Copland I/O Architecture

Preliminary

Developer Press

CHAPTER 1

About the Copland I/O Architecture

Contents

About the Copland I/O Architecture

This chapter provides an overview of the Copland I/O architecture. The Copland I/O architecture is designed to improve the user experience by providing superior performance, better responsiveness, and increasingly robust systems, and by supporting the advancements inherent in a microkernel-based operating system. It improves the developer experience by increasing the predictability of I/O responsiveness, by simplifying driver development, and by providing an updated 68K driver interface and an improved concurrent Device Manager.

You need to understand the framework that the I/O architecture provides for innovation and how it affects compatibility with both hardware and software products if you are one of the following types of developers:

- If you are a Mac OS licensee, you need to understand the I/O architecture to be certain that devices you incorporate into your hardware product will operate with Copland and to understand how software can be loaded into your product when it is turned on.
- If you are a hardware vendor who makes NuBus™ or PCI cards, ADB devices, GeoPort™ pods, or other hardware devices, you need to know how to create software that allows access to your product.
- If you are a system-extension author who produces software products such as network protocol implementations, file system implementations, and virtual device drivers to extend the capabilities of the system, or if you develop system utilities such as driver installers, hard disk formatting and partitioning packages, and emergency repair products, you need to understand the I/O architecture to determine if you need to modify your software product to run on Copland.
- If you are an application developer whose application writes to or otherwise manipulates devices, you need to understand how to take advantage of the new features in the Copland I/O architecture and how to enhance your application's compatibility with future versions of Mac OS.

This chapter briefly introduces the Copland I/O architecture. Then it discusses

- short- and long-term design goals of the I/O architecture
- architectural features, such as the Driver Loader Library, the Driver Services Library, booting services, power management, the user activity monitor, and support for hot swappable devices
- selected aspects of I/O families and plug-ins

- family activation models
- the Name Registry as it is used by the I/O system
- compatibility issues for device driver writers and application developers

You'll find this chapter easier to understand if you are familiar with certain features of Copland, such as its tasking mechanisms, the defined execution environments and execution modes, distinct address spaces, and microkernel messaging. You can find information about these topics in previous chapters in this document and in *Microkernel White Paper*.

Introduction

Copland changes how the lowest levels of the Mac OS work. It implements a tasking model of process management, with address space protection for tasks executing in supervisor mode. Drivers execute in supervisor mode. The transition to a microkernel-based, preemptive, multitasking operating system has significant implications for developers creating drivers and other I/O services for the Mac OS and for applications that use them:

- Applications running in user mode and driver software running in supervisor mode have no direct access to each other's data. Drivers are protected from applications and vice versa. Access to driver services is available only through an I/O family's programming interface.
- I/O devices are not directly accessible to application software, nor is it vulnerable to application error. Applications have access to hardware services only through an I/O family's programming interface.
- The context within which a driver runs and the method by which it interacts with the system are defined by the I/O family to which it belongs.

You can find more information on these topics in the section "Compatibility—Backward and Forward," beginning on page 1-34.

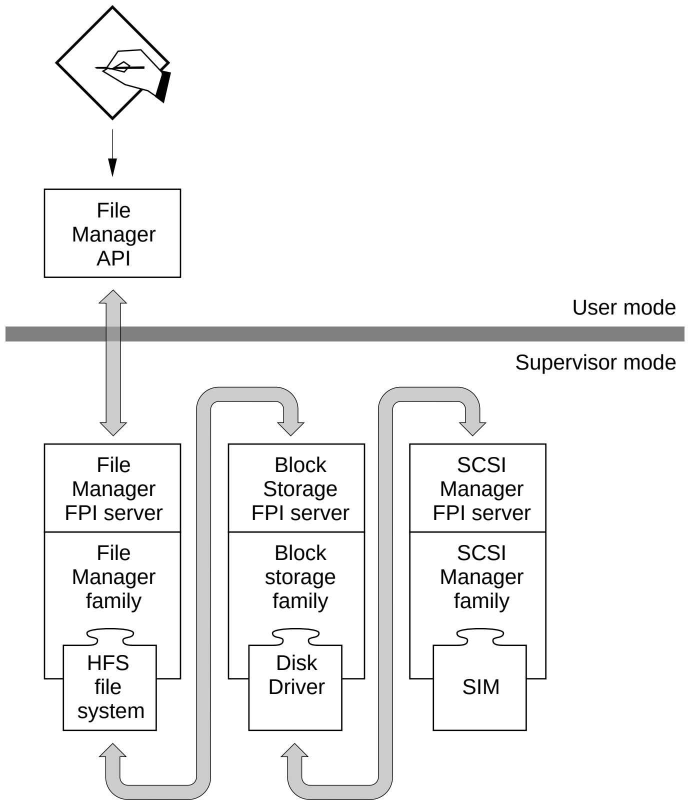
The Copland I/O architecture introduces new terminology. An I/O **family** is a collection of software pieces that provide a single set of services to the system, such as the SCSI family and its SCSI interface modules (SIMs) or the file systems family and its installable file systems. Each family defines a family programming interface (FPI) designed to meet the particular needs of that family. An FPI provides access to a given family's plug-ins.

A **plug-in** is a dynamically loaded piece of software that provides an instance of the service provided by a family. For example, within the file systems family (File Manager), a plug-in implements file-system-specific services. Plug-ins are a superset of device drivers—all drivers are plug-ins, but not all plug-ins are drivers.

Figure 1-1 illustrates an example of the relationship between an application, several I/O families, and their plug-ins. An application requests services through an FPI, shown in the figure as the File Manager API. Typically, the service requests flow as microkernel messages to FPI servers, shown in the figure as gray arrows.

In this architecture, code that executes in supervisor mode, such as plug-ins, family implementations, and the FPI servers, is *trusted*. A failure in one of these software subsystems can cause complete system failure. However, failure of any particular application does not affect the ability of the I/O system and other microkernel-level services to continue serving other clients. The I/O system is insulated from application error.

Figure 1-1 High-level view of an application, I/O families, and plug-ins



Note that Figure 1-1 shows three I/O families that work together to complete a service request. The application makes the service request which then moves through the file system family, the block storage family, and the SCSI family. However, this does not imply any hierarchical relationship among families. In fact, all families are peers of each other.

In introducing the concepts of family and plug-in, the Copland I/O architecture formalizes existing programming practices. For example, when an application accesses the services of a video device through the Display Manager, it is calling the display family. The Display Manager API is tailored to the needs of video devices. Likewise, when an application calls the Sound

Manager, it is calling the sound family. The family concept in the Copland I/O architecture explicitly acknowledges that devices of similar sorts share many characteristics and needs. Therefore, it provides family programming interfaces tailored to the needs of specific device families. These specially tuned sets of services allow drivers for a given family to be as simple as possible.

Families and plug-ins are described in more detail in the next two sections.

Families

The notion of family is fundamental to the Copland I/O architecture. A family provides a distinct set of services to the system. For example, the Open Transport family and its Data Link Provider Interface (DLPI) device drivers provide network services; the block storage family and its block storage drivers provide access to a variety of block storage mediums. Often, a family is associated with a set of devices that have similar characteristics, such as display devices or ADB devices.

