. It also lets the Device Manager pick the best driver among multiple candidates.

```
struct DriverDescription
{
  OSType             driverDescSignature;  // Signature field
  DriverDescVersion  driverDescVersion;    // Version of this data
  DriverType         driverType;           // Type of Driver
  DriverOSRuntime    driverOSRuntimeInfo;  // OS Runtime Requirements
  DriverOSService    driverServices;       // Driver supported API
};
```

6.6.1.2. Native Driver Code Exports

Native device drivers export a single code entry point, `DoDriverIO`, that handles all Device Manager operations. The device driver can determine which I/O action to perform based on the command code (`Initialize, Finalize, Open, Close, Read, Write, Control, Status, KillIO, Replace,` or `Superseded`) and command kind (`Synchronous, Asynchronous,` or `Immediate`).

```
OSErr DoDriverIO (AddressSpaceID    spaceID,   // address space ID
                  IOCommandID       commandID, // command ID
                  IOCommandContents contents,  // cmd specific pb
                  IOCommandCode     code,      // open/close/etc.
                  IOCommandKind     kind );    // synch/asynch/immed
```

6.6.1.3. Native Driver Imports

Native drivers will import their services from the Driver Services Library (DSL) and the Driver Loader Library (DLL). The Driver Loader automatically links the DSL to each generic driver at load time. New `'ndrv'`s may also link with other native family libraries.

Besides these libraries, the Device Manager exports a new routine called `IOCommandIsComplete`. It is the native driver equivalent of `IODone`. The difference is that while IODone implicitly acts on the head of the Device Manager queue for this driver, the request that is to be completed is specified explicitly to `IOCommandIsComplete`.

```
OSStatus IOCommandIsComplete (CommandID  command,  // command ID
                              OSStatus   results); // value for IOPB
```

6.6.2. CONCURRENT DRIVERS

`'DRVR'`s were defined by the Device Manager to handle only one request at a time. While multiple requests could be pending for a particular driver, the Device Manager only passed the next request to driver when the driver had completed processing the previous request[11]. Native device drivers can now indicate that they are concurrent (i.e. capable of handling more than one request at a time) by setting the `kDriverIsConcurrent` flag in the `driverRuntime` flags in its `DriverDescription`.

The driver must use the `IOCommandIsComplete` service of the Device Manager to indicate it has completed a particular request. The returned status value is used by the Device Manager to update the `result` field of the `IOPB`. The driver should not modify `result` directly as this will be ignored by the Device Manager.

6.6.3. Generic Driver Restrictions

Aside from those differences described above, the rules for generic drivers have changed from those for `'DRVR'`s. The following highlight some of the differences:

- A native drive doesn't have access to its DCE in the Unit Table; in fact, the Unit Table per se does not exist in the context within which the native driver executes.
- `Initialize`, `Finalize`, `Open`, `Close`, `KillIO`, `Replace` and `Superseded` are always immediate commands.
- All native drivers must accept and respond to all command codes; however, an error indicating that the command is not supported may be returned.
- `Initialize` and `Finalize` are the first and last commands a native driver receives. `Open` and `Close` commands connect the driver independently of initialization and finalization.
- CFM will perform CFM initialization and termination calls to the driver when the driver is loaded and unloaded. The CFM initialization call precedes the driver being initialized by the Device Manager.
- Native drivers must be reentrant to the extent that they may be reentered with another request during any call from the driver to `IOCommandIsComplete`.

6.6.4. Installing a Native Device Driver

The boot code will be responsible for finding and installing the initial set of drivers. The Driver Loader Library (DLL) and the family expert will conspire to dynamically select, load, install and remove drivers once the system is up.

6.7. The Driver Loader Library

The Driver Loader library (DLL) provides some routines that work with all families and some that work specifically with the Device Manager family. These routines install, remove and replace entries in the Unit Table.

[11] Some DRVRs have used various workarounds (like doing their own queueing and dispatching) to allow them to process more than one request simultaneously. This causes these same DRVRs some compatibility problems in the current environment. Native drivers can now do this in a sanctioned manner.

The Unit Table per se no longer exists in the Copland kernel environment; it exists in the System-7 compatible “Blue” world. `‘ndrv’`s have been cautioned not to depend on it and to only refer to it and the information it contains through the defined DLL calls. These calls use a `DriverRefNum` and/or a `UnitNumber` to refer to the unit table entries. While the use of `UnitNumber`s will have little meaning in the new environment, the `DriverRefNum` interface will be used to reference a new native table entry. To remain compatibility with the existing ‘`ndrv’`s, the refnum returned will be the ones complement of the actual index value as is the case for previous implementations.

