Native Drivers for PCI and other Peripherals in Marconi

April 13, 1994

Please direct all comments to: Wayne Meretsky
AppleLink: WNM
Extension: 4-2955

Scope

This document includes the design and implementation plan for allowing entirely native device drivers to be used with Marconi.

Summary

It is difficult to explain to third party developers of peripheral equipment exactly why the only implementation option available for writing a device driver precludes native code. Combined with the move away from NuBus and towards PCI, the developer story is poor.

To entice card developers to create Macintosh devices and to simplify the transition to a Modern Operating System, a native driver model has been designed that will allow native drivers to seamlessly integrate with System Software.

Finally, Apple must guarantee that these native drivers will continue to function well with future releases of system software including Maxwell.

Approach

Marconi will include a new Device Manager that supports a new format for device drivers. Attempts to open a device driver will cause the Device Manager to look for this new driver format in addition to the presently defined 'DRVR' described by Inside Macintosh Volume II, Chapter 6.

Device Manager Support

The process of locating device drivers during startup does not presently utilize the Device Manager's OPEN service because that service does not provide sufficient control over the exact loading and initialization process that is required by system startup. In fact, it is widely held that the Device Manager's OPEN service is never called to open any driver because the behavior defined by OPEN is always inappropriate.

Due to the added complexity of opening native drivers, the entire process of locating a driver, loading it into memory, and initializing the driver's related data structures will be packaged into a new externally defined and supported API extension to the Device Manager. All system software that presently loads and installs drivers into the system must be changed to utilize this new service if they are to be capable of supporting native device drivers.

In addition to changes to support native drivers, the Device Manager will provide a new flavor of `IODone` to greatly simplify the life of many driver writers.

This spin of request completion, `IORequestDone`, allows the driver to specify which request has completed. The parameter block is presumed to be located

somewhere on the device control block's list of outstanding requests rooted by the dCtlQHdr but, unlike IODone, it need not be the first request on that queue.

The new API is as follows:

```
void IORequestDone (DCtlPtr             theDce,
                    ParmBlkPtr  thePb,
                    OSErr               theResult);
```

Native Device Driver Description

Device drivers are best characterized by three general properties:

- Container format
- API exported from driver to system
- API imported by driver from system

The device drivers defined for use in the present system use the 'DRVR' container format. They export five callable routines: Open, Close, Prime, Control, and Status. The imports of drivers in the present system are not characterized in any fashion by Apple although most device driver authors have discovered what works and what does not.

Native Container Format

The container format for native device drivers will be that supported by the Code Fragment Manager, CFM. This format provides all mechanisms necessary for drivers, is integrated with the system, and is well understood by developers and tools vendors.

Native Driver Exports

Native device drivers will export a single entry point that is used to handle all Device Manager operations. The API for that entry point is as follows:

```
enum
{
        SynchronousIOKind    = 0x00000001,      // Queued, blocking
        AsynchronousIOKind = 0x00000002, // Queued, non-blocking
        ImmediateIOKind      = 0x00000004,      // Non-queued

        OpenCmd              = 0,
        CloseCmd             = 1,
        ReadCmd              = 2,
        WriteCmd             = 3,
        ControlCmd           = 4,
        StatusCmd            = 5
};

typedef unsigned long      RequestKind;
```

```
OSErr  DoDriverIO       (DCtlPtr  theDce,
                         ParmBlkPtr   thePb,
                         IOCommand    theCmd,
                         RequestKind  ioKind);
```

The device driver can determine the nature of the I/O request from the command (Open, Close, Read, Write, Control, or Status) and request kind (Synchronous, Asynchronous, or Immediate). Typically, drivers ignore the kind of request as most do not accept immediate requests and the synchronous and asynchronous aspects are handled by the Device Manager.

The semantics of this routine are the collective semantics of the routines described by section of Inside Macintosh Volume III, Chapter 6 entitled *Writing Your Own Device Drivers*. However, the following exceptions to that section must be followed:

- Ignore the malarkey about parameter passing conventions based upon M68000 registers. The routines receive their parameters as described above subject to the calling conventions specified by the PowerPC Runtime Architecture. In short, if your DoDriverIo routine is written in 'C' the correct behavior is guaranteed.
- Ignore the plethora of detail regarding when to call IODone and when to "return via an RTS instruction". Instead use the ioKind parameter to determine if the request is ImmediateIOKind. If so, then your processing of the request must be completed and the result of the request must be reflected in the return value from your driver. If the request is other than ImmediateIOKind, then IODone or IORequestDone *must* be called by your driver at some point in time, either before returning to the Device Manager or later.
- If a call is ImmediateIOKind then the return value from your driver is the resulting status of the request and must be valid. Immediate requests are finished when your driver returns to the Device Manager. <<<>>>
- If a call is either SynchronousIOKind or AsynchronousIOKind then the return value from your driver is ignored and the status of the request must be specified via a call to either IODone or IORequestDone.
- All Open and Close calls are immediate.
- All Control calls with the csCode of killCode (1) generated by the Device Manager are immediate.
- Don't look at the ioTrap field of the parameter block to determine the kind of request or kind of command. Rather, use theCmd and ioKind parameters.
- Your driver must be reentrant to the extent that at any call from your driver to ioDone, your driver may be reentered with another request.
- CFM allows your code fragment to have an initialization routine that is executed at the time that your driver is loaded. This routine, if present, will run prior to your driver being opened. It is possible that your driver will be loaded and its initialization routine run even though it is never opened and, therefore, never closed. It is important that any processing done in any initialization routine not allocate memory or other system resources because there is no opportunity to release these resources. Rather, your driver should allocate such resources at the time it is opened.