Apple will provide the following families in its first release of Copland:

Device Manager family	Open Transport family
ADB family	Keyboard family
Pointing family	Display family
SCSI family	Sound family
PRAM family	IDE family
Real time clock family	PCI family
File systems family	PCMCIA family
Block storage family	NuBus family

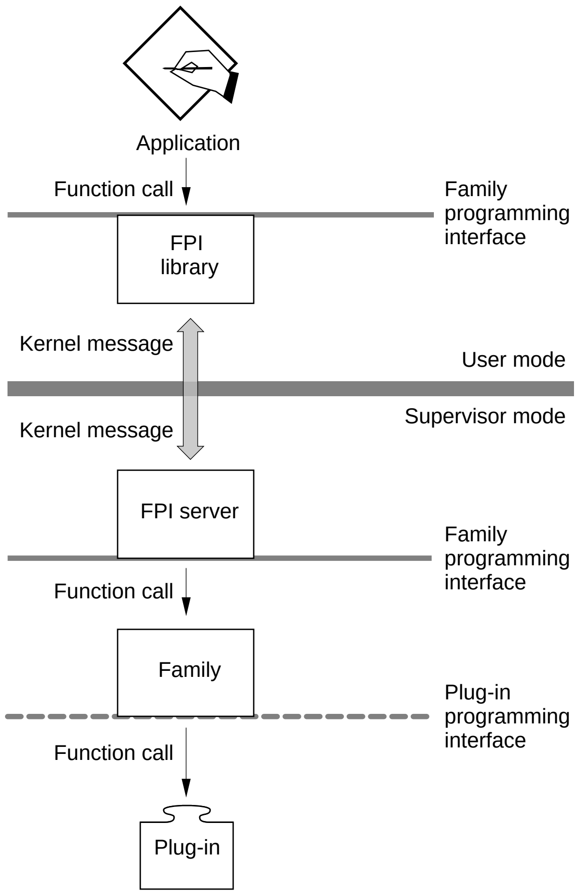
You can create additional I/O families, extending the base system features and APIs. Each family provides the following software pieces:

- a family programming interface and its associated FPI library or libraries for its clients
- an FPI server
- an activation model
- a family expert

- a plug-in programming interface for its plug-ins
- a family services library for its plug-ins

Figure 1-2 provides a high-level view of how selected family software pieces are related.

Figure 1-2 Family software diagram



The **family programming interface** (**FPI**) provides access to the family's services to applications, to plug-ins from other families, and to system software. The term *family programming interface* distinguishes an I/O family's API from other APIs provided by Copland, such as microkernel APIs or high-level Toolbox APIs. Each FPI is designed to provide callers with services appropriate to a particular family.

The FPI library contains the code that passes requests for service to the family FPI server. Typically, an FPI library maps FPI function calls into microkernel messages and sends them to the family's FPI server for servicing. To make certain optimizations possible, a family may provide two versions of its FPI library, one for user-mode clients and one for supervisor-mode clients.

An **FPI server** runs in supervisor mode and responds to service requests from family clients. How it responds to a request depends on the family's activation model. For instance, it may put a request in a queue or it may call a plug-in directly to service the request. If the FPI library and the FPI server use microkernel messaging to communicate, the FPI server supports a message port. The choice of microkernel messages as a communication mechanism is not visible to family clients. Clients use only the FPI to make requests of the family and its plug-ins. This is a change from the existing Mac OS in which both high-level and low-level interfaces to components of the operating system are available.

An **activation model** provides the runtime environment of the family and its plug-ins. For information about activation models, see the section "Activation Models," beginning on page 1-24.

A **family expert** (also referred to as a *high-level expert*) is the code within a family that maintains knowledge of the set of family plug-ins within the system. At system startup, and each time it's notified of a change in the Name Registry, the family expert scans the system's Name Registry for plug-ins that belong to its family. For example, a display family expert looks for display device entries. When a family expert finds an entry for a family plug-in, it instantiates the plug-in, making it available to clients of the family. The system notifies the family expert on an ongoing basis about new and deleted nodes in the Name Registry. As a result, the set of plug-ins known to and available through the family remains current with changes in system configuration.

Family experts do not add or alter information in the Name Registry, nor do they scan hardware. Families don't care about how devices are connected to the system—they are insulated from knowledge of physical connectivity. To learn how device information gets into the Name Registry, see the section "Name Registry," beginning on page 1-33.

The **plug-in programming interface** (**PPI**) provides a family-to-plug-in interface that defines the entry points a plug-in must support so that it can be called and a plug-in-to-family interface that defines the routines plug-ins must call when certain events, such as an I/O completion, occur. In addition, a PPI defines the path through which the family and its plug-ins exchange data.

A **family services library** is a collection of routines that provide services to the family's plug-ins. The services are specific to a given family and may be layered on top of services provided by the microkernel. Within a family, the family services library implements the methods by which data is communicated, memory is allocated, interrupts are registered and serviced, and timing services are provided. Family services libraries also maintain state information needed by a family to dispatch and manage requests.

For example, the services library for the display family provides routines that deal with vertical blanking because display devices care need them. Likewise, because SCSI device drivers must manipulate command blocks, the SCSI family services library contains routines to do that easily. A family services library that provides commonly needed routines simplifies the development of that family's plug-ins.

Plug-ins

A plug-in is a dynamically loaded piece of software that provides an instance of the service provided by a family. For example, within the file systems family, a plug-in implements file-system-specific services. The plug-ins understand how data is formatted in a particular volume format such as HFS or DOS FAT. But file systems family plug-ins don't understand how to get data from a physical device. To do that, a file system family plug-in talks to the block storage family. Block storage plug-ins provide both media-specific drivers—such as a tape driver, a CD-ROM driver, or a hard disk driver—and volume plug-ins that represent partitions on a given physical disk.

With the first release of Copland, Apple will provide plug-ins for the families listed on page 1-7. Third-party hardware developers are encouraged to develop new plug-ins.

All plug-ins share the following characteristics:

- They must conform to their family activation model.
- They cannot call Toolbox routines.
- They run in supervisor mode and have access to the microkernel's protected memory space.
- They are packaged as Code Fragment Manager fragments.
- They can be written in a high-level language.
- They must be written in native PowerPC code.
- They have a layered structure. Most of their work is done in a task. Some small amount of work may be done by interrupt handlers. The layered structure model for plug-in development allows code to be compartmentalized so that it works well within the Copland environment.

The typical parts of a plug-in include

- the main code section that runs as a supervisor-mode task. It is here that the plug-in does most of its work.
- a hardware interrupt handler that services hardware interrupts if the plug-in responds to a physical device. Only essential work that cannot be done in the task should be done by the hardware interrupt handler.

All plug-ins must have a main code section, but not all will have a hardware interrupt handler.

Plug-in code executes in supervisor mode and responds to client service requests made through the FPI. For example, Device Manager family plug-ins (device drivers of family type `'ndrv'`) respond to the functions `Open`, `Close`, `Control`, `Prime`, and so on.

Plug-in code should make no assumptions about particular hardware settings or configurations. The main code section should never attempt to obtain device configuration information directly from APIs such as the Resource Manager or the File Manager. A plug-in obtains configuration information in several ways. It can read the static configuration information stored in the Name Registry. Dynamically changing configuration information is communicated to a plug-in through the plug-in programming interface; when a family client uses the family's programming interface to notify the family of a configuration change, the family notifies the plug-in. In addition, a plug-in can call another family to obtain some types of configuration information. For instance, a video plug-in may call the PRAM family programming interface to obtain video mode information stored in PRAM prior to the last system reboot.