The Device Manager will reimplement the routines listed below for Copland. The externals of these routines will remain the same as is currently defined, but the implementation will be changed to reflect the changes discussed in the section on Unit Table maintenance. There will also be a need for both a user mode callable set of routines for those applications that install `‘ndrv’s` or use `‘ndrv’` information from the Blue address space or native user-mode applications in their own Copland-savvy address space, and a supervisor mode set of routines which are callable from the Device Manager Family Expert.

These entry points will be exported by the Device Manager and called by the DLL implementation (for compatibility) which will re-export them[12].

6.7.1. The Driver Loader Library API for the Device Manager

The following are the functions provided by the DLL will be updated to implement the new method of updating and maintaining the Device Unit Table coherency with the “Blue” world.

```
OSErr InstallDriverFromFragment  (CFragConnectionID fragmentConnID,
                                  RegEntryIDPtr device,
                                  UnitNumber beginningUnit,
                                  UnitNumber endingUnit,
                                  DriverRefNum refNum);

OSErr InstallDriverFromFile     (FSSpecPtr fragmentSpec,
                                 RegEntryIDPtr device,
                                 UnitNumber beginningUnit,
                                 UnitNumber endingUnit,
                                 DriverRefNum refNum);

OSErr InstallDriverFromMemory   (Ptr memory,
                                 long length,
                                 ConstStr63Param fragName,
                                 RegEntryIDPtr device,
                                 UnitNumber beginningUnit,
                                 UnitNumber endingUnit,
                                 DriverRefNum refNum);

OSErr InstallDriverFromDisk      (Ptr theDriverName,
                                  RegEntryIDPtr theDevice,
                                  UnitNumber theBeginningUnit,
```

[12] The supervisor mode library API will have some different function parameters based on the recent interfaces thrash. The updated interface definitions will be included when this thrash is done.

```
                                  UnitNumber theEndingUnit,
                                  DriverRefNum refNum);


OSErr InstallDriverForDevice     (RegEntryIDPtr device,
                                  UnitNumber beginningUnit,
                                  UnitNumber endingUnit,
                                  DriverRefNum refNum);


OSErr GetDriverInformation       (DriverRefNum refNum,
                                  UnitNumber unitNum,
                                  DriverFlags flags,
                                  DriverOpenCount *count,
                                  StringPtr name,
                                  RegEntryID device,
                                  CFragHFSLocator driverLoadLocation,
                                  CFragConnectionID fragmentConnID,
                                  DriverEntryPointPtr *fragmentMain,
                                  DriverDescriptionn *driverDesc);


OSErr OpenInstalledDriver        (DriverRefNum refNum,
                                  Sint8 ioPermission);


OSErr RenameDriver               (DriverRefNum refNum,
                                  StringPtr newDriverName);


OSErr RemoveDriver               (DriverRefNum refNum,
                                  Boolean immediate);



OSErr ReplaceDriverWithFragment  (DriverRefNum refNum,
                                  CFragConnectionID fragmentConnID);


OSErr LookupDrivers              (UnitNumber beginningUnit,
                                  UnitNumber endingUnit,
                                  Boolean emptyUnits,
                                  ItemCount returnedRefNums,
                                  DriverRefNum refNum);


UnitNumber HighestUnitNumber     (void);
```

6.8. PROVIDING ACCESS TO OTHER FAMILIES THROUGH THE SYSTEM-7 DEVICE MANAGER API

There are some clients (i.e. existing applications) of the “classic” Device Manager interface that may in the future require the services of new drivers which have their own family. Rather than having to convert the applications, the other family servers could plug themselves in at the same level as `'DRVR'`s do, redirecting the I/O requests to the appropriate family server.

6.8.1. A Transitional Interface

Since the only clients of the System-7 Device Manager interface are those that live in the "Blue" world, the Device Manager can provide a transitional environment for these other families by having them take an active role in the process. Such a family would install a "shim" 'DRVR' which would be called by the Device Manager as for any other 'DRVR' with some exceptions as noted below. [The implementation assumes that there is currently sufficient information within the parameter blocks to drive the new family since there are no changes to the Device Manager parameter blocks to support this feature.]

