STORAGE

Marvell Solutions

Serial ATA Software Driver User Manual for the 88SX50xx and 88SX60x1 Adapters

Software Version 3.2.0

Doc. No. MV-S800188-00, Rev. 0.9
January 29, 2004
Not approved by Document Control. For review only.

Pre-release Draft
Not Approved by Document Control

MOVING FORWARD FASTER®

MARVELL®

Document Status	
Advanced Information	This document contains design specifications for initial product development. Specifications may change without notice. Contact Marvell Field Application Engineers for more information.
Preliminary Information	This document contains preliminary data, and a revision of this document will be published at a later date. Specifications may change without notice. Contact Marvell Field Application Engineers for more information.
Final Information	This document contains specifications on a product that is in final release. Specifications may change without notice. Contact Marvell Field Application Engineers for more information.
Revision Code:	
Preliminary	Technical Publication:

Document Conventions

Note

Provides related information or information of special importance.

Caution

Indicates potential damage to hardware or software, or loss of data.

Warning

Indicates a risk of personal injury.

Table of Contents

Section 1. Architectural Specification

1.1 Introduction

The 88SX50xx /88SX60x1 is a PCI/PCI-X to 4/8 port Serial ATA (SATA) adapter that provide connectivity to SATA storage devices.

This document describes the software driver architecture of the 88SX50xx /88SX60x1. This architecture provides the system integrator (referred to in the driver documentation as the "user") to ramp up a system better and faster, using the 88SX50xx /88SX60x1, without the need for thorough knowledge of the adapter itself.

The 88SX50xx /88SX60x1 software driver architecture consists of the following components: (from bottom to top):

- CORE driver
- System dependent header file (mvOs.h)
- SCSI to ATA translation layer (SAL)
- Common Intermediate Application Layer Tasks (Common IAL)
- Intermediate Application Layer (IAL)
- Application Layers

1.1.1 Relevant Devices

This document is relevant for the following devices:

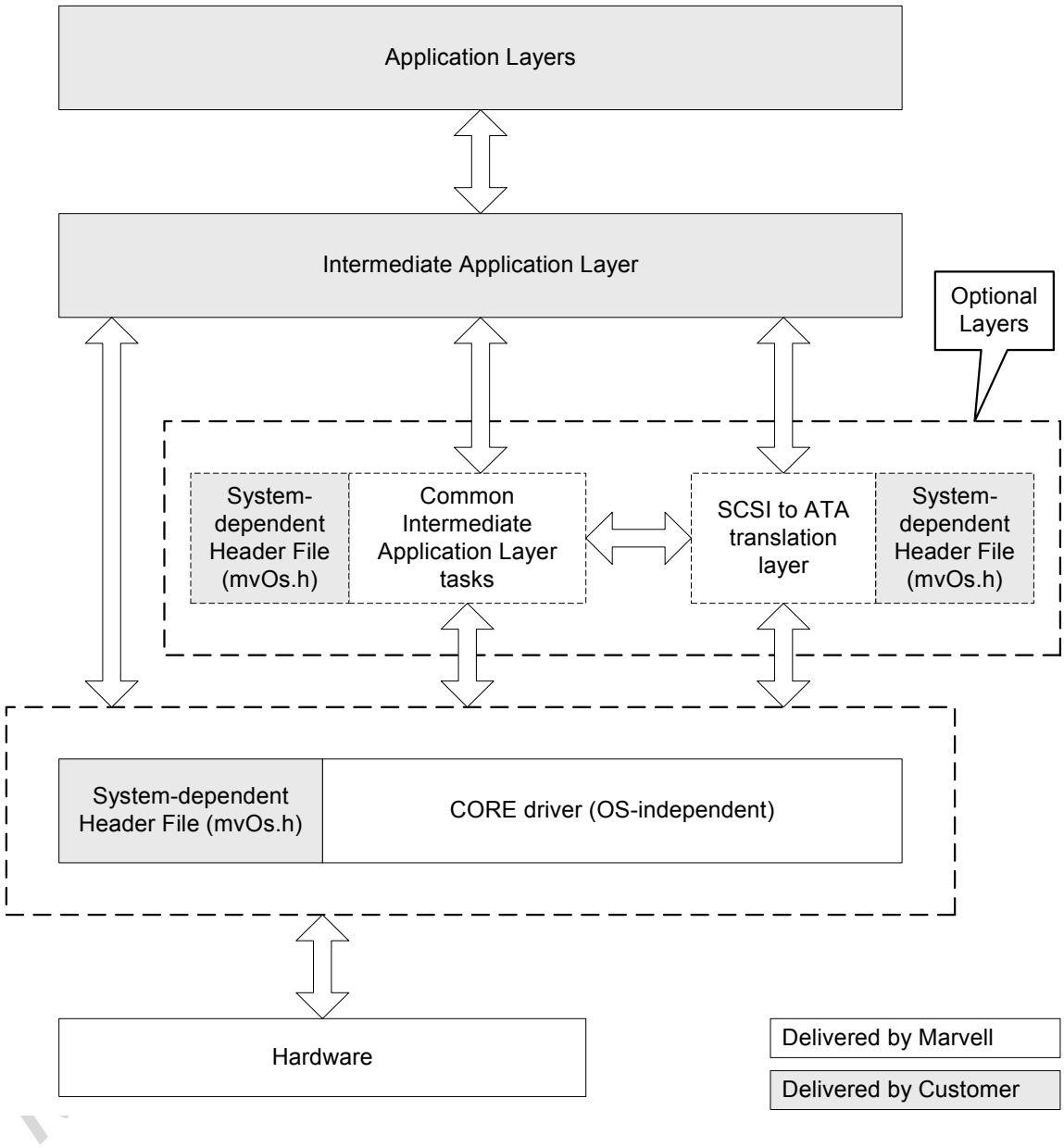
- 88SX5040, 88SX5041, 88SX5080, 88SX5081
- 88SX6081 and 88SX6041