The hardware interrupt handler executes in supervisor mode and responds to interrupts from a physical device. It should perform only essential functions, deferring all other work to the plug-in task or a secondary interrupt handler. The plug-in programming interface specifies how interrupts are managed within a family.

Design Goals for the Copland I/O Architecture

The next two sections describe the short-term and long-term design goals of the Copland I/O architecture.

Short-Term Design Goals

In the first release of Copland, the I/O architecture is targeted to meet the following design goals:

- **End-user flexibility.** Mac OS provides end users with tremendous value that is directly attributed to the flexibility and adaptability of its I/O system. For example, its plug-and-play capability and dynamic monitor configuration are features that are simply not possible with many I/O architectures. The Copland I/O architecture is designed to provide these end-user features and to retain the flexibility of the Mac OS.
- **Performance.** The architecture favors lower-latency responses over higher bandwidths to provide greater responsiveness to users. To help achieve this goal, all drivers and all their support services are native. Additionally, very little code is permitted to run at the hardware-interrupt level. Although the architecture does not guarantee the best performance for burst and single-stream high-bandwidth clients, the Copland implementation will produce much better throughput results than that available in System 7. The I/O architecture provides support for the real-time needs of MIDI, Sound, GeoPort, and QuickTime and enables implementations that meet or exceed the performance of competing platforms.
- **PCI driver compatibility.** The Copland I/O architecture extends the architecture for the I/O system on PCI-based Mac-compatible computers.

Drivers compliant with the specification for driver development contained in the document *Designing PCI Cards and Drivers for Power Macintosh Computers* will continue to function well within the Copland I/O model. In addition, Copland seeks to provide binary compatibility with PCI ROM-based video and network drivers developed in accordance with the specification for native drivers described in *Designing PCI Cards and Drivers for Power Macintosh Computers*.

- **Reliability, availability, and serviceability.** In Copland, the I/O system works as expected and continues to work acceptably in the face of failures of particular subsystems. For instance, disk I/O continues to work if a failure in the serial hardware occurs. When failures do occur, the I/O system provides support for analysis and corrective measures by the user and by support organizations.
- **Resource allocation and control.** Having limited resources, the components of the Copland I/O system distribute those resources in a fair and meaningful fashion among themselves. In particular, the first driver loaded cannot consume resources such as memory, message ports, timers, interrupt latency, or bus bandwidth in a way that prevents subsequent drivers from loading or operating correctly. Configurations that cannot work because their needs are mutually exclusive are recognized and reported in a meaningful way.
- **Power management.** Obviously required for battery-powered systems such as PowerBook™ computers, the need for integrated power management is increasing for all systems. The I/O architecture provides an infrastructure to enable optimal power management in diverse systems.
- **Extensibility.** The Copland I/O architecture enhances the ability of OEMs to create Mac-compatible hardware and peripherals. It is intended that all hardware-dependent software fall into one of two categories:
  - software based on clearly defined hardware invariants such as big-endian addressing and the PowerPC 601, 603, and 604 processors
  - software that is dynamically loadable at system startup time, such as drivers, the SCSI Manager, and SCSI interface modules

Long-Term Design Goals

In subsequent releases of Mac OS, the I/O architecture is targeted to meet these additional design goals:

- **Multiprocessor support.** High-quality support for a limited number of tightly coupled, cache-coherent processors is a long-term goal of the architecture. While revisions to the architecture may be desirable for multiprocessor systems, conforming I/O components should be compatible within multiprocessor versions of the architecture.
- **Real-time I/O support.** The architecture specifies basic support for real-time I/O needs, largely as a subset of the resource allocation and control mechanisms provided by the architecture. Families and plug-ins are prioritized according to their needs to better support real-time clients.
- **Improved reliability, availability, and serviceability (RAS).** RAS is the natural successor to the Mac OS plug-and-play capability. The addition of RAS to Mac OS provides users, system administrators, and technicians with a broad set of tools for maintaining a Mac OS system, resulting in lower training and support costs. RAS is one of the mechanisms by which Mac OS will maintain its lead as the easiest and most configurable system available.
- **Visual system administration.** Enabling end users, system administrators, and support staff to examine and manipulate the configuration of a specific system is a natural extension to the benefits of RAS support.
- **Scalable to future technologies.** Copland provides sufficient architectural integrity to ensure that implementations of technologies that are not quite available today are obtainable on desktop platforms. ATM and infrared networking and Firewire bus connectivity are examples of such technologies.
- **Distributed computing.** As system performance increases, it is increasingly reasonable to provide access to devices that are not attached directly to the CPU on which an application is running. For example, with high-cost, high-speed networks, video capture via a frame-grabbing card plugged into a computer in another office is possible today. As networking costs decrease, distributed services become feasible on increasing numbers of desktop systems. Distribution of I/O subsystems across a suitable network is a long-term goal of this architecture.
- **Universal booting.** A single system image that boots on all hardware configurations that support Copland is a goal of the architecture. In addition, these systems will support both minimal and third-party customized installations of Mac OS.

Architectural Features

This section describes several fundamental I/O system services provided by the Copland I/O architecture. They are baseline services present in the system. They are not specific services for different classes of devices such as serial devices or video display monitors.

Driver Loader Library

The I/O architecture provides a Driver Loader Library. The **Driver Loader Library** is a set of routines that all I/O families can use to locate and instantiate their plug-ins. The routines work with all plug-ins regardless of whether the plug-in is a driver and regardless of whether the driver touches hardware. The services provided by the Driver Loader Library fall into three categories:

- routines that provide family experts with an easy way to instantiate plug-ins. All plug-ins are packaged as Code Fragment Manager fragments, frequently referred to as shared libraries. This set of utility routines serves as a wrapper around CFM functions. They hide CFM complexities, giving family experts a simple set of functions to access the shared libraries they need and load them into memory.
- driver matching routines that help family experts locate a device driver for a given piece of hardware. This makes driver replacement easy and provides support to families that manage drivers for hot swappable devices.
- routines that work with the Device Manager family. They install, remove, and replace driver entries in the unit table.

Driver Services Library

The **Driver Services Library** provides basic driver services to families. It contains all the base-level generic services needed by families and plug-ins, such as interrupt registration, timing facilities, allocation and deallocation of memory, and secondary interrupt-handling capabilities.

The Interrupt Manager is part of the Driver Services Library. It provides routines that allow drivers to install the interrupt handlers that are invoked when a device presents an interrupt to the system.

Families can extend the base system services in family-appropriate ways by adding a family services library to augment the services available from the Driver Services Library. In some cases, a family services library will replace the Driver Services Library. For example, plug-ins belonging to the Open Transport family don't link to the Driver Services Library, because the Open Transport family services library provides all the services they need.

Booting Services

The I/O architecture provides a method for loading and launching the system software. The Copland microkernel booting architecture maintains the Mac OS user experience at system startup. The user should not be required to build a system tailored for the hardware that the system will run on. Many users may choose to install hardware support for a large class of devices that might be connected to their computers. For those users, the system finds the right support software at startup time and configures that software into a runnable system without user intervention.

Power Management

The I/O architecture provides mechanisms for power state transitions within the system, such as bringing the system up the first time, shutting it down completely, moving from low to high power, and maintaining a sleep state. It provides APIs for power management at the application, plug-in, and system levels.