Essentially any family that wants to support such an interface installs a "shim" driver in the System-7 Unit Table using the new `DriverInstallShim` interface. The Device Manager then knows these as "shims" and treats them specially allowing them to be concurrent (as it does for native drivers). When a client requests one of these shim drivers, the shim is called immediately with all requests. The shim driver is responsible for interpreting the information in the parameter block and acting accordingly. The shim driver is responsible for implementing the appropriate interface to the target family activation model. This frees the Device Manager from having to special case for certain implementations and provides a means to support other non-Apple implementations that require the same functionality.

The API calls added are:

```
pascal OSErr DriverInstallShim (DRVRHeaderPtr drvrPtr,
                                short *refNum);
```

`drvrPtr`	A pointer to a device driver header
`refNum`	A pointer to where the driver reference number is stored if the shim is successfully installed (`OSErr == noErr`)

The `DriverInstallShim` function does exactly the same job as the `DriverInstall` but also marks the driver as a shim in the `dCtlFlags` so that the Device Manager knows to treat it as concurrent.

```
pascal OSErr DriverRemoveShim (short refNum);
```

`refNum`	The driver reference number

The `DriverRemoveShim` function does exactly the same job as the `DriverRemove` but does not release the driver resource.

Because the driver is treated as concurrent, it will have to notify the Device Manager about which request is being completed and about the success of failure of that request. The Device Manager exports an entry point modelled on the `IOCommandIsComplete` interface for this purpose.

```
OSErr                                        // results from DM
IOShimCommandIsComplete (ParmBlkPtr thePB,    // -> parameter block
                         OSErr      results); // value for IOPB
```

`thePB`	is a pointer to the parameter block that represents the completed request
`results`	is the success or failure indication to be returned to the original

7. REQUESTOR

THE DEVICE MANAGER ACTIVATION MODEL

The Copland Device Manager uses the task-per-plugin activation model. This model is a compromise; it is used because the processing of I/O requests can vary widely among the plug-ins. The plug-in however, is insulated from microkernel tasking mechanisms and from synchronization issues that result from system resource contention and multiple client requests to a single plug-in. Both a user mode and a kernel mode FPI library are provided.

7.1. ACTIVATION MODEL OVERVIEW

The Copland Device Manager FPI server is an accept function that presents data to an event-based loop. The FPI server receives requests from calling clients and passes those requests to the family plug-ins. The FPI server is responsible for making the data associated with the request available to the family which in turn makes it available to the plug-in that services the request.

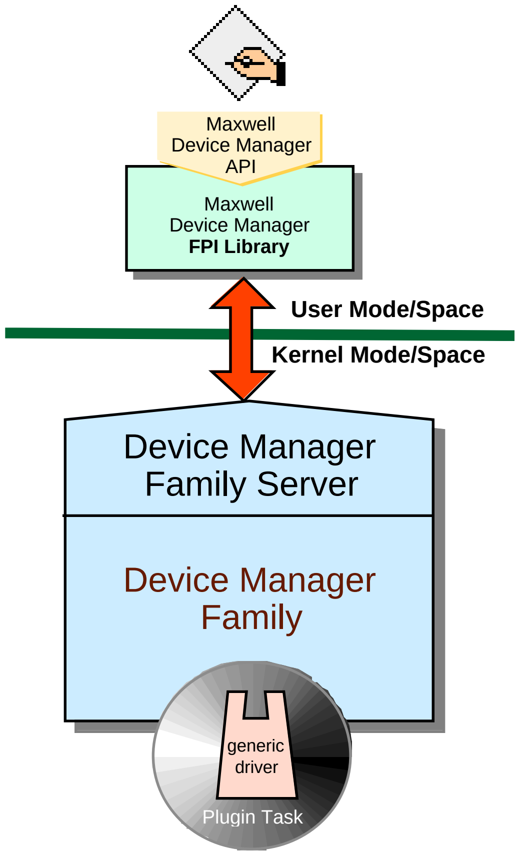

Figure 10.1 Copland Native Device Manager Activation Model

The family creates one task for each family plug-in. The tasks act as wrappers for the plug-ins - all tasking knowledge is located in the family code.

When the plug-in receives a service request, via its `DoDriverIO` entry point, the task calls the plug-ins entry point, waits for the plug-ins' response, and then responds to the service request. The plug-in performs the work to actually service the request.