1.1.2 Relevant Documents

For further information regarding the 88SX50xx /88SX60x1, see the following datasheets:

- 88SX5040, 88SX5041, 88SX5080, & 88SX5081 PCI/PCI-X to 8-Port Serial ATA Host Controller (Document Control number MV-S101357-00).
- 88SX6081 and 88SX6041 PCI/PCI-X to 8-Port Serial-ATA STorage Controller (Document Control Number MV-S101495-00).

Figure 1: 88SX50xx /88SX60x1 Software Driver Architecture



1.2 CORE Driver

The CORE driver is operating-system-independent source code that manages all accesses to the hardware that are needed for a system. The structure of the CORE driver and its API make possible easy integration of a single or multiple 88SX50xx /88SX60x1s in a system.

When the CORE driver is compiled with the system-dependent header file (mvOs.h) provided by the user, it generates code to which the IAL and other layers can connect, using the CORE driver API. It also generates code that can access the hardware using functions supplied by mvOs.h.

The CORE driver provides the following functionality:

- 88SX50xx /88SX60x1 adapter management, initialization, diagnostics and status reporting.
- Executes UDMA ATA commands.
- Executes non-UDMA ATA commands.
- Manages command completion and events notification, based on call-back functions.
- Interrupt Service.

Note

The 88SX50xx /88SX60x1 CORE driver is completely ANSI-C compliant source code.

1.3 System-Dependent Header File (mvOs.h)

The system-dependent single header file is named "mvOs.h".

The purpose of this header file is to provide extensions to the CORE driver and other layers that enable it to access system resources, lock and unlock resources (by using semaphores) and log events.

The purposes of this file are to:

- Provide an extension to the CORE driver and other layers, for accessing system resources such as memory and PCI buses.
- Define data types.
- Provide a data structure and function library for the CORE driver, for initializing, locking and unlocking semaphores.
- Provide a function that generates a delay in micro-seconds resolution.
- Provide a function for log messages, with a type of log message differentiator.
- Provide a function for printing a formatted string into a buffer.