There are at least three systemwide power states:

- **Full power-on mode.** The core system is available for service requests. Within this mode, some devices, applications, and services may manage their power requirements independent of the system as a whole. Low-power mode is a substate of full-power mode, in that it affects only those devices that can continue to perform with less power.
- **Sleep mode.** The contents of memory are preserved, but active processing is halted.

- **Power-off mode.** The entire system is powered down and no processing of any sort is possible.

For the purposes of power management, there are three classes of devices and services:

- CPUs that have low-power modes in which some processing can still take place.
- Devices and services with a user interface that are therefore directly tied to user actions, such as keyboards, screens, modems, applications, and networks.
- Devices without a user interface, such as hard disks that may be controlled independently from user activity.

Given the fuzzy boundaries in the device and service categories and the varying nature of each device, the I/O architecture provides mechanisms for controlling power state transitions without setting policy for devices or services. A centralized power management service provides coordinated systemwide power state changes based on input from services and drivers.

The power state and power requirements of each device that is power managed is maintained in the centralized power management service. This power management service receives input from the User Activity Monitor service and individual applications and services. It provides notification to applications, drivers, and services, manages systemwide power state transitions, and provides centralized administration of device power behavior.

User Activity Monitor

Power management requires the ability to detect when the user is doing something with the computer. In Copland, the User Activity Monitor provides the power management service with information about user activity so that it can know when to put the system into sleep mode, turn a monitor down or off, and so forth.

Copland uses an activity timer to detect idle periods. Activity is defined as mouse motion or keyboard activity. Other events, such as the arrival of data on a serial interface, can also be considered activity.

The User Activity Monitor accepts requests for notification from I/O subsystems. Subsystems can request to be notified when a specified amount of time elapses during which there is no user activity. Any of the events defined as user activity cause the timer to be reset. Subsystems may also be notified that activity has occurred. This is useful when subsystems have already received notification of inactivity and powered down their hardware. Here are some examples of why a subsystem should use the User Activity Monitor:

- The screen backlight on a PowerBook computer needs to dim after a user-controllable amount of time elapses with no activity.
- The CPU should transition into low-power mode when no compute-bound process is running and a user-controllable amount of time elapses with no activity.
- The entire computer needs to transition into sleep mode after a user-controllable amount of time elapses with no activity.

The subsystems that can register activity must do so. They must tell the User Activity Monitor that activity has occurred, causing it to reset its inactivity timer and notify requesters (if any) of the event.

Support for Hot Swappable Devices

The Copland I/O architecture provides support for hot swappable devices such as PCMCIA cards—that is, it can support dynamic changes in connectivity to devices that may appear and disappear at any time. This feature allows a user to insert and remove devices such as disk driver card or modem card without powering down and restarting the computer. The family expert code that locates and instantiates the family plug-ins remains resident for families whose plug-ins exhibit dynamic plug-and-play characteristics.

A Closer Look

This section consists of selected topics concerning I/O families and plug-ins.

Families

The next sections discuss family programming interfaces and family communication models.

Family Programming Interfaces

A family provides either a user-mode or a supervisor-mode FPI library, or both, to support the family's FPI. Figure 1-3 illustrates an abstracted view of the Copland I/O architecture. Each of the large blocks in the area below the thick horizontal line represents an instance of a family. Boxes that share an edge represent directly callable interfaces.

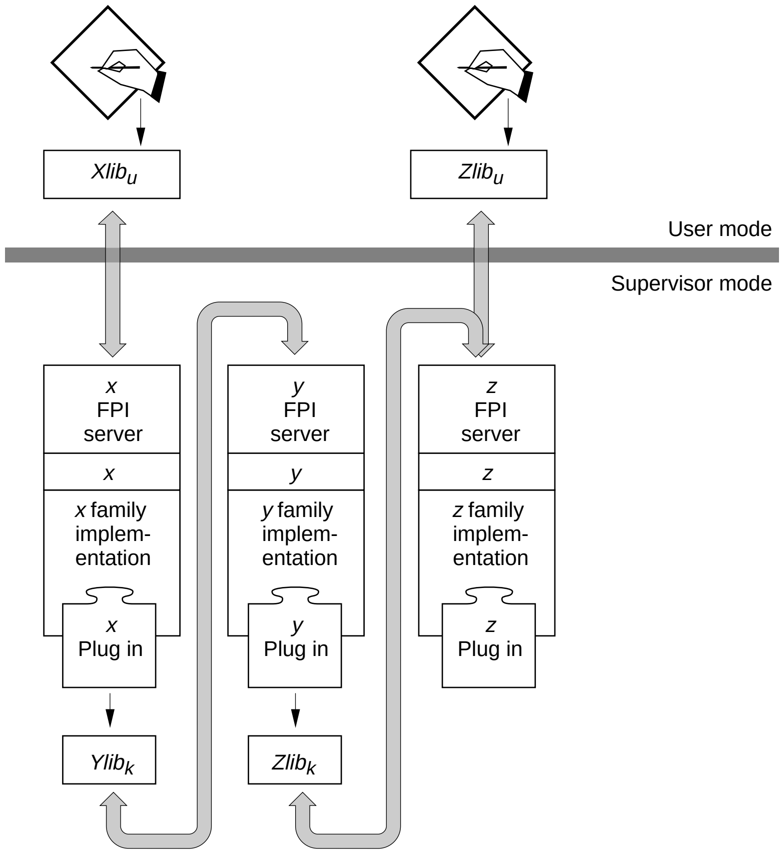
In the area above the thick horizontal line, the boxes labeled $xlib_u$ and $zlib_u$ represent the FPI libraries that support the programming interfaces for families *x* and *z*, and that are available to user-mode clients. In the area below the thick horizontal line, the boxes labeled $ylib_k$ and $zlib_k$ represent the FPI libraries for families *y* and *z* that are available to supervisor-mode clients. Typically, FPI libraries map FPI functions into microkernel messages.

Both the user-mode and the supervisor-mode versions of the FPI libraries present exactly the same interface to clients—a single FPI is the only way family services can be accessed. Copland distinguishes between the user-mode and supervisor-mode versions to permit optimization of the supervisor-mode FPI libraries in some instances. For example, operations that must be implemented in the user-mode library, such as copying data across address space boundaries, may be unnecessary in the supervisor-mode library. In some instances, the user-mode and supervisor-mode versions maybe the same.

An FPI server dispatches requests for services to the family. Typically, it does this by receiving a microkernel message, mapping the message back into the FPI function called by the client, and then calling the function. There is a one-to-one correspondence between the FPI functions called by clients and the functions called by FPI servers as a result. Take as an example the *x* family in Figure 1-3. The box labeled *x* represents the interface presented to the FPI server by the *x* family. It is exactly the same as the FPI available to applications or other system software.

The box labeled *x family implementation* represents the family activation model that defines how the request is actually serviced by family code and plug-in code.

Figure 1-3 A closer look at the Copland I/O architecture



Family Communications

Microkernel messaging is assumed to be the normal communication method for I/O families—between the FPI libraries and the FPI server for a given family, between different families, and between plug-in *x* and family *z*. That doesn't preclude the possibility of other communication mechanisms. The choice is up to the family. Whatever the communication method, it is completely opaque to a client requesting a family service.

The messaging model facilitates the development of families and plug-ins by providing a very easy programming model. It is a straightforward interfamily communication mechanism that fits well within Copland tasking mechanisms. The use of microkernel messaging permits greater independence of family activation models.