Device Manager family generic drivers can either be *concurrent* or *nonconcurrent*; Device Manager clients can make either *synchronous* or *asynchronous* requests.

The Device Manager FPI server knows that *nonconcurrent* drivers cannot handle multiple requests concurrently. It provides a mechanism to queue client requests. No subsequent requests are made of the plug-ins' task until it signals completion of the previous request by calling the `IOCommandIsComplete` function provided by the Device Manager Family Library context.

For *concurrent drivers*, all queuing and state information describing an I/O request is contained within the plug-in code and data and within any queued requests. The FPI library forwards all requests to the FPI server regardless of the status of the outstanding I/O requests.

The FPI library makes sure that both synchronous and asynchronous clients see appropriate behavior. When a client calls a family function asynchronously, the FPI library sends an asynchronous kernel message to the FPI server and returns to the caller.

When the client makes a synchronous request, the FPI library sends a synchronous kernel message which blocks the requesting client. The plug-in task continues to run within its' own task context permitting other clients to make requests of this plug-in concurrent with the processing of other synchronous requests. When the FPI server replies to the message, the client is unblocked and able to continue.

The FPI server queues the incoming request to the target plugin queue. The per-plugin task code manages concurrent and nonconcurrent drivers appropriately. It sends all requests to the driver; if the diver is nonconcurrent, the plugin code then waits on an event which will be set when the driver returns with the I/O completion message. When the plug-in signals that the I/O operation is complete, the FPI server replies to the original message. When the Device Manager FPI receives the reply, it either returns to the synchronous client thus unblocking it, or calls the asynchronous client's I/O completion routine. When it finishes processing a message, the per-plugin task then loops back to process the kernel queue that is fed by the Device Manager Server and `IOCommandIsComplete`.

7.2. RELIABILITY, AVAILABILITY AND SERVICEABILITY (RAS)

The Device Manager provides recovery and persistence by making use of the kernel notify for task termination, a special "henchtask" to process terminations and exception handlers. The exception handlers, at a minimum, will prevent system failure. However, when possible it will reflect data access and other client related errors back to the client as a failure indication on the appropriate request. After cleaning up, the exception handler will then longjmp to the beginning of the tasks' processing loop. Failure during the exception handler processing or recursive failure will cause the task to be terminated. The henchtask will then attempt a task restart.

By monitoring task termination, tasks that terminate unexpectedly (i.e. the Family Server task or plugin tasks) will be restarted automatically (i.e. will be reloaded and reinitialized) thus providing additional availability and will be impervious to corruption of task context data. Faults will be isolated to individual plugins or to the Device Manager but will not affect the entire system. Methods will to be put in place to prevent recursive failure and in some cases a complete failure of the Device Manager family might occur, but the entire machine will not be affected (although, if Device Manager work is what the user needs done, that's not much consolation). *[More work is needed to determine what information is required by the "henchtask"in order to perform proper cleanup and restart.]*

Serviceability requires that individual failures will be logged so that failures can be diagnosed off-line. This information will include as much environmental data as is relevant to the failure; software error records will be logged for all unexpected conditions and failures. *[A discussion of the use of the Kernel logging facility needs to be integrated into this section.]*

7.3. THE DEVICE MANAGER FAMILY API

The Device Manager Family API is much like the family-to-plugin API. The Device Manager FPI Library exports the `DoDeviceManagerIO` entry point which will be used by the System-7 API Library code and Copland native applications to receive the parameters that will be passed to the accept routine. The single entry point is modeled after the `DoDriverIO` call.

```
OSStatus
DoDeviceManagerIO (IOCommandID       commandID, // command ID
                   IOCommandContents contents,  // cmd specific pb
                   IOCommandCode     code,      // open/close/etc.
                   IOCommandKind     kind );    // synch/asynch/immed
```

7.4. DEVICE MANAGER FLOWS

The following flows describe the process that takes place for the three different kinds of request - immediate, synchronous and asynchronous. Immediate requests are handled synchronously and immediately from the requestor through to the driver. Synchronous and asynchronous requests are only differentaited at the Send interface and control the state of the requesting task while a request is outstanding. he server and plugin tasks treat synchronous and asynchronous requests the same (except for running the ioCompletion routines). The differentiation for concurrent and nonconcurrent drivers is not shown in these flows. It is explained in the text along with the synchronous and asynchronous flows however. The flows are shown for Blue task clients, but are the same for native Copland-savvy tasks except for running the ioCompletion routine for asynchronous requests. This is discussed below also. (The dark numbered circles are the sequence of events in the flow diagrams.)