- Never "JMP to the IODone routine" as described throughout the Device Manager documentation. Rather, call IODone to notify the Device Manager that a given request has been completed.
- A native device driver does not have any sort of header akin to that described by figure 2 of Inside Macintosh Volume III, chapter 6 entitled *Driver Structure*. Therefore, the Device Manager does not have any information upon which to base the dCtlEntry fields dCtlFlags or dCtlDelay. It is the responsibility of a native driver to fill in these fields during the processing of its' Open request. The bits of interest are dNeedGoodbye (bit 12), dNeedTime (bit 13), and dConcurrent (bit 2). Bits dRAMBased (bit 6), dReadEnable (bit 8), dWritEnable (bit 9), dCtlEnable (bit 10), dStatEnable (bit 11), and dNeedLock (bit 14) are ignored and *must* always remain zero. Bits dOpened (bit 5), drvrActive (bit 7) are managed solely by the Device Manager and should never be modified by the driver. Bits 0, 1, 3, 4, and 15 are undefined, reserved for future use, and *must* remain zero to insure future compatibility.
- A native device driver cannot make use of the dCtlEMask and dCtlMenu fields of its' driver control block. Native drivers cannot be used for creating desk accessories.

Many clients of the present Device Manager go to great lengths to allow their driver to handle multiple requests concurrently. The Device Manager allows native drivers to accomplish this more simply. Drivers that want to handle multiple requests simultaneously should set the dConcurrent bit (bit 2) of the dCtlFlags word in the device control block. In concurrent mode, the management of requests by the device manager is altered as follows:

- All Read, Write, Control, and Status requests received by the Device Manager are immediately forwarded to the appropriate driver.
- The Parameter Blocks corresponding to the requests are placed onto the device's request queue rooted by the dCtlQHdr field of the device control block.
- The drvrActive bit (bit 7) in the dCtlFlags field of the device control block is never set.
- The driver must remove the request from the device's request queue rooted by the dCtlQHdr field of the device control block prior to completing the request.
- The driver must use the IORequestDone service to complete the request. It must *not* use the IODone service to complete any requests.
- The driver is responsible for ensuring that all requests have been completed prior to returning from a Close request. Once a Close request has been made to a concurrent driver, no further requests will be made to the driver until the driver has completed the Close request and the driver is again opened.

Native Driver Imports

The nature of CFM requires that fragment imports be identified in some manner. This is done by linking the

fragment against a library that indicates particular symbols will be bound at execution time. Apple will provide a library of symbols for use when linking drivers. This library should be used

in lieu of any other Apple supplied libraries when a developer is linking a device driver.

The intent behind explicitly specifying the system entry points available to a driver is to guarantee compatibility of drivers thus linked with future releases of the system.

Limitations

Nothing about the ability of the system to utilize native drivers should be construed to mean that there is a native I/O subsystem in Marconi or that mid-M68000 instruction interrupt handling is possible. Additionally, the performance of a native device driver may be greater or lesser than the M68000 equivalent. At this time, no commitment has been made to a native or fat device manager for Marconi.

Additionally, this document only discusses the ability to have native software for Device Manager Drivers. Other driver-like-things, such as ADB drivers, which are not managed by the Device Manager realize no benefit from any of the system enhancements described herein.

Related Work

As discussed above, numerous changes will need to be made to the startup sequence and various I/O subsystems to fully support native drivers. An incomplete list of these changes follows:

Serial Drivers

The Serial Drivers are massaged into the Device Manager's unit table during startup and are neither 'DRVR' nor CFM in format. If a native Serial Driver is desired for Marconi that startup code and the packaging of the Serial Driver will have to be changed.

SCSI Drivers

The SCSI Manager uses its own mechanisms for loading device drivers from SCSI devices. The loaded data from the device is not actually a driver. Rather, it is simply code that is executed when the system is initializing. If the SCSI device is to be utilized by HFS, for example a Hard Drive or CD-ROM, then the code loaded by the SCSI manager shoe horns itself into the Device Manager's unit table. Therefore, significant changes to both the SCSI Manager and the code located on the disk must be made if such disk drivers are to be native.