An added benefit to using microkernel messaging is that improvements in the messaging and tasking performance of the microkernel are reflected in corresponding performance improvements throughout the I/O system.

Plug-ins

Family plug-ins must operate within the activation model mandated by the family and provide the code and data exports described by family documentation. For example, *Designing PCI Cards and Drivers for Power Macintosh Computers* contains descriptions of the required interfaces and activation models for networking and video plug-ins. The required code and data exports and the activation model for each of these two families of drivers is family specific and different. The packaging for the two family driver types is the same.

The standard family and plug-in definitions cover most cases of I/O component development. However, there are exceptions to the model. The next sections describe two; there may be more.

Extending Family Programming Interfaces

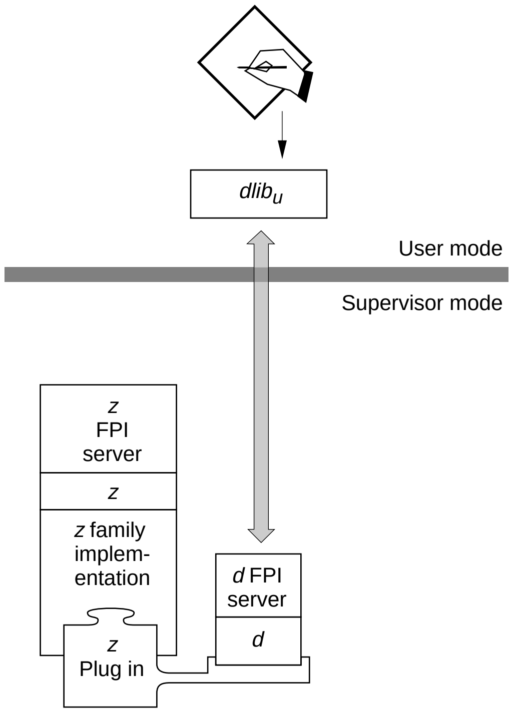
A plug-in may provide a plug-in-specific interface that extends its functionality beyond that provided by its family. This feature is useful in a number of situations. Take, for example, a block storage plug-in for a CD-ROM device. In addition to the block storage plug-in interface required of the CD-ROM device, many CD-ROM devices also present an interface that allows knowledgeable application software to control audio volume and to play, pause, stop, and so forth. Such added capabilities require a plug-in-specific API.

Most family interfaces provide some level of extensibility to the family's plug-ins. For example, the Device Manager allows extensible sets of control and status selectors that may be used to gain device-specific information and control. And Open Transport device drivers may receive special calls to extend the device information and control. This kind of device extension within the family framework is not changed with the Copland I/O architecture. If, however, a device wishes to export extended functionality outside the family framework, it needs to provide a separate message port and an interface library for that portion of the device driver, as shown in Figure 1-4.

Figure 1-4 illustrates a plug-in module labeled *z plug-in* that extends beyond the *z* family boundary. *z plug-in* is a plug-in with an extended API—it offers features in addition to those available to clients through it's family's programming interface. To make its extra services available, the plug-in must provide the additional software shown in Figure 1-4:

- *dlib*$_u$: the interface library
- *d FPI server*: the message port code
- *d*: the code that implements the extra features

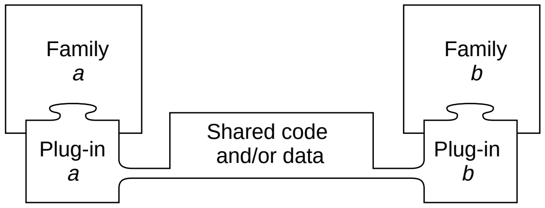
Figure 1-4 Extending a family programming interface



Sharing Code and Data Between Plug-ins

Two or more plug-ins can share data or code or both, regardless of whether the plug-ins belong to the same family or to different families. Sharing code or data is desirable when a single device driver wishes to subscribe to two or more families. Such a driver needs a plug-in for each family. These plug-ins can share libraries that contain information about the device state and common code. Figure 1-5 illustrates two plug-ins that belong to separate families and that share code and data.

Figure 1-5 Plug-ins that share code and data



Plug-ins can share code and data through Code Fragment Manager fragments, (shared libraries). The Code Fragment Manager allows you to instantiate independently plug-ins that share code or data without encountering problems related to simultaneous instantiation. The first plug-in to be opened and initialized gets access to the shared libraries, but it does not share access at that point. When the second plug-in is opened and initialized, it establishes a new connection to the shared libraries. From that point, the two plug-ins contend with each other for access to the shared libraries.

Sharing code or data is also desirable in certain special cases. Some of the special-case solutions provided on System 7 use two or more separate device drivers that use shared data as a communication mechanism. Typically, special case solutions install a set of devices and a set of special drivers. The closely coupled devices use a high-speed data path to move data between them. For example, a video input device puts video data in a shared buffer; subsequently, a video compression device reads and compresses the data it finds in the shared buffer. Access to the high-speed data path via the shared buffer is synchronized by solution specific mechanisms. In essence, this solution is a vendor-supplied family, and its plug-ins are the device drivers that come with the solution.

Activation Models

A family's activation model defines how the family software is implemented and the environment within which a family's plug-ins execute. It defines the relationship between family code and its plug-ins, including such things as

- the tasking model a family uses
- the opportunities the family plug-ins have to execute and the context of those opportunities (for instance, are the plug-ins called at task level? at secondary interrupt level? and so forth)
- the knowledge about states and processes that a family and its plug-ins are expected to have
- the portion of the service requested by the client that is performed by the family and the portion that is performed by the plug-ins
- the required characteristics of plug-ins, such as whether the plug-in blocks or returns an error when it encounters resource exhaustion

If you want to develop a new I/O family, you need to design and implement an activation model that best suits the needs of your I/O family. If you want to develop a new plug-in, you need to understand the activation model used by the family to which your plug-in belongs.

This section describes three family activation models used in the Copland I/O system. Each model provides a distinctly different environment for the plug-ins to the family, and different implementation options for the family software. The activation models discussed are

- the single-task model
- the task-per-plug-in model
- the task-per-request model

Many variations of (and hybrid approaches to) the activation models discussed here are possible and to be expected. The choice of activation model is left to the family designer. The selected models are simply examples of how you can implement a family.

To provide the asynchronous or synchronous behavior desired by the family client, the three activation models discussed here use microkernel messaging as the interface between the FPI libraries that family clients link to and the FPI servers. Within all activation models, asynchronous I/O requests are provided a task context. In all cases, the implementation of the FPI server depends on the family activation model.

The choice of activation model limits the plug-in implementation choices. For example, the activation model defines the interaction between a driver's hardware interrupt handler and the family environment in which the main driver code runs. A plug-in must conform to the activation model employed by its family.

