Technical Note TN1192
ATA Interface Modules

CONTENTS

This Technote describes how to write a device driver for an ATA host bus controller, known as an ATA Interface Module, or AIM. An AIM is the ATA equivalent of the SCSI Interface Module (SIM). It does not control a device on the ATA bus, but implements a standard hardware abstraction for the bus itself.

AIMs operate at the very lowest level of the traditional Mac OS I/O subsystem, which makes them hard to write and hard to debug. Only experienced Mac OS device driver writers should consider developing an AIM.

This Note is directed at developers of ATA host bus controller cards (typically PCI or CardBus).

Updated: [Dec 06 1999]

Introduction

This technote describes how to write a Mac OS device driver for ATA host bus controller hardware. Such a driver is known as an ATA Interface Module, or AIM. In some respects it is the ATA equivalent to the more commonly known SCSI Interface Module (SIM).

This technote is divided into two sections. This section, and the Core Concepts sections which follows it, represent the introductory material. They describe high-level issues with writing AIMs; understanding these sections is critical to writing a reliable AIM. The remaining sections are reference material. They describe how you can register an ATA bus with ATA Manager, how you should package your AIM so that it is recognized by ATA Manager, what entry points ATA Manager expects you AIM to implement, the list of function codes your AIM's action routine must support, and some support routines that ATA Manager exports to support AIM implementation.

IMPORTANT:
AIMs are only supported by ATA Manager 4.0 and above. Development of third-party ATA buses is not supported on older versions of ATA Manager. Device 0/1 support is not available on some early ATA Manager 4.0 systems; however, your AIM should always support device 0/1 operations.

Before You Begin

Before you consider developing an AIM, you should be familiar with the following concepts:

- The native driver model, as described in Designing PCI Cards and Drivers for Power Macintosh Computers.
- The Mac OS ATA programming interface, as described in ATA Device Software for Macintosh Computers and amended by the ATA Device 0/1 Software Developer's Guide and DTS Technote 1098 ATA Device Software Guide Additions and Corrections.
- The ATA bus protocol; ANSI NCITS 317-1998 *AT Attachment - 4 with Packet Interface Extension* is a good reference.

You will also need the following:

- "ATA.h" from the latest version of Universal Headers.
- If that version of "ATA.h" does not include the AIM-related declarations described in this technote, you will need the "ATAExtras.h" header file that is included with this technote. [2404935]
- Similarly, if the libraries included with Universal Headers does not include ATAManager stub library, you will need the one included here.
- Information about your bus controller from your hardware vendor.

Back to top

Core Concepts

Before starting your AIM, you should be familiar with some fundamental concepts, as described in the following sections.

Theory of Operation

AIMs represent an abstraction of an ATA bus. Your AIM isolates the ATA Manager and its clients from the specific details of your ATA bus controller hardware. For instance, an ATA bus may be part of a PC Card slot, or perhaps reside on a PCI card, or may even be integrated as part of a system chipset. Each physical ATA controller will have specific requirements for how it is addressed, what cycle times are supported and how they are programmed, how interrupts from the controller are routed to the system, and so on. By bundling the hardware-specific details of the ATA controller in a plug-in software module, the ATA Manager presents a consistent interface for drivers and applications to communicate with ATA and ATAPI devices.

All of the operations which directly touch the hardware in any manner are handled by your AIM. This includes:

- describing the features of the bus controller, primarily the supported I/O modes and cycle times, and controlling those features
- preparing physical DMA transfer buffers and building DMA programs for those buffers
- alignment issues
- reading and writing the device's task file registers
- writing ATAPI command packets
- handling ATA and ATAPI transfer operations
- servicing hardware interrupts
- timing out failed bus transactions
- resetting the bus

Current versions of ATA Manager will only dispatch a single request for a single device on the bus controlled by your AIM at any given time. Overlapped ATA/ATAPI features are not currently supported. AIMs written to the specification in this document will never be expected to handle concurrent requests.

In general, your AIM should be interrupt-driven. It should do as much work as it can on a request, then return to the ATA Manager pending an interrupt from the hardware. For instance, when you receive a request to read data, you would set your AIM's internal flags as needed, write the task file to the device, and then return to ATA Manager. When the device is ready to transfer data, it will assert an interrupt which will call your AIM's hardware interrupt handler. Your AIM would then call `ATAFamBusEventForAIM` to queue a secondary handler and return from the hardware interrupt handler. ATA Manager will then call your AIM's `MyHandleBusEvent` function where you would complete the data transfer, clear your AIM's internal flags, and call `ATAFamIODone`. It is important that all your cleanup be done before calling `ATAFamIODone`, because ATA Manager may call you with another request during `ATAFamIODone`. Also, your AIM's `MyHandleBusEvent` function may be called immediately when calling `ATAFamBusEventForAIM`. Other than these two cases, the AIM will not be called reentrantly.

There are some limits on the controller hardware that can be supported by an AIM:

- Each ATA bus must be capable of simultaneous operation, independent of any other ATA bus. If an ATA controller chip implements two bus controllers, but they share common wiring or a DMA engine such that only one bus can be active at a time, then only one bus may be supported under ATA Manager. ATA Manager has no synchronization mechanism between buses.

- ATA controllers incapable of generating interrupts are not supported.
- PCI ATA controllers which operate only in x86 "legacy mode" (that is, they hard-decode only the lower-address bits and cannot be relocated in PCI space) are not supported. However, it is possible to support addressing the controller in either memory or PCI I/O space.

Global Variables

Your AIM should allocate its per-bus global variables in a memory block (typically allocated use `PoolAllocateResident`) and use ATA Manager's per-AIM `refCon` facility to track its globals.

The reason to use the `refCon` (rather than C `extern` and `static` variables, which are stored in your fragment's data section) is that the ATA Manager has a facility to update your AIM to a newer version "on the fly." The process, described below, involves closing the CFM connection to the older version of your AIM and replacing it with a connection to the newer version. If you had important data in the older AIM's data section, that data is lost when the data section is disposed as part of closing the CFM connection. In contrast, ATA Manager explicitly passes the `refCon` to the new version of your AIM.

Note:
An AIM is a native driver (`'ndrv'`) and shares many of the properties of native drivers in general. One of these properties is that your AIM is instantiated once for each instance of your ATA hardware on the machine (technically, one per ATA node in the Name Registry). Therefore you can store per-bus global variables in your data section, and you will automatically get one copy of these global variables per bus.

However, you should avoid using this strategy for per-bus global variables for the reason described above.

It is acceptable to use your data section for global variables as long as you have a strategy for passing them between versions. You could, for example, store the data primarily in you data section, push it into the AIM's `refCon` memory block when your AIM is suspended, and restore it to your data section when your AIM is resumed. Or you might choose to pass data between versions of your AIM via Name Registry properties. However, both of these techniques are significantly more convoluted than using the AIM's `refCon`, hence the recommendation to just use the `refCon` approach.

Regardless of how you allocate your global variables, you must ensure that they are held resident in memory (in the virtual memory sense). All of the techniques described above guarantee this.

Finally, your AIM should use PowerPC structure alignment for its global variables. This has two advantages:

1. If you do atomic operations on a global variable, PowerPC alignment ensures that the variable is aligned appropriate.
2. PowerPC alignment yields better performance for native code than 68K alignment.

AIM Update Process

As mentioned above, the ATA Manager has a process for updating your AIM to a newer version "on the fly," without shutting the bus down. The ATA Manager accomplishes this in a 7 step process:

1. Open a CFM connection to the new AIM
2. Wait for all pending AIM requests to terminate
3. Block any new AIM requests from starting
4. Call the old AIM's suspend routine (`MyAIMSuspend`)
5. Call the new AIM's resume routine (`MyAIMResume`), passing it the `refCon` from the old AIM
6. Unblock the AIM to allow bus operations to continue
7. Close the CFM connection to the old AIM

There are a number of important things about this process to keep in mind.

- The ATA Manager finds replacement AIMs by matching the Name Registry node name to the `nameInfoStr` field of the `DriverType` structure of its driver description. In order for your AIM to be updated, these names much match exactly.
- In addition, the ATA Manager will only update an AIM if the `version` field of the `DriverType` structure of its driver description is newer than the version of the existing AIM.
- You must structure your per-bus globals such that they can be passed from one version of your AIM to the next. Some suggested techniques are:

  - Using your AIM's `refCon` to store a pointer to your per-bus storage simplifies the process of passing

this information between versions. See the previous section for more details.
    - Put the version number of your AIM in your per-bus storage so that the newer AIM knows which older version it is taking over from (and can compensate for known bugs, oversights, and so on).
    - Include some "reserved" fields in your per-bus storage to give future versions of your AIM some room for expansion.
    - You can also use Name Registry properties to pass information between versions of your AIM.

- AIM resume routines are not defined to return an error code. You should design your AIM so that its resume routine cannot fail.

AIM Synchronization Model

As long as you follow one simple rule, the ATA Manager takes care of most of the synchronization problems in writing an AIM. The rule is:

Note:
When your AIM is called by anyone other than ATA Manager, you must synchronize with ATA Manager by posting a bus event.

Posting a bus event (`ATAFamBusEventForAIM`) is a the way your AIM informs ATA Manager that its bus event handler (`MyAIMHandleBusEvent`) should be called. ATA Manager queues the bus event and eventually calls the bus event handler when it is safe to do so.

If you follow this rule, ATA Manager guarantees that it will never call your AIM on more than one thread of execution at a time. So as long as you're executing within the context of a routine that is called by ATA Manager, you can access global variables without worrying about synchronization issues.

An obvious example of where this is useful is for handling interrupts. As a rule, your AIM will service I/O requests as a state machine. When ATA Manager calls your AIM to start a request (`MyAIMAction`), your AIM will start an asynchronous I/O operation and then return to ATA Manager. When the asynchronous operation completes, your hardware will interrupt the processor, and system software will execute your interrupt service routine. This interrupt service routine executes at hardware interrupt time. If you access your global variables from it, you must worry about synchronizing that access with lower execution levels. The solution is for your AIM to post a bus event. ATA Manager will defer calling your bus event handler (`MyAIMHandleBusEvent`) until it can guarantee that it is the only thread of execution running inside your AIM. Your bus event handler can therefore safely access global variables without worrying about synchronization.