Immediate requests cause a synchronous Send to be done by the FPI Library. The accept function running in the context of the client task sees this request and calls the plugin directly (via the DoDriverIO entry point). It uses the information from the parameter block contained in the DMServerMessage along with other information it has about the plugin to construct the parameters for the call. The plugin (driver) completes the request and returns to the accept function which in turn returns to the message system which completes the Send. This unblocks the FPI Library code which then returns to the application.

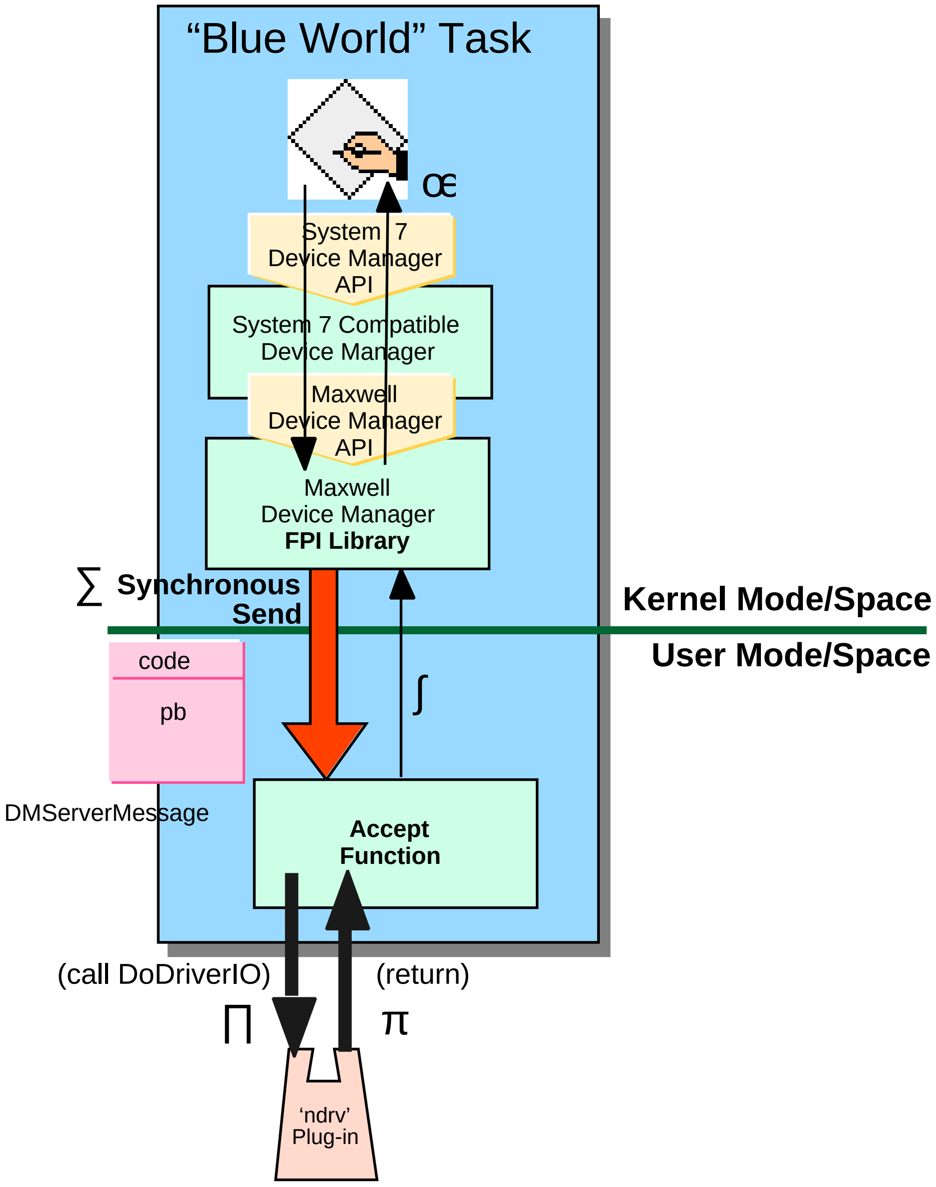

Figure 10.2 Immediate Device Manager Request Flow

A synchronous request causes the FPI Library to issue a synchronous Send to the Device Manager Server port. The accept function sees this request and builds a DMServerQElement which it enqueues on kernel queue being waited on by the plugin task for this plugin. The plugin task wakes on the queue and calls the plugin with the request using its DoDriverIO entry point. If this is a nonconcurrent driver, after the driver returns, the plugin task code waits on an event flag in order to serialize requests to the driver. When the driver completes the request, it calls the