You cannot understand the discussion of activation models without some understanding of Copland's messaging system and the tasking and interrupt mechanisms that define the environments in which software executes. You can find information about these topics in earlier chapters in this document and in *Microkernel White Paper.*

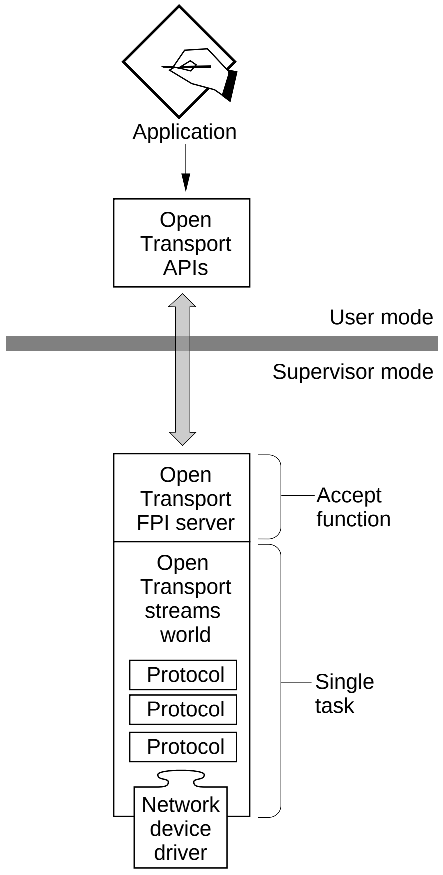
Single-Task Model

In the single-task activation model, the family runs as a single monolithic task that is fed from above by a request queue and from below by interrupts delivered by the plug-ins. Requests are delivered from the FPI library to an accept function that queues the request for processing by the family's processing task and wakes the task if it is sleeping. Queuing, synchronization, and communication mechanisms within the family follow a well-defined set of rules specified by the family.

The interface between an FPI server and a family implementation using the single-task model must be asynchronous. Regardless of whether the family client called a function synchronously or asynchronously, the FPI server always calls the family code asynchronously. The FPI server must maintain the set of microkernel message IDs that correspond to messages to which the FPI server has not yet replied.

Consider as an example the Open Transport family, which uses the single-task activation model, shown in Figure 1-6. The Open Transport FPI server is an accept function that executes on the thread of the calling client via the FPI library. An accept function, unlike message-receive-based microkernel tasks, is able to access data within the user and microkernel bands directly. The accept function messaging model requires that the Open Transport FPI server be reentrant because the calling client task may be preempted by another Open Transport client task making service requests.

Figure 1-6 Single-task activation model



When an I/O request completes within the Open Transport environment, the Open Transport stream's completion notification trickles upstream until it reaches the stream head and from there the Open Transport family's FPI server converts the completion into the appropriate microkernel message ID reply. The Open Transport family implementation is insulated from the microkernel; it has no microkernel structures, IDs, or tasking knowledge. On the other hand, the relationship between the FPI server and the Open Transport family code is rich, asynchronous, and has internal knowledge of Open Transport data structures and communication mechanisms.

The single-task model is best for families of devices that have either of two characteristics:

- Each I/O request requires little CPU effort. This characteristic applies not only to keyboard and mouse devices but also to DMA devices to the extent that the CPU need only set up the transfer.
- No more than one I/O request is ever handled at once. This characteristic might apply to sound, for example, or to any device for which exclusive access is required. It also applies to families that monitor their own scheduling for the interleaving of family I/O processing, such as Open Transport.

Here are the key questions to ask before deciding whether to choose this model:

- Can the CPU initiate an I/O request rapidly and then not be involved until the request completes?
- Do supported devices implicitly allow only one I/O request to be completed at a time or does the family provide for its own I/O scheduling?

If the answer to either question is yes, the single-task model is the right choice.

Task-per-Plug-in Model

In the task-per-plug-in activation model, for each plug-in instantiated by the family, the family creates a task that provides the context within which the plug-in operates. In Copland, the Device Manager family uses the task-per-plug-in activation model. Figure 1-7 illustrates the task-per-plug-in model using the Device Manager family as the representative family,

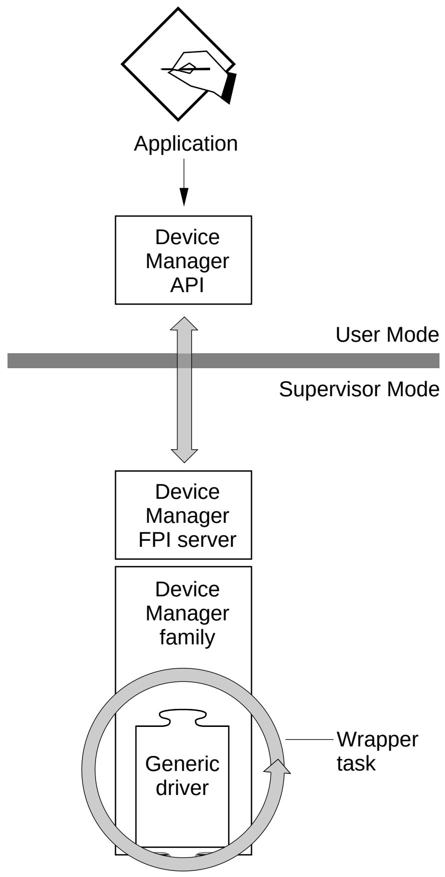
Typically with this model, the FPI server is a simple task-based message-receive loop or an accept function that presents data to an event-based task loop. The FPI server receives requests from calling clients and passes those requests to the family plug-ins. The FPI server is responsible for making the data associated with a request available to the family, which in turn makes it available to the plug-in that services the request. In some instances, buffers associated with the original request message may need to be copied or mapped once.

The family code consists in part of one or more tasks, one for each family plug-in. The tasks act as wrappers for the family plug-ins—all tasking knowledge is located in the family code.

When a plug-in's task receives a service request (by whatever mechanisms the family implementation uses), the task calls its plug-in's entry points, waits for the plug-in's response, and then responds to the service request.

The plug-in performs the work to actually service the request. It doesn't need to know about the tasking model used by the family or how to respond to event queues and other family mechanisms. It just needs to know how to perform its particular function.

Figure 1-7 Task-per-plug-in model



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 regardless of the status of outstanding I/O requests to the FPI server. When the client makes a synchronous service request, the FPI library sends a synchronous microkernel message. This message blocks the requesting client, but the plug-in's task continues to run within its own task context, permitting clients to make requests of this plug-in even while another client's synchronous request is being processed.

For the Device Manager family, generic drivers can be either concurrent or nonconcurrent; clients of the Device Manager family can make both synchronous and asynchronous requests. The Device Manager FPI server knows that nonconcurrent drivers cannot handle multiple requests concurrently. Therefore, it provides a mechanism to queue client requests. It makes no subsequent requests to a plug-in’s task until the task signals completion of an earlier I/O request.

The FPI library makes sure both synchronous and asynchronous clients see appropriate behavior. When a client calls a family function asynchronously, the FPI library sends an asynchronous microkernel message to the FPI server and returns to the caller. When a client calls a family function synchronously, the FPI library sends a synchronous microkernel message to the FPI server and does not return to the caller until the FPI server replies to the message, thus blocking the caller’s execution until the I/O request is complete.

In either case, the behavior of the Device Manager FPI server is exactly the same: for all incoming requests, it either queues the request or passes it to a family task, depending on whether the target plug-in is busy. When the plug-in signals that the I/O operation is complete, the FPI server replies to the original microkernel message. When the FPI library receives the reply, it either returns to the synchronous client, unblocking its execution, or it calls the asynchronous client’s I/O completion routine.

The task-per-plug-in model is intermediate between the single-task and task-per-request models in terms of the number of tasks it typically uses. It is best used where the processing of I/O requests varies widely among the plug-ins. In this model, the plug-in 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.

Task-per-Request Model

The task-per-request model shares the following characteristics with the two activation models already discussed:

- The FPI library to FPI server communication provides the synchronous or asynchronous calling behavior requested by family clients.