IMPORTANT:
Your interrupt service routine must perform the following tasks as quickly as possible:

- Identify the source of the interrupt.
- If the interrupt is from your device, clear the source of the interrupt and post a bus event.

The bulk of the work in handling an interrupt should be done in your bus event handler.

If your AIM's hardware is hosted on a PCI bus, you must be sure to handle hardware interrupts in an expansion chassis friendly fashion. See DTS Technote 1135 Dealing with PCI Expansion Chassis Problems for details.

Controllers which contain two ATA buses which share a single PCI interrupt must extend the interrupt source tree so that each ATA bus has a separate interrupt node. Typically this is done in your AIM's initialization routine.

A less obvious example of the use of bus events is to handle timeouts. If your AIM implements timeouts using the system timer service (`SetInterruptTimer`), the timer routine will be executed at hardware interrupt time. Your AIM can avoid synchronization problems by posting this event as a bus event as well.

Note:
ATA Manager uses the bus event mechanism to both guarantee synchronization and to defer the processing of bus events until interrupts are enabled (by way of a secondary interrupt). You should not rely on ATA Manager's use of secondary interrupts. Older versions of ATA Manager implemented this using a deferred task, and the implementation might change again in the future.

In addition to guaranteeing that only one thread of execution can be running inside your AIM at any point in time, the ATA Manager also guarantees to dispatch only a single request to your AIM at a time. Between the point when ATA Manager dispatches a request (by calling `MyAIMAction`) and the point when your AIM completes it (by calling `ATAFamIODone`), ATA Manager will not dispatch any further requests to your AIM. Instead, it will queue these requests on its internal queues. This "one request at a time" guarantee is in recognition of the fact that the ATA bus architecture does not support parallel overlapped requests and that, by guaranteeing this, ATA Manager simplifies your life.

Both of the synchronization guarantees described in this section are defined on a bus-by-bus basis. If your AIM is multiply instantiated on the system, and those instances share common data, you must be careful to synchronize access to that common data.

Finally, ATA Manager also handles enabling and disabling of user code (in the virtual memory sense) for you. Readers who are familiar with SIMs (the equivalent to AIMs for SCSI buses) know that they are required to call `EnteringSIM` and `ExitingSIM` whenever they enter or leave the SIM. AIM developers are not required to jump through that particular hoop because the ATA Manager knows when non-reentrant portions of the AIM are executing (because the non-reentrant parts of the AIM are always executed as a result of the ATA Manager calling the AIM) and so it can enable and disable user code appropriately.

ATA I/O Modes

This section explains one of the trickier aspects of the ATA Manager's API for setting transfer modes and timings. In addition to the discussion here, you should read the ATA Device Software for Macintosh Computers (and its errata, DTS Technote 1098 ATA Device Software Guide Additions and Corrections) carefully to fully understand how ATA client software expects your AIM to handle transfer modes and timings.

Prior to ATA Manager 3.0, which was the first version of the ATA Manager to support DMA transfers, transfer modes were specified as absolute numbers. Thus, a value of 2 for a transfer mode meant PIO mode 2. Starting with ATA Manager 3.0, transfer modes were specified as bitmaps. Thus a value of 1 meant transfer mode 0, a value of 2 meant transfer mode 1, and so on. The `ataModeType` field of the `ATAReqBlock` determines which mode this request is in.

This is important to remember when trying to establish the correct timing mode on the ATA bus. For example, the flag bit `mATAFlagUseConfigSpeed` has different meanings depending on the value of the `ataModeType` field. If the `ataModeType` field is set to `kATAModeAbsolute` (pre-ATA Manager 3.0) then the flag bit `mATAFlagUseConfigSpeed` indicates whether to use timing values for the mode last set with a Set Driver Configuration request, or to use timing values for the PIO mode specified in the field `ataPBIOSpeed`. If the `ataModeType` field is set to `kATAModeBitmap` (ATA Manager 3.0 and above) then the flag bit `mATAFlagUseConfigSpeed` should *always* be set. If so, your AIM must use the bus mode (PIO, singleword DMA, multiword DMA, UltraDMA) and timing values that were stored from the last Set Driver Configuration request. If `mATAFlagUseConfigSpeed` is not set, your AIM should execute the request at the slowest possible transfer speed.

AIMs versus SIMs

In many respects, AIMs are architecturally similar to SCSI Interface Modules (SIMs), the name given to host bus controller drivers in the Mas OS SCSI architecture. The following table compares various features of AIMs and SIMs.

Feature	**AIM**	**SIM**
Single Thread	Yes	No
Single Request	Yes	No
Interrupt Polling	Yes	Yes
Enable/Disable User Code	No	Yes
Per-Bus Storage	Maintained by ATA Manager	Maintained by SCSI Manager
Per-Request Storage	Single request, therefore put per-request data in per-bus globals	Allocated by client as part of SCSI parameter block
Update "on the fly"	Yes	Not provided by SCSI Manager
`'ndrv'`	Required	Possible, but not required

Back to top

ATA Manager Additions

ATA Manager 4.0 defines two new ATA Manager function codes to be used with the `ataManager` system call. The new functions are defined below. The codes are used to add and remove ATA buses, respectively.

```
enum {
    kATAMgrAddATABus              = 0x93,
    kATAMgrRemoveATABus           = 0x94
};
```

IMPORTANT:
If your AIM is loaded from an expansion ROM on a card, you do not need to register it manually with ATA Manager. At startup time, ATA Manager will search the Name Registry for ATA nodes and automatically register an ATA bus for any node with an available AIM. ATA Manager considers any node whose "device_type" property is `kATADeviceType` ("ata\15\0") to be an ATA node.

For compatibility reasons, ATA Manager also recognizes nodes of type "ide\0". New AIMs should use always `kATADeviceType`.

Adding an ATA Bus

To add an ATA bus to ATA Manager, you must call the `ataManager` system call, passing in a parameter block of type `ataAddATABus`.

```
struct ataAddATABus {
    ataPBHeader *                   ataPBLink;
    UInt16                          ataPBQType;
    UInt8                           ataPBVers;
    UInt8                           ataPBReserved;
    Ptr                             ataPBReserved2;
    ProcPtr                         ataPBCallbackPtr;
    OSErr                           ataPBResult;
    UInt8                           ataPBFunctionCode;
    UInt8                           ataPBIOSpeed;
    UInt16                          ataPBFlags;
    SInt16                          ataPBReserved3;
    UInt32                          ataPBDeviceID;
    UInt32                          ataPBTimeOut;
    Ptr                             ataPBClientPtr1;
    Ptr                             ataPBClientPtr2;
    UInt16                          ataPBState;
    UInt16                          ataPBSemaphores;
    SInt32                          ataPBReserved4;
    RegEntryIDPtr                   ataNameRegEntry;
    CFragConnectionID               connID;
    UInt32                          busID;
    UInt8                           flags;
    UInt8                           socketType;
    Ptr                             iconData;
    Ptr                             stringData;
};
typedef struct ataAddATABus ataAddATABus;
```

The fields have the following meaning:

`ataPBLink`
`ataPBQType`
`ataPBVers`
`ataPBReserved`
`ataPBReserved2`
`ataPBCallbackPtr`
`ataPBResult`
`ataPBFunctionCode`
`ataPBIOSpeed`
`ataPBFlags`
`ataPBReserved3`
`ataPBDeviceID`
`ataPBTimeOut`
`ataPBClientPtr1`
`ataPBClientPtr2`
`ataPBState`
`ataPBSemaphores`
`ataPBReserved4`

Standard ATA Manager parameter block header. See ATA Device Software for Macintosh Computers for details. You must initialize `ataPBFunctionCode` to `kATAMgrAddATABus`, `ataPBVers` to `kATAPBVers2`.

`ataNameRegEntry`

You must set this to the Name Registry node of the ATA bus which you wish to add. The ATA Manager will use Driver Loader Library to locate your native driver (AIM) for this bus. See AIM Packaging for more information about how your native driver must be structured.

`connID`

Reserved. You must set this field to zero and ignore any value returned.

`busID`

On successful completion of the request, the ATA Manager sets this field to the ATA bus ID of the newly created ATA bus.

`flags`

You must set this to the bus flags for this bus. The possible flags are defined below. The undefined flag bits are reserved; you must set them to zero.

`socketType`

You must set this to the ATA socket type of the bus using one of the constants defined in "ATA.h" (currently one of `kATASocketInternal`, `kATASocketMB`, or `kATASocketPCMCIA`).

`iconData`

You must set this to either a pointer to a black and white icon (256 bytes of data in `'ICN#'` format) that represents the ATA bus, or to `nil` if there is no such icon. ATA Manager makes a copy of the data, so you can dispose of it when the call completes.

`stringData`

You must set this to either a pointer to a string that describes the location of the ATA bus, or to `nil` if you do not wish to supply a location. The string is a C string (zero terminated) of at most 31 characters in the system script encoding; longer strings will be truncated by ATA Manager. ATA Manager makes a copy of the data, so you can dispose of it when the call completes.

In response to this request, the ATA Manager opens your AIM and creates the internal data structures necessary for it to track the AIM and its attached devices. As part of processing this call, the ATA Manager calls your AIM's initialization routine (`MyAIMInit`), which must probe your bus for attached devices. If your AIM's initialization routine returns an error, the ATA Manager cleans up and completes the request with that error. If your AIM's initialization routine succeeds and indicates that devices are attached to the bus, ATA Manager will issue `kATAUpdateEvent` and `kATAOnlineEvent` events for each device.