IOCommandIsComplete entry point in the Plugin task. This function enqueues the response DMServerQElement to the plugin tasks kernel queue. If this was a nonconcurrent driver, the event flag that the plugin task is waiting on is also set allowing the plugin task to return to the WaitOnQueue at the beginning of its processing loop and receive the next work request. When it processes the response from the plugin, it does a ReplyToMessage allowing the FPI Library code to wake up and return to the application requestor with the results.

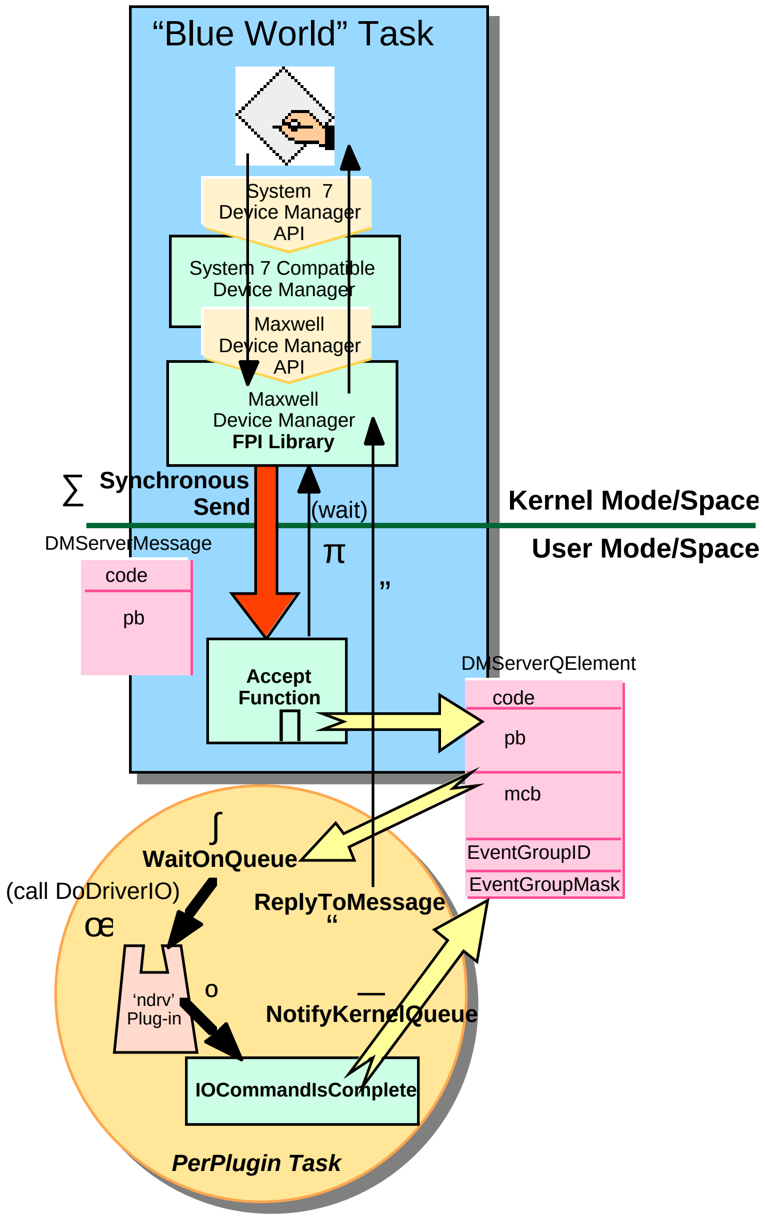

Figure 10.3 Synchronous Device Manager Request Flow

An asynchronous request causes the FPI Library to issue an asynchronous send which causes the accept function to run and enqueue a DMServerQElement for the

PerPluginTask. It then returns to the message system which returns to the application and allows it to continue (the dark numbered circles represent this sequence). The plugin task awakens on the kernel queue and processes the request by sending it to the plugin for processing. The process is the same as for a synchronous request from this point until after the ReplyToMessage is done at ∞. For asynchronous requests that specify an ioCompletion routine, for Blue clients a cooperative interrupt is run to allow the ioCompletion routine to be thunked in the data context of the requesting Blue task. For native tasks, a software interrupt is used to run the ioCompletion routine.