- The FPI library and FPI server use microkernel messages to communicate I/O requests between themselves.

In the task-per-request model, the FPI server's interface to the family implementation is completely synchronous.

In this model, one or more internal family request server tasks, and, optionally, an accept function, wait for messages on the family message port. An arriving message containing information describing an I/O request awakens one of the request server tasks, which calls a family function to service the request. All state information necessary to handle the request is maintained in local variables on the thread of execution of the request server task. The request server task is blocked until the I/O request completes, at which time it replies to the microkernel message from the FPI library to indicate the result of the operation. After replying, the request server task waits for more messages from the FPI library.

As a consequence of the synchronous nature of the interface between the FPI server and the family implementation, code calling through this interface must be running as a blockable task. This calling code is either the request server task provided by the family to service the I/O (for asynchronous I/O requests) or the task of the requester of the I/O (for certain optimized synchronous requests).

The task-per-request model is best for a family where an I/O request can require continuous attention from the CPU and multiple I/O requests can be in progress simultaneously. A family that supports dumb, high-bandwidth devices is a good candidate for this model. The Copland File Manager uses the task-per-request model.

One problem associated with this activation model is tuning the number of request server tasks to permit the desired level of concurrence. Tuning can be done dynamically: When the family detects that performance could benefit from more request server tasks to process more requests concurrently and there are resources to permit it, new tasks can be created as needed. Similarly, when resources become scarce or the number of concurrent requests is much smaller than the number of request server tasks available to handle them, some tasks can be destroyed, freeing their resources for other uses. This programming model requires the family plug-in code to have microkernel tasking knowledge and to use microkernel facilities to synchronize multiple threads of execution contending for family and system resources.

Family Programming Issues

The choice of activation model is the biggest family programming issue. Each of the models discussed previously has merit. Within each model, there are issues to be addressed. The single-task and task-per-plug-in models require state information to be stored either within the FPI libraries, the plug-ins, or the family activation code, or within some combination of those. The task-per-request model is the simplest model, but it will probably be the most expensive model in terms of system overhead. It makes heavy use of microkernel messaging and tasking resources.

Unless there are multiple task switches within a family, the tasking overhead is identical within all of the activation models. The shortest task path from application to I/O is completely synchronous because all code runs on the caller's task thread. For a long I/O path, through multiple families, the greater the use of synchronous calls, the smaller the number of task switches. However, using only synchronous calls decreases the responsiveness of the application making the request— its activity stops pending the completion of an outstanding I/O request. Providing at least one level of asynchronous call between an application and an I/O request results in the best latency results from the user perspective. Within the file system, the application task is not used as the thread of completion for I/O. A task switch at the File Manager API level allows a user-visible application, such as the Finder, to continue. The File Manager creates an I/O task thread to handle the I/O request, and that task might be used via synchronous calls by the block storage and SCSI families to complete their part in I/O transaction processing.

This kind of short-cut communication between families requires a very clear understanding of the relationships between the families, including the stack needs of the called family, the activation model of the called family, and the asynchronous and synchronous paradigms used by the called family. This is part of the decision-making process in developing each family activation model.

Name Registry

The Name Registry is a high-level Mac OS naming service that stores system information. It is key to implementing several important features in the Copland I/O architecture:

- **Effective driver replacement and overloading capability.** This capability allows you to release updates to drivers.
- **Dynamic driver loading and unloading.** The Name Registry provides a dynamic and flexible environment for identifying devices. This type of capability is necessary for supporting devices such as hot swappable PCMCIA cards.
- **Simplification of driver writing.** You won't need to follow different rules for writing device drivers located on the main logic board, NuBus, the PCI bus, or the PCMCIA bus.
- **Hardware-independent device drivers.** The Name Registry provides the layer of abstraction necessary for driver writers to remove conflicting device identification and device information callouts (as occurred previously with the Slot Manager) that prevented drivers from being portable to new versions of Macintosh hardware.

The Name Registry is a tree-structured collection of entries, each of which can contain an arbitrary number of name-value pairs called properties. Family experts peruse the Name Registry to locate devices or plug-ins available to the family. Low-level experts, described later in this section, describe platform hardware by populating the Name Registry with device nodes.

The Name Registry contains a subtree pertinent to the I/O architecture: the device portion of the Name Registry describes the configuration and connectivity of the hardware in the system. Each entry in the device subtree has properties that describe the hardware represented by the entry and may contain a reference to the driver in control of the device.

A **low-level expert**, sometimes referred to as a *bus expert* or *motherboard expert*, has specific knowledge of a piece of hardware such as a bus or a main logic board. It knows how physical devices are connected to the system and it installs and removes that information in the device portion of the Name Registry.

For example, a SCSI bus expert scans a SCSI bus for devices and installs an entry into the device portion of the Name Registry for each device that it finds. The SCSI bus expert knows nothing about a particular device for which it installs an entry. As part of the installation, the SCSI bus expert invokes the driver matching routines in the Driver Loader Library to associate a driver with the entry. The driver knows the capabilities of the device and specifies that the device belongs to a given family.

Low-level experts and family experts use the Name Registry notification mechanism to recognize changes in the system configuration and to take family-specific action in response to those changes.

Here's an example of how family experts, low-level experts, and the Name Registry service work together to stay aware of dynamic changes in system configuration. Suppose that a Macintosh Duo is docked. The Duo motherboard expert notices that a new bus, a new network interface, and a new video device have appeared within the system. The Duo motherboard expert adds a bus node, a network node, and a video node to the device portion of the Name Registry. The Name Registry service notifies all software that registered to receive notifications of these events.

Once notified that changes have occurred in the Name Registry, the networking and video family experts scan the Name Registry and notice the new entry belonging to their family type. Each instantiates the new entry within the family.

The SCSI bus expert notices an additional bus, and probes for SCSI devices. It adds a node to the Name Registry for each SCSI device that it finds. New SCSI devices in the Name Registry result in perusal of the Registry by the block storage family expert. The block storage expert notices the new SCSI devices and loads the appropriate drivers, and then creates the appropriate volume Registry entries to make these volumes available to the File Manager. The File Manager receives notification of changes to the block storage family portion of the Registry, and notifies the Finder that volumes are available. Those volumes then appear on the user's desktop.

Compatibility—Backward and Forward

The following sections discuss Copland compatibility issues for developers of device drivers and applications.

If You Develop Device Drivers

Copland and its I/O architecture introduce a new environment for device drivers—one that is fundamentally different from that familiar to current Macintosh driver developers. Although Copland places some restrictions on drivers, it greatly increases system stability and protects drivers from application error.

The System 7 I/O architecture is based on resources of type `'DRVR'` and on the Device Manager API. Many different types of software use these mechanisms. Some types are affected by the changes introduced by Copland I/O and some are not.

Copland employs a more restricted concept of driver software. In the Copland I/O architecture, a driver is the native code that controls a physical device or that manages a system service. (Code that controls a virtual device such as a RAM disk may also be considered a driver in Copland.) This type of software (that controls a physical device or manages a system service) is affected by the new I/O architecture in Copland. Example of this type of software include

- serial drivers (.AIn, .BOut)
- protocol stacks (.MPP, .IPP)
- network drivers (.ENET, ADEVs, MDEVs)
- video drivers (.Display)
- SCSI interface modules (SIMs)

Software that uses the 'DRVR' resource type and the Device Manager API to provide application-level functionality is not directly affected by Copland I/O changes. Examples of this type of software include:

- desk accessories
- print drivers

For backward compatibility, Copland supports, through the Device Manager, emulated drivers of type `'DRVR'` that do not touch hardware. Such software is not a plug-in. It runs in user mode outside the I/O system and can exist only in the traditional application environment that uses the `WaitNextEvent` function and that has full access to the Toolbox.