The possible flags for the `flags` field of the `ataAddATABus` structure are defined below.

```
enum {
    mATANoDMAOnBus                = 0x80
};
```

The meaning of these flags is:

`mATANoDMAOnBus`
: If this flag is set, the bus the ATA Manager's `kATAMgrBusInquiry` function will indicate that the bus does not support DMA, even if your AIM indicates that it does. This allows the bus expert which registers this bus to override AIM defaults for DMA support. The actual effect of this flag is that the ATA Manager clears the `ataSingleDMAModes`, `ataMultiDMAModes` and `ataUltraDMAModes` fields returned by your AIM in response to a `kATAFnBusInquiry` request before returning those fields as part of the client's `kATAMgrBusInquiry` request.

You must issue this request at system task time.

Removing an ATA Bus

To remove an ATA bus, you must call the `ataManager` system call, passing in a parameter block of type `ataAddATABus`.

```
struct ataRemoveATABus {
    ataPBHeader *                   ataPBLink;
    UInt16                          ataPBQType;
    UInt8                           ataPBVers;
    UInt8                           ataPBReserved;
    Ptr                             ataPBReserved2;
    ProcPtr                         ataPBCallbackPtr;
    OSErr                           ataPBResult;
    UInt8                           ataPBFunctionCode;
    UInt8                           ataPBIOSpeed;
    UInt16                          ataPBFlags;
    SInt16                          ataPBReserved3;
    UInt32                          ataPBDeviceID;
    UInt32                          ataPBTimeOut;
    Ptr                             ataPBClientPtr1;
    Ptr                             ataPBClientPtr2;
    UInt16                          ataPBState;
    UInt16                          ataPBSemaphores;
    SInt32                          ataPBReserved4;
    UInt32                          busID;
    RegEntryIDPtr                   ataNameRegEntry;
};
typedef struct ataRemoveATABus ataRemoveATABus;
```

The fields have the following meaning:

`ataPBLink`
`ataPBQType`
`ataPBVers`
`ataPBReserved`
`ataPBReserved2`
`ataPBCallbackPtr`
`ataPBResult`
`ataPBFunctionCode`
`ataPBIOSpeed`
`ataPBFlags`
`ataPBReserved3`
`ataPBDeviceID`

`ataPBTimeOut`
`ataPBClientPtr1`
`ataPBClientPtr2`
`ataPBState`
`ataPBSemaphores`
`ataPBReserved4`
: Standard ATA Manager parameter block header. See ATA Device Software for Macintosh Computers for details. You must initialize `ataPBFunctionCode` to `kATAMgrRemoveATABus`, `ataPBVers` to `kATAPBVers2`.

`busID`
: Reserved. You must set this field to zero and ignore any value returned.

`ataNameRegEntry`
: You must set this to the Name Registry node of the ATA bus which you wish to add.

In response to this request the ATA Manager will remove the ATA bus associated with the Name Registry node specified in `ataNameRegEntry`. The ATA Manager executes the following steps:

1. It calls your AIM's action routine (`MyAIMAction`) with the `kATAFnKillIO` function code, to indicate that your AIM should stop processing the current request.
2. It completes any pending I/O requests (including the current one) for your AIM with the `nsDrvErr` error code.
3. It calls your AIM's device light routine (`MyAIMDeviceLight`) to turn off the device's light.
4. It issues the `kATARemovedEvent` event for each device on your bus.
5. It calls your AIM's close routine (`MyAIMClose`), ignoring any error result.
6. It disposes of the resources it used to track the bus and releases its CFM connection to your AIM.

You may issue this request at any execution level, although if you issue it at anything other than system task level the ATA Manager's connection to the AIM is not closed until a future system task time.

Back to top

AIM Packaging

An AIM is packaged as a native driver (`'ndrv'`). For ATA Manager to load your AIM, it must be available to the Driver Loader Library. See Designing PCI Cards and Drivers for Power Macintosh Computers for more information on how Driver Loader Library finds native drivers.

Your AIM must export two named entry global variables:

- TheDriverDescription -- This is the standard native driver description.
- ThePluginDispatchTable -- This contains information specific to ATA Manager.

The structure of these global variables is described in the following sections.

TheDriverDescription

An AIM must export the standard native driver description structure under the name "TheDriverDescription". An example of how your AIM should fill out this structure is given below.

```
01 extern DriverDescription TheDriverDescription = {
02
03     // Signature
04
05     kTheDescriptionSignature,
06     kInitialDriverDescriptor,
07
08     // Driver Type
09
10     {
11         "\pMyAIMName",
12         kmajorRev, kminorAndBugRev, kstage, knonRelRev,
13     },
14
15     // OS Runtime Requirements of Driver
16
17     {
18         kDriverIsUnderExpertControl,
19         "\pMyAIMName",
20     },
21
22     // Service Category List
23
24     // Only one service category required
25
26     1,
27
28     // First Service Category
29
30     kServiceCategoryATA,
31     0,
32     1, 0, 0, 0
33 };
```

Some parts of this definition deserve further explanation.

Line 11
: You must set the `nameInfoStr` field of the `DriverType` structure to the name of the Name Registry node your driver controls. This allows the Driver Loader Library, and hence the ATA Manager, to associate your driver with the appropriate hardware.

Line 12
: You must set the `version` field of the `DriverType` structure the version number of your driver. This allows the Driver Loader Library to load the latest version of your driver if more than one happens to be installed. It also allows the ATA Manager to replace an initial ROM-based AIM with a newer file-based AIM once the File Manager is available.

Line 18
: AIMs are explicitly loaded by the ATA Manager (the expert), so you must set the `driverRuntime` flags of the `DriverOSRuntime` structure to `kDriverIsUnderExpertControl`.

Line 19
: The driverName field of the `DriverOSRuntime` structure is typically used to hold the unit table name of the driver. As AIMs are not installed in unit table, this field is not significant. You should set to the same string you used in Line 11.

Lines 26 through 32
: AIMs are required to export at least one service category, namely `kServiceCategoryATA`. This allows the ATA Manager to check that it correctly matched the AIM to the Name Registry node. The other fields of the `DriverServiceInfo` structure are reserved for future versions of the ATA Manager; you must initialize them as shown.

ThePluginDispatchTable

In addition to the standard drive description, you AIM must export a plugin dispatch table under the name "ThePluginDispatchTable". The `ATAPluginDispatchTable` type describes the format of this table.

```
struct ATAPluginDispatchTable {
    ATAPluginHeader                 header;
    ATAPluginInit                   init;
    ATAPluginClose                  close;
    ATAPluginAction                 action;
    ATAPluginHandleBusEvent         busEvent;
    ATAPluginPoll                   poll;
    ATAPluginEjectDevice            eject;
    ATAPluginDeviceLight            light;
    ATAPluginDeviceLock             lock;
    ATAPluginSuspend                suspend;
    ATAPluginResume                 resume;
};
typedef struct ATAPluginDispatchTable ATAPluginDispatchTable;

typedef OSStatus ATAPluginInit(ATAInitInfo *pb);
typedef OSStatus ATAPluginClose(UInt32 refCon, RegEntryIDPtr aimRegEntry);
typedef void ATAPluginAction(UInt32 refCon, ATAReqBlock *pb);
typedef void ATAPluginHandleBusEvent(UInt32 refCon, UInt32 aimData);
typedef Boolean ATAPluginPoll(UInt32 refCon, UInt32 interruptLevel, UInt32 *aimData);
typedef void ATAPluginEjectDevice(UInt32 refCon);
typedef void ATAPluginDeviceLight(UInt32 refCon, UInt32 whichDevice, UInt32 lightState);
typedef void ATAPluginDeviceLock(UInt32 refCon, UInt32 whichDevice, UInt32 lockState);
typedef void ATAPluginSuspend(UInt32 refCon);
typedef void ATAPluginResume(UInt32 refCon);
```

The fields have the following meaning:

`header`
: A header which describes the version of the overall plugin dispatch table. See below.

`init`
: You must set this to a pointer to your AIM's initialization routine (`MyAIMInit`).

`close`
: You must set this to a pointer to your AIM's close routine (`MyAIMClose`).

`action`
: You must set this to a pointer to your AIM's action routine (`MyAIMAction`).

`busEvent`
: You must set this to a pointer to your AIM's bus event handler routine (`MyAIMHandleBusEvent`).

`poll`
: You must set this to a pointer to your AIM's interrupt poll routine (`MyAIMPoll`).

`eject`
: You must set this to a pointer to your AIM's eject routine (`MyAIMEjectDevice`) or `nil` if your AIM does not have an eject routine.

`light`
: opt You must set this to a pointer to your AIM's device light routine (`MyAIMDeviceLight`) or `nil` if your AIM does not have an eject routine.

`lock`
: You must set this to a pointer to your AIM's device lock routine (`MyAIMDeviceLock`) or `nil` if your AIM does not have an eject routine.

`suspend`
: You must set this to a pointer to your AIM's suspend routine (`MyAIMSuspend`).

`resume`
: You must set this to a pointer to your AIM's resume routine (`MyAIMResume`).

The `ATAPluginHeader`, as defined below, structure describes the version of the overall plugin dispatch table.

```
struct ATAPluginHeader {
    NumVersion                      headerVersion;
    NumVersion                      dispatchVersion;
    UInt32                          reservedA;
    UInt32                          reservedB;
};
typedef struct ATAPluginHeader ATAPluginHeader;

enum {
    kATAPluginVersion           = 0x00000001,
    kATAPluginCurrentVersion    = kATAPluginVersion
};
```

The fields have the following meaning:

`headerVersion`
: You must set this to the version number of your AIM, as defined above.

`dispatchVersion`
: If your AIM conforms to this version of the specification, you must set this field to `kATAPluginVersion`.

`reservedA`
: Reserved. You must set this to zero.

`reservedB`
: Reserved. You must set this to zero.

Note:
ATA Manager does not currently look at the `dispatchVersion` field. Moreover, Apple's AIMs do not currently set this field correctly. Because of this confusion, any future version of ATA Manager that implements an extended plug-in interface will not use this field to determine which version of the plug-in interface that your AIM conforms to.