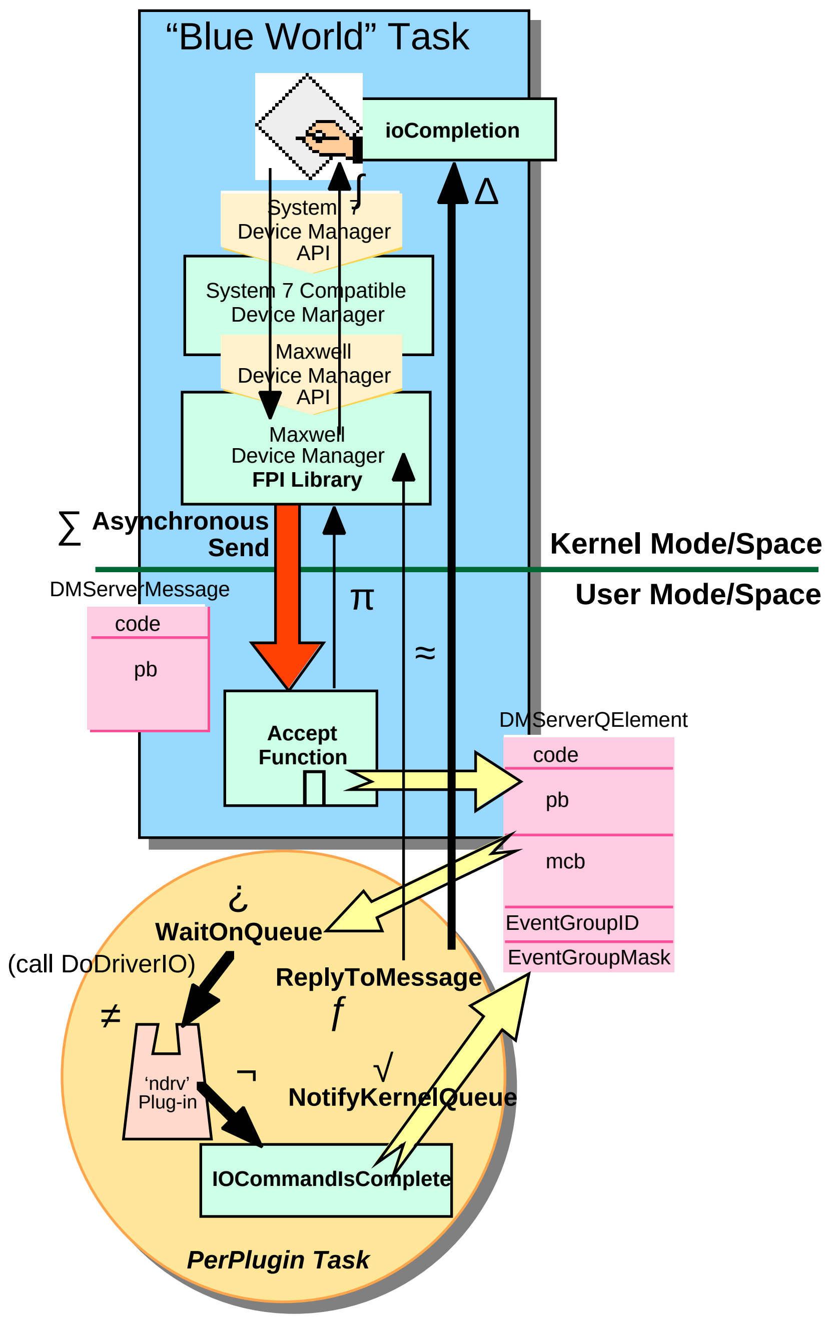

Figure 10.4 Asynchronous Device Manager Request Flow

7.5. PSEUDO-CODE IMPLEMENTATION

The following is c-like pseudo code outlining the Device Manager Server implementation. This code does not show any of the private FPI calls. It demonstrates the main logic. It is not meant to describe all the details of the implementation, but rather to describe the flavor of the implementation structure and flow. The psuedo-code here also shows a static fixed-size plugin table. This will be dynamic in the actual implementation.

.