The Copland I/O system is the first complete implementation of the I/O architecture described in this chapter. A subset of the I/O architecture is implemented to support PCI devices on upcoming Power Macintosh™ models. The document *Designing PCI Cards and Drivers for Power Macintosh Computers* describes the capabilities provided to driver writers for the first PCI-based Power Macintosh computers. If you write a PCI driver according to the specifications there, PCI cards with ROM-based drivers will work unchanged between the version of Mac OS delivered on upcoming PCI-based Power Macintosh models and subsequent PCI-based hardware platforms running Copland.

The Copland driver environment differs from the System 7 driver environment in several ways:

- The system distinguishes between software that runs in user mode or in supervisor mode. In System 7, drivers run in the same environment as applications in a single address space. In Copland, drivers run in supervisor mode and have access to the microkernel's protected memory space. Applications can't touch the hardware or the driver code or data directly.
- Drivers are packaged as Code Fragment Manager fragments (shared libraries).
- Distinct execution environments are defined in which different sets of services are available. Because drivers execute in supervisor mode, they cannot call Mac OS Toolbox routines. On the other hand, by executing in supervisor mode, drivers gain a fine granularity of control over devices and overall system responsiveness. Drivers use microkernel, driver, and family service libraries as appropriate. Families and their plug-ins are expected to adhere to the rules appropriate to their execution environment.
- The system employs new tasking and messaging mechanisms that allow prioritizing of I/O processing and that make I/O latency predictable. These mechanisms are the foundation for preemptive multitasking and memory protection.
- Drivers exist as plug-ins to a particular I/O family and must conform to the activation model employed by that family. Therefore, when writing your driver, you need to adhere to the plug-in programming interface and the family's implementation guidelines. I/O family provide libraries of commonly needed routines, thus simplifying your development effort.
- Drivers that touch hardware must be written in native PowerPC code. As a result, Copland will deliver superior I/O performance. Emulated 68K drivers that directly access hardware are not supported.

As a result of these changes, you need to change the way you write a device driver. With the exception of drivers written according to specifications for PCI-based Macintosh computers, System 7 drivers that access hardware will *not* run under Copland.

The next two sections give more information on the separation of application and device driver interfaces and the packaging of driver software and they describe benefits that result from these changes.

Separation of Application and Device Driver Interfaces

In System 7 there is only one kind of programming interface: the application programming interface (API). This makes all Mac OS services available to all varieties of software. Copland distinguishes between programming interfaces available to applications and those available to device drivers. Programming contexts become increasingly specialized in Copland.

In Copland, drivers have available to them plug-in programming interfaces specifically tuned to the needs of different types of devices, such as display devices or SCSI devices. The plug-in programming interfaces provide a fine level of control over core operating system facilities such as paging and interrupts. Use of plug-in programming interfaces is essential to your driver's portability in future Mac OS releases. These interfaces are guaranteed to be common across OS releases.

Because drivers operate outside the application software context in Copland, they do not have access to the rich set of APIs available to applications. If you find that a service you depend on has been removed from the plug-in programming interface for your driver, you should contact Apple at the AppleLink address NEW.IO or new.io@applelink.apple.com.

Common Packaging of Loadable Software

In Copland, all drivers are created as Code Fragment Manager (CFM) fragments (shared libraries). Each CFM fragment must export a driver description structure that the system uses to locate, load, and initialize the driver.

Copland drivers, therefore, are packaged differently from previous Macintosh device drivers. Because they are CFM fragments, they are allowed to have specific static data storage, and they can be written in a high-level language without assembly-language headers. Each instance of a single driver has private static data and shares code with every other instance of that driver. A device driver no longer locates its private data by means of a field in its Device Unit Table entry.

One consequence of drivers as CFM fragments is that a single device driver no longer controls multiple devices. Normally there is a driver instance for each device, although only one copy of the driver's code is loaded into memory.

If You Develop Applications

Adjusting to the architectural shift in the I/O system should be relatively easy for the application developer. For compatibility with System 7 applications, the Copland Device Manager supports all of the functions described in the chapter "Device Manager" of *Inside Macintosh: Devices*. However, a smaller set of devices will be available through the Device Manager; for them, the system supports a compatibility layer that converts old function calls to new ones. Thus, if your application calls the Device Manager, it will continue to run on Copland, but it will incur a performance penalty going through the compatibility layer.

For better performance and for access to services well suited to a given class of device, you should update your application to use the FPI for that device rather than the Device Manager. For example, if your application uses the Display Manager, you benefit from a set of routines tuned to work with display devices.

In most cases, Copland FPIs will be the same as or very similar to existing APIs, such as those provided by the File Manager, the Display Manager, and Open Transport. If your application uses these higher-level APIs, it is insulated from underlying changes in the Copland I/O architecture and Copland device drivers and you shouldn't have to change it to work with Copland.

In addition to benefiting from the more effective services available through Copland FPIs, adopting the new FPIs now facilitates subsequent development for versions of the Mac OS beyond Copland. APIs that Copland maintains for compatibility may not be available with versions of the Mac OS beyond Copland. For example, the networking paradigm for the Mac OS is changing, moving in the direction of Open Transport. Although Copland will support System 7 AppleTalk interfaces, later versions of the Mac OS will not. Versions of the Mac OS beyond Copland will require you to use the Open Transport FPI.

If your application ignores public APIs and instead uses nonstandard methods to access a device, you'll need to change your application. In Copland, hardware is not mapped into application address space and attempts to touch hardware will result in access violations. Devices and drivers are not directly accessible to an application. The only access to their services is through a family programming interface or an API maintained for compatibility.

Device Manager Compatibility

In Copland, the Device Manager functions described in the chapter "Device Manager" of *Inside Macintosh: Devices* are supported. Drivers that provide their services through the Device Manager API belong to the Device Manager family and are called **generic drivers**. The Device Manager functions constitute the FPI for the Device Manager family. The family has its own activation model and set of services, but it is not tuned to the needs of a given type of device.

Although the Device Manager API is more limiting than that provided by family FPIs, the Device Manager family offers a migration path to driver developers who implement the basic changes required by Copland without totally converting to the Copland I/O architecture.

If no family for a device exists, the Device Manager offers a way to use it in Copland. Consider, for example, a PCI card that receives data, encrypts it, and sends it back. An encryption family doesn't currently exist. By writing the driver according to the rules for drivers of family type `'ndrv'` described in *Designing PCI Cards and Drivers for Power Macintosh Computers*, the card is supported in Copland as a plug-in to the Device Manager family.

To summarize, the Copland Device Manager supports drivers that have been revised to run in Copland but that have not taken advantage of the enhanced driver services available through Copland I/O families, or for which no family exists. As a result, the Device Manager family's plug-ins are likely to differ quite a bit among themselves, rather than belonging to a general class of devices such as video monitors. For example, Device Manager family plug-ins may include drivers for instrumentation bus adapters, graphics devices, encryption hardware, and so forth. Typically, plug-ins in the Device Manager family are drivers that talk to hardware, but they can also talk to virtual devices such as a RAM disk or loopback software.