Back to top

AIM Entry Points

Your AIM must implement a number of routines and export those routines to the ATA Manager via the plugin dispatch table. This section describes these routines in detail.

MyAIMInit

```
extern OSStatus MyAIMInit(ATAInitInfo *aimInit);
```

`aimInit`	A pointer to an `ATAInitInfo` parameter block, described below.
result	An error code; see below for details.

The ATA Manager calls your AIM's initialization routine to commence operations on the ATA bus controlled by your AIM. This routine is called when an ATA bus is registered with ATA Manager. Your AIM must allocate any private resources it needs (typically per-bus storage), install its interrupt handler, initialize its hardware, probe the bus for attached devices (and determine whether they are ATA or ATAPI), and return information about those devices to the ATA Manager. Depending on your hardware, your AIM may need to reset the ATA bus to determine if any devices are attached.

Your AIM must also create child Name Registry nodes for the attached ATA devices. The nodes must have at least the following properties:

- The "name" property must be set to either "ata-disk" or "atapi-disk".
- The "device_type" property must be set to "block".
- The "device_id" property must be set to a `UInt32` that contains the ATA bus ID and device ID (in the same format as the `ataPBDeviceID` field of ATA Manager parameter blocks).

Your AIM initialization routine must not issue any commands to the ATA device.

Your AIM's initialization routine receives the address of an `ATAInitInfo` parameter block as a parameter. The parameter block contains both input and output fields.

```
struct ATAInitInfo {
    UInt32                          busID;
    ATADevInfo                      FirstDevice;
    ATADevInfo                      SecondDevice;
    RegEntryIDPtr                   aimRegEntry;
    UInt32                          refCon;
};
typedef struct ATAInitInfo ATAInitInfo;
```

The fields have the following meaning:

`busID`
: ATA Manager sets this to the ATA bus ID it has assigned to the new bus that is being registered.

`FirstDevice`
: Your AIM must initialize this structure to hold information about the first device on the ATA bus. See below for a description of the `ATADevInfo` structure. If the bus supports ATA devices 0 and 1, and device 1 is present but device 0 isn't, your AIM should set this structure to represent device 1 and set `SecondDevice` to indicate that no second device is attached.

`SecondDevice`
: Your AIM must initialize this structure to hold information about the second device on the ATA bus.

`aimRegEntry`
: The ATA Manager sets this to the Name Registry node of the ATA bus which is being registered.

`refCon`
: Your AIM may set this field to any 32-bit value, typically a pointer to your per-bus storage. The ATA Manager will pass this value as a parameter (typically named `refCon`) whenever it calls your other AIM entry points.

IMPORTANT:
The `AIMInitInfo` data structure and the structures it points to are deallocated as soon as your AIM returns from `MyAIMInit`. If you wish to retain access to this data, you must copy it to your own storage. Specifically, you should make a copy of the `RegEntryID` pointed to by `aimRegEntry`.
Copying the `RegEntryIDPtr` is not sufficient!

The `ATADevInfo` structure holds information about a specific ATA device on a bus. Your

```
struct ATADevInfo {
    UInt8                              devType;
    SInt8                              devID;
};
typedef struct ATADevInfo ATADevInfo;

enum {
    kATAInvalidDeviceID          = -1,
    kATADevice0DeviceID          =  0,
    kATADevice1DeviceID          =  1
};
```

The fields have the following meaning:

`devType`
: Your AIM must set this field to indicate the type of the attached device. Possible ATA device types are listed in "ATA.h" (current `kATADeviceUnknown`, `kATADeviceATA`, `kATADeviceATAPI`, and `kATADeviceReserved`).

`devID`
: Your AIM must set this field to indicate the ATA device ID of the attached device. Possible device IDs are `kATAInvalidDeviceID`, `kATADevice0DeviceID` and `kATADevice1DeviceID`.

If your AIM wants to indicate that no ATA device is attach, it must set devType to `kATADeviceUnknown` and `devID` to `kATAInvalidDeviceID`.

If your AIM initialization routine returns an error, or it indicates that there are no devices on the bus, ATA Manager fails the request to register the ATA bus, unloads your AIM, and dispose of all references to it.

IMPORTANT:
It is especially important to take note of the circumstances under which your AIM will be unloaded when extending the interrupt source tree for multiple bus controllers that share an interrupt. Remember, your buses may be initialized in any order. The first instance of an AIM that successfully initializes should extend the interrupt tree and store the child interrupt nodes for each bus in the Name Registry. When the second bus is initialized, it can look in the Name Registry to determine whether the interrupt tree has been extended or not.

Suggested result codes include:

- `noErr` Initialization successful.
- `memFullErr` Unable to allocate private data.
- `nsDrvErr` No device detected on node
- `paramErr` Bad parameter
- `nrInvalidNodeErr` Invalid Name Registry node
- `nrNotFoundErr` A required property was not found in the Name Registry node
- `ATAInitFail` Initialization failed

This routine is always be called at system task time.

MyAIMClose

```
extern OSStatus MyAIMClose(UInt32 refCon, RegEntryIDPtr aimRegEntry);
```

`refCon`	A pointer to your per-bus storage, as returned by `MyAIMInit`
`aimRegEntry`	The Name Registry node of the ATA bus which is being deregistered.
result	An error code; see below for details.

The ATA Manager calls your AIM's close routine to terminate operations on the ATA bus controlled by your AIM. This routine is called when an ATA bus is deregistered with ATA Manager. Your AIM must shut down its hardware, remove any interrupt handlers, and release any resources it owns. This will be the last request that a particular instance of your AIM will receive.

Any error code returned by your AIM is ignored. You should structure your AIM such that its close routine can not fail.

This routine will always be called at system task time.

MyAIMAction

```
extern void     MyAIMAction(UInt32 refCon, ATAReqBlock *pb);
```

`refCon`	A pointer to your per-bus storage, as returned by `MyAIMInit`
`pb`	A pointer to an `ATAReqBlock` parameter block, described below.

The ATA Manager calls your AIM's action routine to perform a transaction on the ATA bus. The `ATAReqBlock` parameter block specifies the action to perform and the place to store the results. The meaning of many of the fields is dependent on the `ataFunctionCode` field, which specifies exactly what operation is to be performed. Each function code is described in detail in AIM Action Function Codes.

```
struct ATAReqBlock {
    UInt32                          connectionID;
    UInt32                          MsgID;
    ATAResult *                     result;
    ATADiagResult *                 DiagResult;
    ATABusInfo *                    busInfo;
    ATADevConfig *                  devConfig;
    ATADataObject                   ioObject;
    ataTaskFile                     ataPBTaskFile;
    ATAPICmdPacket                  packetCBD;
    Duration                        Timeout;
    UInt32                          BusID;
    SInt8                           DevID;
    UInt8                           ataFunctionCode;
    UInt32                          AbortID;
    UInt32                          ataPBLogicalBlockSize;
    UInt32                          ataPBFlags;
    UInt32                          reserved;
    struct ATAReqBlock *            nextREQ;
    OSStatus                        ataPBResult;
    UInt8                           ataPBErrorRegister;
    UInt8                           ataPBStatusRegister;
    UInt32                          ataPBactualXferCount;
    UInt32                          ataPBState;
    UInt32                          ataPBSemaphores;
    UInt8                           XferType;
    UInt8                           ataModeType;
    UInt8                           ataPBIOSpeed;
    UInt8                           reserved2;
    UInt16                          reserved3;
};
typedef struct ATAReqBlock ATAReqBlock;
```

Unless otherwise stated, a field is has the same meaning for all function codes (except `kATAFnKillIO`). The fields have the following meaning:

`connectionID`
: Reserved. You must not modify this field or depend on its contents.

`MsgID`
: Reserved. You must not modify this field or depend on its contents.

`result`
: This field is used to hold the result of an AIM action. This field is a structure (`ATAResult`), not a simple '`ioResult`' value, but the fields of the structure are only relevant to `the` `kATAFnExecIO` function code. The only field relevant to the other function codes is `ataResult` field, but your AIM does not need to explicitly set this field because `ATAFamIODone` does it for you.

`DiagResult`
: This field is significant only for the `kATAFnRegAccess` function code and is described along with that function code.

`busInfo`
: This field is significant only for the `kATAFnBusInquiry` function code and is described along with that function code.

`devConfig`
: This field is significant only for the `kATAFnGetDriveConfig` and `kATAFnSetDriveConfig` function codes and is described along with those function codes.

`ioObject`
: This field is significant only for the `kATAFnExecIO` and `kATAFnDriveIdentify` function codes and is described along with those function codes.

`ataPBTaskFile`

This field is significant only for the `kATAFnExecIO` function code and is described along with that function code.

`packetCBD`
: This field is significant only for the `kATAFnExecIO` function code and is described along with that function code.

`Timeout`
: The ATA Manager sets this field to a timeout (in milliseconds) for the request. It derives the value from the `ataPBTimeOut` field of the client's request, or sets it to a default value (currently 31000) if `ataPBTimeOut` was zero.

`BusID`
: ATA Manager sets up this field to the bus ID of the ATA bus on which to perform the action. It derives this value from `ataPBDeviceID` field of the client's request. Typically your AIM ignores this field because the bus is already uniquely identified by the per-bus storage pointed to by the `refCon` passed to `AIMAction`.

`DevID`
: ATA Manager sets up this field to either `kATADevice0DeviceID` or `kATADevice1DeviceID` to describe which device on the ATA bus to act upon. It derives this value from the `ataPBDeviceID` field of the client's request.

`ataFunctionCode`
: ATA Manager sets up this field to describe the action that the AIM should take. AIM Action Function Codes lists the defined function codes.

`AbortID`
: ATA Manager does not use this field except insofar as to initialize it to zero. Your AIM can use this field as it sees fit.

`ataPBLogicalBlockSize`
: This field is significant only for the `kATAFnExecIO` function code and is described along with that function code.

`ataPBFlags`
: The ATA Managers sets this field to the value of the `ataPBFlags` field of the client's request. Your AIM is expected to read this field and act on the flags it contains.

`reserved`
: Reserved. You must not modify this field or depend on its contents.

`nextREQ`
: ATA Manager does not use this field except insofar as to initialize it to zero. Your AIM can use this field as it sees fit.

`ataPBResult`
: ATA Manager does not use this field except insofar as to initialize it to zero. Your AIM can use this field as it sees fit, although typically it is used to hold a temporary result.

`ataPBErrorRegister`
: ATA Manager does not use this field except insofar as to initialize it to zero. Your AIM can use this field as it sees fit, although typically it is used as temporary storage for the error register.

`ataPBStatusRegister`
: ATA Manager does not use this field except insofar as to initialize it to zero. Your AIM can use this field as it sees fit, although typically it is used as temporary storage for the status register.

`ataPBactualXferCount`
: ATA Manager does not use this field except insofar as to initialize it to zero. Your AIM can use this field as it sees fit, although typically it is used as temporary storage for the transfer count.

`ataPBState`
: ATA Manager does not use this field except insofar as to initialize it to zero. Your AIM can use this field as it sees fit, although typically it is used to track the state of the request.

`ataPBSemaphores`
: ATA Manager does not use this field except insofar as to initialize it to zero. Your AIM can use this field as it sees fit, although typically it is used to hold flags that indicate the status of the request.

`XferType`
: ATA Manager does not use this field except insofar as to initialize it to zero. Your AIM can use this field as it sees fit, although typically it is used to hold the I/O transfer type (PIO, single-word DMA, multi-word DMA).

`ataModeType`
: ATA Manager sets up this field to indicate whether the `ataPBIOSpeed` field contains an absolute value (`kATAModeAbsolute`) or a bitmap of possible values (`kATAModeBitmap`). It derives this value from the `ataPBVers` field of the client request; absolute mode is used only if the parameter block is less than version 3. See ATA I/O Modes for more information about how your AIM should interpret this and the `ataPBIOSpeed` field.

`ataPBIOSpeed`
: ATA Manager sets this field to the `ataPBIOSpeed` field of the client's request.

`reserved2`
: Reserved. You must not modify this field or depend on its contents.

`reserved3`
: Reserved. You must not modify this field or depend on its contents.

Each field is described in more detail with the appropriate function code.

Your AIM typically processes an action request using a state machine. When it receives the action request, it initializes the state machine and starts the first (asynchronous) step of processing the request. It then returns control to the ATA Manager. When the first step is complete, the hardware generates an interrupt and the AIM's interrupt service routine is called. It notifies ATA Manager of the bus event by calling `ATAFamBusEventForAIM`. ATA Manager then calls the AIM's bus event handler routine (`MyAIMHandleBusEvent`), which starts the next (asynchronous) step of processing the request. When the last step is done, the AIM calls `ATAFamIODone`. ATA Manager completes the original client's request and then calls the AIM to start the next request.

This routine may be called at any execution level.

MyAIMHandleBusEvent

```
extern void     MyAIMHandleBusEvent(UInt32 refCon, UInt32 aimData);
```

`refCon`	A pointer to your per-bus storage, as returned by `MyAIMInit`
`aimData`	The bus event type, as passed in to `ATAFamBusEventForAIM` or returned from `MyAIMPoll`; AIM Synchronization Model for more details

The ATA Manager calls your AIM's bus event handler routine to handle events detected on your bus. Typically these events are interrupts from your bus's interrupt hardware. Your AIM informs ATA Manager of these events in one of two ways:

- When an interrupt occurs (and is not masked), your interrupt service routine is called. Your interrupt service routine must inform ATA Manager of the bus event by calling `ATAFamBusEventForAIM` and then return.
- When interrupts are masked (or otherwise deferred), ATA Manager calls your AIM's interrupt poll routine (`MyAIMPoll`) to poll for masked interrupts. If the interrupt poll routine detects an interrupt, it returns an indicative status to ATA Manager.

Regardless of how it informs ATA Manager of the bus event, your AIM can provide a bus event type which indicates what type of bus event occurred. The ATA Manager passes the same bus event type back to the bus event handler in the `aimData` parameter. Typically this bus event type is used to distinguish the type of bus event that has occurred. For example, your AIM might use a value of 1 to indicate that a DMA interrupt has occurred and a value of 2 to indicate that an I/O interrupt has occurred. ATA Manager does not interrupt this value.

Typically your AIM responds to a bus event by moving the current I/O request to the next state. For example, if the current I/O request is waiting for an I/O completion bus event, your AIM would respond to that bus event by calling `ATAFamIODone` to inform ATA Manager that the I/O request is complete. On the other hand, if the current I/O request is not complete, your AIM would respond by setting up the next asynchronous I/O operation.

This routine may be called at any execution level. It is typically executed with interrupts enabled (either from a deferred task or a secondary interrupt) but that is not guaranteed.

MyAIMPoll

```
extern Boolean  MyAIMPoll(UInt32 refCon,
                          UInt32 interruptLevel,
                          UInt32 *aimDataPtr);
```

`refCon`	A pointer to your per-bus storage, as returned by `MyAIMInit`.
`interruptLevel`	The current 68K interrupt mask: a value from 0 to 7.
`aimDataPtr`	If your AIM detected a bus event, it should set the value pointed to by this parameter to the type of bus event that occurred; see AIM Synchronization Model for more details.
result	True if the AIM detected a bus event, otherwise false.

The ATA Manager calls your AIM's interrupt poll routine when it detects that a synchronous I/O request is blocked because interrupts are masked (or otherwise deferred). Your AIM must look at its interrupt hardware to determine if there is an interrupt pending. If there is, your AIM must set the memory pointed to by `aimDataPtr` to the type of bus event associated with that interrupt (see AIM Synchronization Model for more details on bus event types) and return true. If there is no interrupt pending, your AIM must return false.

The `interruptLevel` parameter is a convenience only. Your AIM may use this value to determine which interrupt sources to check. For example, if your AIM receives device interrupts at level 2 and DMA interrupts at level 4, and the current interruptLevel is 3, it need not check the status of the DMA interrupt line because that interrupt is not being masked.

This routine may be called at any execution level.

Background Material

Under the Mac OS I/O system architecture, it is possible for the system to take a page fault in three hard-to-handle cases:

- inside an interrupt handler (for example, Sound Manager callbacks are made directly from a sound hardware interrupt handler, but applications must be able to take page faults in these callbacks)
- when interrupts are masked (for example, the OS Utilities routine Enqueue will set the interrupt mask to 7 and then manipulate queue headers, which may be paged out)
- when critical system resources are busy (for example, the ATA Manager typically defers processing of bus events until secondary interrupt time, but secondary interrupts are serialized and a page fault from a hardware interrupt handler while an unrelated secondary interrupt was running would deadlock the system)

Page faults result in synchronous disk driver I/O requests. If the underlying I/O hardware requires interrupts to complete an I/O request, and interrupts are masked or otherwise deferred when the page fault happens, the system will deadlock.

In order to avoid this deadlock, the system polls for interrupts during any "sync wait" loop which occurs while interrupts are masked (or otherwise deferred). Given that the system has no knowledge of your AIM's interrupt architecture, it calls your AIM's interrupt poll routine to accomplish this polling.

See DTS Technote 1094 Virtual Memory Application Compatibility and DTS Q&A DV 34 Secondary Interrupts on the Page Fault Path for more details about this technique.

MyAIMEjectDevice

```
extern void     MyAIMEjectDevice(UInt32 refCon);
```

`refCon`	A pointer to your per-bus storage, as returned by `MyAIMInit`

The ATA Manager calls your AIM's eject routine in response to a `kATAMgrDriveEject` request.

IMPORTANT:
This routine ejects the entire ATA bus, not simply the media from a device on the bus. The distinction is a subtle but important one. An example of an AIM that implements the eject routine is the built-in PC Card AIM, which ejects the PC Card in response to this call.

If your ejection hardware is asynchronous, this operation should simply start the ejection operation. If an asynchronous ejection operation is not complete by the time the ATA bus is deregistered, your close routine (`MyAIMClose`) is responsible for waiting until it is.

This routine is optional. If your AIM does not support this function, it should set the appropriate plugin dispatch table entry to `nil`.

This routine may be called at any execution level.

MyAIMDeviceLight

```
extern void     MyAIMDeviceLight(UInt32 refCon,
                                 UInt32 whichDevice,
                                 UInt32 lightState);
```

`refCon`	A pointer to your per-bus storage, as returned by `MyAIMInit`
`whichDevice`	The ATA socket type of the device, derived from the `socketType` field of the `ataAddATABus` structure used to register the bus
`lightState`	The state in which to set the light: one of the constants given below

The ATA Manager calls your AIM's device light routine to turn on and off the activity light, if any, associated with your AIM. Typically the ATA Manager enables the activity light in response to a device driver request (the driver sets the `mATAFlagLEDEnable` flag in the `ataPBFlags` field of the parameter block it passes to the `ataManager` system call).

The constants for the `lightState` parameter are defined below.

```
enum {
    kATADeviceLightOff          = 0x00,
    kATADeviceLightOn           = 0x01
};
```

An example of an AIM that implements this routine is the media bay AIM for PowerBook computers, where it controls the LED on the media bay device.

This routine is optional. If your AIM does not support this function, it should set the appropriate plugin dispatch table entry to `nil`.

This routine may be called at any execution level.

MyAIMDeviceLock

```
extern void     MyAIMDeviceLock(UInt32 refCon,
                                UInt32 whichDevice,
                                UInt32 lockState);
```

`refCon`	A pointer to your per-bus storage, as returned by `MyAIMInit`
`whichDevice`	The ATA socket type of the device, derived from the `socketType` field of the `ataAddATABus` structure used to register the bus
`lockState`	Whether the device is locked or not; one of the constants given below

The ATA Manager calls your AIM's device lock routine to lock and unlock the hardware associated with your AIM. A locked device cannot be ejected by the user. The ATA Manager locks and unlocks the AIM based on the `mATApcLockUnlock` flag in the `atapcValid` field of the `ataDevConfiguration` parameter block supplied to a `kATAMgrSetDrvConfiguration` request.

The constants for the `lockState` parameter are defined below.

```
enum {
    kATADeviceUnlock            = 0x00,
    kATADeviceLock              = 0x01
};
```

An example of an AIM that implements this routine is the built-in PC Card AIM, which prevents users from ejecting the PC Card if it has been locked.

This routine is optional. If your AIM does not support this function, it should set the appropriate plugin dispatch table entry to `nil`.

This routine may be called at any execution level.

MyAIMSuspend

```
extern void     MyAIMSuspend(UInt32 refCon);
```

`refCon`	A pointer to your per-bus storage, as returned by `MyAIMInit`

The ATA Manager calls your AIM's suspend routine as part of the process of updating an AIM to a newer version. See AIM Update Process for more details.

Your AIM must "disconnect" itself from all system callbacks except those directly associated with the ATA Manager. This includes:

- interrupt handlers
- timer tasks
- power management callbacks

This allows the ATA Manager to unload the code associated with the older version of your AIM. Your AIM must not dispose of the per-bus storage associated with `refCon`. ATA Manager will pass the same `refCon` to the resume routine of the newer AIM, which is responsible for "reconnecting" the AIM to the system.

The ATA Manager guarantees that it will not call your AIM from the beginning of your suspend routine until your resume routine returns.

This routine will always be called at system task time.

MyAIMResume

```
extern void     MyAIMResume(UInt32 refCon);
```

`refCon`	A pointer to your per-bus storage, as returned by `MyAIMInit`

The ATA Manager calls your AIM's resume routine as part of the process of updating an AIM to a newer version. See AIM Update Process for more details.

Your AIM must "redisconnect" itself to all system callbacks which were disconnected by the suspend routine. This includes:

- interrupt handlers
- timer tasks
- power management callbacks

Your AIM is required to continue operations using the per-bus storage inherited from the older version and pointed to by `refCon`. If the format of your per-bus storage changes between versions, your resume routine must convert from the old to the new format.

The ATA Manager guarantees that it will not call your AIM from the beginning of your suspend routine until your resume routine returns.

This routine will always be called at system task time.

Back to top

AIM Action Function Codes

When ATA Manager calls your AIM's action routine (`MyAIMAction`), it sets the `ataFunctionCode` field of the `ATAReqBlock` parameter block to a value which identifies the type of operation to be performed. The possible function codes are listed below:

```
enum {
    kATAFnNOP                   = 0x00,
    kATAFnExecIO                = 0x01,
    kATAFnBusInquiry            = 0x02,
    kATAFnQRelease              = 0x03,
    kATAFnCmd                   = 0x04,
    kATAFnAbort                 = 0x05,
    kATAFnBusReset              = 0x06,
    kATAFnRegAccess             = 0x07,
    kATAFnDriveIdentify         = 0x08,
    kATAPIFnExecIO              = 0x09,
    kATAPIFnCmd                 = 0x0A,
    kATAFnGetDriveConfig        = 0x0B,
    kATAFnSetDriveConfig        = 0x0C,
    kATAFnKillIO                = 0x0D
};
```

The following sections describe each function code in detail. If your AIM receives a request with a function code it does not recognize, it should fail the request with a status of `paramErr`.

No Operation (kATAFnNOP)

This is a "no operation" request. ATA Manager should never issue this request to your AIM. If it does, your AIM should immediately complete the request successfully by calling ATAFamIODone with a status of noErr.

Execute I/O (kATAFnExecIO)

ATA Manager issues this request to your AIM as the result of a client's kATAMgrExecIO request. Your AIM is expected to execute the specified I/O transaction to the specified device on the specified bus. The bulk of the ATAReqBlock is set up as described above; only the fields specific to this request are described here.

The structure of the result field of the ATAReqBlock is shown below.

```
struct ATAResult {
    OSStatus                        ataResult;
    SInt8                           ataStatusRegister;
    SInt8                           ataErrorRegister;
    UInt32                          actualXferCount;
    ataTaskFile *                   TaskFile;
};
typedef struct ATAResult ATAResult;
```

The fields have the following meaning:

ataResult
: The overall error result for the request. Unlike the other fields in the ATAResult structure, this field is relevant for all function codes. Your AIM must not explicitly set this field because ATAFamIODone does it for you.

ataStatusRegister
: For execute I/O requests, your AIM must set this field to the contents of the ATA status register. When you complete the request, ATA Manager copies this field to the ataPBStatusRegister field of the client's parameter block.

ataErrorRegister
: For execute I/O requests, your AIM must set this field to the contents of the ATA error register. When you complete the request, ATA Manager copies this field to the ataPBErrorRegister field of the client's parameter block.

actualXferCount
: For execute I/O requests, your AIM must set this field to number of bytes actually transferred. When you complete the request, ATA Manager copies this field to the ataPBActualTxCount field of the client's parameter block.

TaskFile
: For execute I/O requests, your AIM must copy the current contents of the ATA task file to the structure pointed to by this field, if it is not nil. Typically you copy this directly from the ataPBTaskFile field of the ATAReqBlock. ATA Manager does not currently look at this field.

The structure of the ioObject field of the ATAReqBlock is shown below.

```
struct ATADataObject {
    UInt8 *                         ioBuf;
    UInt32                          Count;
};
typedef struct ATADataObject ATADataObject;
```

The meaning of the fields in the ATADataObject structure are dependent on whether scatter/gather is enabled for this request. Scatter/gather is enabled if mATAFlagUseScatterGather is set in the ataPBFlags for the request. If scatter/gather is not enabled, the fields have the following meaning:

`ioBuf`
: ATA Manager sets this field to point to the start of the data buffer for the transfer.

`Count`
: ATA Manager sets this field to the count of the number of bytes to transfer.

If scatter/gather is enabled, the fields have the following meaning:

`ioBuf`
: ATA Manager sets this field to point to an array of `IOBlock` structures (defined in "ATA.h"). Your AIM should transfer the data between the device and the scatter/gather buffer defined by this array.

`Count`
: ATA Manager sets this field to the number of `IOBlock` structures in the array pointed to by `ioBuf`.

The `ataPBTaskFile` field of the `ATAReqBlock` has the same structure as the `ataPBTaskFile` field of the `ataIOPB` (defined in "ATA.h"). Before issuing an execute I/O request to your AIM, ATA Manager copies, without interpretation, the task file from the client's `ataIOPB` to the `ataPBTaskFile` field of the `ATAReqBlock`. [It does, however, force the `mATADriveSelect` bit of the `ataTFSDH` field of the task file based on the `devID` field of the `ATAReqBlock`.] When your AIM completes the request, the ATA Manager copies the `ataPBTaskFile` field back to the client's `ataIOPB`.

The `packetCBD` field of the `ATAReqBlock` has the same structure as the `ATAPICmdPacket` type defined in "ATA.h". By default, the ATA Managers clears this field before issuing an execute I/O request to your AIM. However, if the client issued an ATAPI request (`mATAFlagProtocolATAPI` was set in the `ataPBFlags` and `ataPBPacketPtr` was not nil), ATA Manager copies the `ATAPICmdPacket` pointed to by `ataPBPacketPtr` into the `packetCBD` field of the `ATAReqBlock`.

For execute I/O requests, the ATA Manager sets the `ataPBLogicalBlockSize` field of the `ATAReqBlock` to `ataPBLogicalBlockSize` field of the client's request.

Bus Inquiry (kATAFnBusInquiry)

ATA Manager issues this request to your AIM as the result of a client's `kATAMgrBusInquiry` request. Your AIM is expected to return information about the specified ATA bus. The bulk of the `ATAReqBlock` is set up as described above; only the fields specific to this request are described here.

The structure of the `busInfo` field of the `ATAReqBlock` is shown below:

```
struct ATABusInfo {
    UInt8                           ataPIOModes;
    UInt8                           ataSingleDMAModes;
    UInt8                           ataMultiDMAModes;
    UInt8                           ataUltraDMAModes;
    UInt32                          ataIOPBsize0;
    UInt32                          ataIOPBsize1;
    SInt8                           ataContrlType[16];
    NumVersion                      ataHBAversion;
    UInt32                          reserved3;
};
typedef struct ATABusInfo ATABusInfo;
```

The fields have the following meaning:

`ataPIOModes`
: Your AIM must set this field to a bit mask representing the PIO transfer modes it supports. On completion of the request, ATA Manager copies this field into the `ataPIOModes` field of the client's request.

`ataSingleDMAModes`
: Your AIM must set this field to a bit mask representing the singleword DMA transfer modes it supports. On completion of

the request, ATA Manager copies this field into the `ataSingleDMAModes` field of the client's request (unless the bus was registered with the `mATANoDMAOnBus` flag, in which case this field is ignored and the ATA Manager clears the `ataSingleDMAModes` field of the client's request).

`ataMultiDMAModes`
: Your AIM must set this field to a bit mask representing the multiword DMA transfer modes it supports. On completion of the request, ATA Manager copies this field into the `ataMultiDMAModes` field of the client's request (unless the bus was registered with the `mATANoDMAOnBus` flag, in which case this field is ignored and the ATA Manager clears the `ataMultiDMAModes` field of the client's request).

`ataUltraDMAModes`
: Your AIM must set this field to a bit mask representing the UltraDMA transfer modes it supports. On completion of the request, ATA Manager copies this field into the `ataUltraDMAModes` field of the client's request (unless the bus was registered with the `mATANoDMAOnBus` flag, in which case this field is ignored and the ATA Manager clears the `ataUltraDMAModes` field of the client's request).

`ataIOPBsize0`
: Your AIM must set this field to `kATADefaultBlockSize` (512). This field was originally intended to hold the size of an ATA sector on device 0, but developments in the ATA standard (namely ATAPI) have obviated the need for this information.

`ataIOPBsize1`
: Your AIM must set this field to `kATADefaultBlockSize` (512). This field was originally intended to hold the size of an ATA sector on device 1, but developments in the ATA standard (namely ATAPI) have obviated the need for this information.

`ataContrlType`
: Your AIM may set this field to any value, including characters or binary data. The field is intended as a mechanism to report a vendor or model name, or other data for identification or diagnostic purposes. If you do not implement this functionality, you should clear the entire field. On completion of the request, ATA Manager copies this field into the `ataContrlType` field of the client's request.

`ataHBAversion`
: Your AIM must put its version number in this field. On completion of the request, ATA Manager copies this field into the `ataHBAversion` field of the client's request.

`reserved3`
: Reserved. You must not modify this field or depend on its contents.

I/O Queue Release (kATAFnQRelease)

Requests of this type should never be passed through to your AIM. Your AIM should treat this as an unrecognized function code.

ATA Command (kATAFnCmd)

Requests of this type should never be passed through to your AIM. Your AIM should treat this as an unrecognized function code.

Abort Command (kATAFnAbort)

Requests of this type should never be passed through to your AIM. Your AIM should treat this as an unrecognized function code.

Reset ATA Bus (kATAFnBusReset)

ATA Manager issues this request to your AIM as the result of a client's `kATAMgrBusReset` request. Your AIM is expected to reset the specified ATA bus. The `ATAReqBlock` is set up as described above; there are no fields specific to this request.

Register Access (kATAFnRegAccess)

ATA Manager issues this request to your AIM as the result of a client's `kATAMgrRegAccess` request. Your AIM is expected to read or write the specified ATA registers. The bulk of the `ATAReqBlock` is set up as described above; only the fields specific to this request are described here.

> **IMPORTANT:**
> Your AIM must determine whether to read or write the ATA registers based on the `mATAFlagIOWrite` flag in the `ataPBFlags` field of the `ATAReqBlock`.

The structure of the `DiagResult` field of the `ATAReqBlock` is shown below.

```
struct ATADiagResult {
    UInt16                          ataRegMask;
    OSStatus                        ataResult;
    UInt16                          ataDataReg;
    UInt8                           ataTFFeatures;
    UInt8                           ataTFCount;
    UInt8                           ataTFSector;
    UInt8                           ataTFCylinderLo;
    UInt8                           ataTFCylinderHi;
    UInt8                           ataTFSDH;
    UInt8                           ataTFCommand;
    UInt8                           ataAltStatDevCnt;
};
typedef struct ATADiagResult ATADiagResult;
```

The fields have the following meaning:

`ataRegMask`
: The ATA Manager sets bits in this field to indicate which registers to read or write. The bit mask are defined in "ATA.h" (`mATAAltSDevCValid`, `mATAStatusCmdValid`, `mATASDHValid`, and so on).

`ataResult`
: Reserved. You must not modify this field or depend on its contents.

`ataDataReg`
: For a write operation, your AIM must write this field to the ATA data register (always a 16-bit write) if `mATADataValid` is set in `ataRegMask`. For a read operation, your AIM must read the ATA data register (always a 16-bit read) and put it in this field if `mATADataValid` is set in `ataRegMask`.

`ataTFFeatures`
: For a write operation, your AIM must write this field to the ATA error register if `mATAErrFeaturesValid` is set in `ataRegMask`. For a read operation, your AIM must read the ATA features register and put it in this field if `mATAErrFeaturesValid` is set in `ataRegMask`.

`ataTFCount`
: For a write operation, your AIM must write this field to the ATA sector count register if `mATASectorCntValid` is set in `ataRegMask`. For a read operation, your AIM must read the ATA sector count register and put it in this field if `mATASectorCntValid` is set in `ataRegMask`.

`ataTFSector`
: For a write operation, your AIM must write this field to the ATA sector number register if `mATASectorNumValid` is set in `ataRegMask`. For a read operation, your AIM must read the ATA sector number register and put it in this field if `mATASectorNumValid` is set in `ataRegMask`.

`ataTFCylinderLo`
: For a write operation, your AIM must write this field to the ATA cylinder low register if `mATACylinderLoValid` is set in `ataRegMask`. For a read operation, your AIM must read the ATA cylinder low register and put it in this field if `mATACylinderLoValid` is set in `ataRegMask`.

`ataTFCylinderHi`
: For a write operation, your AIM must write this field to the ATA cylinder high register if `mATACylinderHiValid` is set in `ataRegMask`. For a read operation, your AIM must read the ATA cylinder high register and put it in this field if `mATACylinderHiValid` is set in `ataRegMask`.

`ataTFSDH`
: For a write operation, your AIM must write this field to the ATA SDH register if `mATASDHValid` is set in `ataRegMask`. For a read operation, your AIM must read the ATA SDH register and put it in this field if `mATASDHValid` is set in `ataRegMask`.

`ataTFCommand`
: For a write operation, your AIM must write this field to the ATA command register if `mATAStatusCmdValid` is set in `ataRegMask`. For a read operation, your AIM must read the ATA status register and put it in this field if

`mATAStatusCmdValid` is set in `ataRegMask`.

`ataAltStatDevCnt`
: For a write operation, your AIM must write this field to the ATA device control register if `mATAAltSDevCValid` is set in `ataRegMask`. For a read operation, your AIM must read the ATA alternate status register and put it in this field if `mATAAltSDevCValid` is set in `ataRegMask`.

Drive Identify (kATAFnDriveIdentify)

ATA Manager issues this request to your AIM as the result of a client's `kATAMgrDriveIdentify` request. Your AIM is expected to execute an ATA drive identify command (`kATAcmdDriveIdentify`). This request is very similar to a standard execute I/O request except for the following:

- Your AIM must force the I/O to be byte swapped (typically by setting `mATAFlagByteSwap` in `ataPBFlags`).
- Your AIM must always attempt to transfer 512 bytes (typically by overwriting the `Count` field of the `ioObject` with 512).
- Your AIM must always use a 512 byte logical block size (typically be overwriting `ataPBLogicalBlockSize` with 512).
- Your AIM must issue an ATA drive identify command (typically by overwriting the `ataTFCommand` field of the `ataPBTaskFile` with `kATAcmdDriveIdentify` and the `ataTFSDH` of the `ataPBTaskFile` with `mATASectorSize`).

Once it has modified the parameter block in this way, your AIM can simply pass this request through to the execute I/O logic.

Execute ATAPI I/O (kATAPIFnExecIO)

Requests of this type should never be passed through to your AIM. Your AIM should treat this as an unrecognized function code.

ATAPI Command (kATAPIFnCmd)

Requests of this type should never be passed through to your AIM. Your AIM should treat this as an unrecognized function code.

Get Drive Configuration (kATAFnGetDriveConfig)

ATA Manager issues this request to your AIM as the result of a client's `kATAMgrGetDrvConfiguration` request. Your AIM is expected to return information about a device's current configuration. The bulk of the `ATAReqBlock` is set up as described above; only the fields specific to this request are described here.

The structure of the `devConfig` field of the `ATAReqBlock` is shown below.

```
struct ATADevConfig {
    SInt32                          ataConfigSetting;
    UInt8                           ataPIOSpeedMode;
    UInt8                           reserved;
    UInt16                          atapcValid;
    UInt16                          ataRWMultipleCount;
    UInt16                          ataSectorsPerCylinder;
    UInt16                          ataHeads;
    UInt16                          ataSectorsPerTrack;
    UInt16                          ataSocketNumber;
    UInt8                           ataSocketType;
    UInt8                           ataDeviceType;
    UInt8                           atapcAccessMode;
    UInt8                           atapcVcc;
    UInt8                           atapcVpp1;
    UInt8                           atapcVpp2;
    UInt8                           atapcStatus;
    UInt8                           atapcPin;
    UInt8                           atapcCopy;
    UInt8                           atapcConfigIndex;
    UInt8                           ataSingleDMASpeed;
    UInt8                           ataMultiDMASpeed;
    UInt16                          ataPIOCycleTime;
    UInt16                          ataMultiCycleTime;
    UInt8                           ataUltraDMASpeed;
    UInt8                           reserved2;
    UInt16                          ataUltraCycleTime;
    UInt16                          Reserved1[5];
};
typedef struct ATADevConfig ATADevConfig;
```

The fields have the following meaning:

`ataConfigSetting`
: Your AIM must set this field to indicate the device configuration settings currently in use. The possible values are defined in ATA Device Software for Macintosh Computers. On completion of the request, ATA Manager copies this field into the `ataConfigSetting` field of the client's request.

`ataPIOSpeedMode`
: Your AIM must set this field to indicate which PIO models are enabled for this device. On completion of the request, ATA Manager copies this field into the `ataPIOSpeedMode` field of the client's request.

`reserved`
: Reserved. You must not modify this field or depend on its contents.

`atapcValid`
: Reserved. You must not modify this field or depend on its contents. On completion of the request, ATA Manager updates the `mATApcLockUnlock` flag to indicate whether the device is currently locked.

`ataRWMultipleCount`
`ataSectorsPerCylinder`
`ataHeads`
`ataSectorsPerTrack`
: Reserved. **Your AIM must set these fields to zero.** On completion of the request, ATA Manager copies this field into the corresponding fields of the client's request.

`ataSocketNumber`
: Reserved. You must not modify this field or depend on its contents. This field was used in previous versions of ATA Manager (which handled much of the PC Card socket configuration internally) but is now obsolete, replaced by functionality in PC Card Manager.

`ataSocketType`
: Your AIM should ignore this field; on completion, ATA Manager will set it based on your AIM's socket type.

`ataDeviceType`
: Your AIM must set this to the type of the device (for example, `kATADeviceATA` or `kATADeviceATAPI`) specified by the `devID` and `busID` fields of the `ATAReqBlock`. Typically your AIM returns a copy of the information it derived during initialization (`MyAIMInit`) On completion of the request, ATA Manager copies this field into the `ataDeviceType` field of the client's request.

`atapcAccessMode`
: Reserved. You must not modify this field or depend on its contents. This field is obsolete with ATA Manager 4.0. It was previously defined to support different access modes for PC Card devices but that support was never implemented.

`atapcVcc`
`atapcVpp1`
`atapcVpp2`
`atapcStatus`
`atapcPin`
`atapcCopy`
`atapcConfigIndex`
: Reserved. You must not modify these fields or depend on their contents. These fields were used in previous versions of ATA Manager (which handled much of the PC Card socket configuration internally) but are now obsolete, replaced by functionality in PC Card Manager.

`ataSingleDMASpeed`
: Your AIM must set this field to indicate which singleword DMA speeds are enabled for this device. On completion of the request, ATA Manager copies this field into the `ataSingleDMASpeed` field of the client's request.

`ataMultiDMASpeed`
: Your AIM must set this field to indicate which multiword DMA speeds are enabled for this device. On completion of the request, ATA Manager copies this field into the `ataMultiDMASpeed` field of the client's request.

`ataPIOCycleTime`
: Your AIM must set this field to the current minimum cycle time (in milliseconds) for mode 3 or greater PIO transfers. On completion of the request, ATA Manager copies this field into the `ataPIOCycleTime` field of the client's request.

`ataMultiCycleTime`
: Your AIM must set this field to the current minimum cycle time (in milliseconds) for multiword DMA transfers. On completion of the request, ATA Manager copies this field into the `ataMultiCycleTime` field of the client's request.

`ataUltraDMASpeed`
: Your AIM must set this field to indicate which UltraDMA speeds are enabled for this device. On completion of the request, ATA Manager copies this field into the `ataUltraDMASpeed` field of the client's request.

`reserved2`
: Reserved. You must not modify this field or depend on its contents.

`ataUltraCycleTime`
: Your AIM must set this field to the current minimum cycle time (in milliseconds) for UltraDMA transfers. On completion of the request, ATA Manager copies this field into the `ataUltraCycleTime` field of the client's request.

`Reserved1`
: Reserved. You must not modify this field or depend on its contents.

Set Drive Configuration (kATAFnSetDriveConfig)

ATA Manager issues this request to your AIM as the result of a client's `kATAMgrSetDrvConfiguration` request. Your AIM is expected to set the device's current configuration based on the supplied parameter block. The bulk of the `ATAReqBlock` is set up as described above; only the fields specific to this request are described here.

IMPORTANT:
To understand how your AIM should interpret the various I/O mode and cycle time fields of this request, see the ATA I/O section, earlier in this document.

The `devConfig` field of the `ATAReqBlock` is defined above. For a `kATAFnSetDriveConfig` request, the fields have the following meaning:

`ataConfigSetting`
: The ATA Manager sets this field to required device configurations settings. The value is derived from the `ataConfigSetting` field of the client request. The possible values are defined in ATA Device Software for Macintosh Computers. Your AIM must act on these configuration settings for all subsequent I/O operations.

`ataPIOSpeedMode`
: The ATA Manager sets this field to required PIO speed mode for the device. The value is derived from the

`ataPIOSpeedMode` field of the client request. Your AIM must use this PIO speed for all subsequent PIO transfers.

`reserved`
: Reserved. You must not modify this field or depend on its contents.

`atapcValid`
: Reserved. You must not modify this field or depend on its contents. This field was used in previous versions of ATA Manager (which handled much of the PC Card socket configuration internally) but is now obsolete, replaced by functionality in PC Card Manager. Note that ATA Manager still honors the `mATApcLockUnlock` flag in this field by calling your device lock (MyAIMDeviceLock) routine as part of handling a `kATAMgrSetDrvConfiguration` request.

`ataRWMultipleCount`
`ataSectorsPerCylinder`
`ataHeads`
`ataSectorsPerTrack`
: Reserved. You must not modify this field or depend on its contents.

`ataSocketNumber`
: Reserved. You must not modify this field or depend on its contents. This field was used in previous versions of ATA Manager (which handled much of the PC Card socket configuration internally) but is now obsolete, replaced by functionality in PC Card Manager.

`ataSocketType`
: Reserved. You must not modify this field or depend on its contents.

`ataDeviceType`
: Reserved. You must not modify this field or depend on its contents.

`atapcAccessMode`
: Reserved. You must not modify this field or depend on its contents. This field is obsolete with ATA Manager 4.0. It was previously defined to support different access modes for PC Card devices but that support was never implemented.

`atapcVcc`
`atapcVpp1`
`atapcVpp2`
`atapcStatus`
`atapcPin`
`atapcCopy`
`atapcConfigIndex`
: Reserved. You must not modify these fields or depend on their contents. These fields were used in previous versions of ATA Manager (which handled much of the PC Card socket configuration internally) but are now obsolete, replaced by functionality in PC Card Manager.

`ataSingleDMASpeed`
: The ATA Manager sets this field to the required singleword DMA speed modes for the device. The value is derived from the `ataSingleDMASpeed` field of the client request. It is only valid if `ataModeType` is `kATAModeBitmap`. Your AIM may use these speeds for all subsequent singleword DMA transfers.

`ataMultiDMASpeed`
: The ATA Manager sets this field to the required multiword DMA speed modes for the device. The value is derived from the `ataMultiDMASpped` field of the client request. It is only valid if `ataModeType` is `kATAModeBitmap`. Your AIM may use these speeds for all subsequent multiword DMA transfers.

`ataPIOCycleTime`
: The ATA Manager sets this field to the required maximum PIO cycle time for the device. The value is derived from the `ataPIOCycleTime` field of the client request. It is only valid if `ataModeType` is `kATAModeBitmap`. Your AIM may use this, or a slower time, for subsequent PIO transfers.

`ataMultiCycleTime`
: The ATA Manager sets this field to the required maximum multiword DMA cycle time for the device. The value is derived from the `ataMultiCycleTime` field of the client request. It is only valid if `ataModeType` is `kATAModeBitmap`. Your AIM may use this, or a slower time, for subsequent multiword DMA transfers.

`ataUltraDMASpeed`
: The ATA Manager sets this field to the required UltraDMA speed modes for the device. The value is derived from the `ataUltraDMASpeed` field of the client request. It is only valid if `ataModeType` is `kATAModeBitmap`. Your AIM may use these speeds for all subsequent UltraDMA transfers.

`reserved2`
: Reserved. You must not modify this field or depend on its contents.

`ataUltraCycleTime`
: The ATA Manager sets this field to the required maximum UltraDMA cycle time for the device. The value is derived from the `ataUltraCycleTime` field of the client request. It is only valid if `ataModeType` is `kATAModeBitmap`. Your AIM may use this, or a slower time, for subsequent UltraDMA transfers.

`Reserved1`

Reserved. You must not modify this field or depend on its contents.

Kill Current I/O (kATAFnKillIO)

The ATA Manager issues this request to your AIM as part of the process of removing your ATA bus. You AIM must respond to this request by terminating any hardware transaction on the ATA bus. This is an immediate request: your AIM must complete the request before returning from its action routine (`MyAIMAction`) and must not call `ATAFamIODone` for the request.

> **IMPORTANT:**
> `kATAFnKillIO` is different from other action requests in that none of the standard `ATAReqBlock` fields are set up for `kATAFnKillIO`. The only valid field in the `ATAReqBlock` for a `kATAFnKillIO` request is the function code itself (`ataFunctionCode`).

Back to top

AIM Support Routines

This section describes the AIM support routines exported by the ATA Manager for convenience of AIMs. Your AIM must use the routines described below to signal the ATA Manager that certain events have occurred.

ATAFamIODone

```
extern void ATAFamIODone(ATAReqBlock *theReq, OSStatus result);
```

`theReq`	The action request to complete
`result`	The final status of the request, either `noErr` or a negative error code

Your AIM must call this routine to inform ATA Manager that the AIM action request is complete. ATA Manager executes the following steps:

1. It copies information from the AIM request block (`theReq`) into the client's ATA request block.
2. If stores `result` in the `ataResult` field of the client's request block.
3. It calls the client's completion routine, if one was supplied.
4. It dispatches the next ATA request, if any, to the AIM's action routine.

You must call this routine from the context of your AIM's action routine (`MyAIMAction`) or its bus event handler (`MyAIMHandleBusEvent`).

ATAFamBusEventForAIM

```
extern void ATAFamBusEventForAIM(UInt32 busID, UInt32 aimData);
```

`busID`	The ATA bus on which the event occurred
`aimData`	The bus event type; the ATA Manager does not interpret this value, it simply passes it back to your AIM's bus event handler (`MyAIMHandleBusEvent`)

Your AIM must call this routine when it wants to scheduled its bus event handler (`MyAIMHandleBusEvent`) to be executed. Typically it does this from a hardware interrupt handler. ATA queues the bus event and calls your AIM's bus event handler at the next opportune moment.

See AIM Synchronization Model for an in-depth discussion of why this is both necessary and convenient.

You may call this routine at any execution level.

References

Designing PCI Cards and Drivers for Power Macintosh Computers

ATA Device Software for Macintosh Computers

ATA Device 0/1 Software Developer's Guide

DTS Technote 1098 ATA Device Software Guide Additions and Corrections.

ANSI NCITS 317-1998 *AT Attachment - 4 with Packet Interface Extension*

DTS Technote 1094 Virtual Memory Application Compatibility

DTS Q&A DV 34 Secondary Interrupts on the Page Fault Path

Back to top

Downloadables

	Acrobat version of this Note (128K).	Download
	AIM Interfaces and Libraries (8 KB)	Download

Back to top

Technical Notes by API | Date | Number | Technology | Title
Developer Documentation | Technical Q&As | Development Kits | Sample Code