1.4 SCSI to ATA Translation Layer (SAL)

The SCSI to ATA Translation Layer is an operating-system-independent layer that provides functionality for translating SCSI commands to ATA commands.

Such a layer is not mandatory layer. Typically it is used in cases where the IAL connects to a SCSI subsystem and the requests that are being handled in the IAL scope are SCSI commands. For example, if a SCSI VERIFY command is forwarded to the IAL, then the command can be forwarded as is to the SAL for translation to an ATA command, then the SAL forwards this command to the CORE driver for execution using the CORE driver API.

The SCSI to ATA translation layer provides the following functionality:

- Translation of certain SCSI commands with immediate completion (without queuing to CORE driver).

- Translation of certain SCSI commands to ATA and then further queuing to CORE driver.
- Error reporting of failing SCSI commands through SCSI sense code.

Note

The SCSI to ATA translation layer is tested with external tools that cover all cases and options of the supported SCSI commands.

1.5 Common Intermediate Application Layer Tasks (Common IAL)

Common IAL Tasks is an operating-system-independent layer. This layer is optional. It provides various functionality that are usually required by IALs.

This layer is used with the SCSI to ATA translation layer when the IAL connects to a SCSI subsystem and utilizes SCSI to ATA translation layer functionality for translating SCSI commands to ATA commands.

The Common IAL Tasks layer provides the following functionality:

- Discovery of storage devices connected to a serial ATA channel using point-to-point based connectivity and port-multiplier-based connectivity.
- Initialization of storage devices connected to serial ATA channels.
- Parsing of IDENTIFY DEVICE ATA command response buffer.
- Access of TWSI interface on some of the 88SX60X1 devices.

Due to the fact that part of the Common IAL uses a SCSI to ATA translation layer API for performing the discovery of storage devices, this part can be compiled and linked only when SAL is available.

Note that the SAL does not have this dependency and it can be compiled and linked with the driver with any part of Common IAL being available.

1.6 Intermediate Application Layer (IAL)

The IAL is user-specific code that functions as an intermediate layer between the higher application layers and the CORE driver.

Marvell provides three different IALs which, when integrated with the CORE Driver, SAL and Common IAL, provide a Linux SCSI low-level driver, Windows SCSI mini-port, and a BIOS extension driver.

The IAL definition is according to the system requirements, but what the different IAL implementations have in common is the way they connect to the CORE driver (and possibly SAL and Common IAL layers), i.e., by using the CORE driver API and data structures.

Examples of application layers to which the IAL can connect:

- **IAL connects to SCSI subsystem:** In this case, the function of the IAL connecting to an application-layer-specific SCSI interface, receiving SCSI commands and forwarding them to the SAL for SCSI to ATA translation then for further execution using the CORE driver.
- **IAL connects to ATA subsystem:** In this case, the IAL functions as glue software for delivering the ATA commands received from the ATA subsystem to the CORE driver.
- **IAL does not connect to application layers:** This case can be used for verification of a single or multiple 88SX50xx /88SX60x1s in a newly integrated system.

The IAL provides the following functionality:

From the Application layer's point of view:

- Representation of 88SX50xx /88SX60x1 adapters and their channels and storage devices to the Application layers.
- Proper command and request reception from the Application layers and proper completion of them.
- Proper error propagation of events to the Application layers

From the CORE driver API's point of view:

- Trigger 88SX50xx /88SX60x1 initialization sequences through the CORE driver API.
- Management of 88SX50xx /88SX60x1 SATA channels and the storage devices connected to them, through the CORE driver API.
- Translation/Delivery/Generation of ATA commands and their delivery to hardware, through the CORE driver API.
- Proper scheduling of commands and requests to hardware, through the CORE driver API.
- Error handling and reporting, through the CORE driver API call-back functions.
- Calls CORE driver ISR function.

From the SAL API's point of view:

- Forwarding SCSI commands to SAL for execution by the 88SX50xx /88SX60x1.

From the Common IAL API's point of view:

- Trigger of storage device discovery and initialization sequences.
- Parsing of IDENTIFY DEVICE ATA buffer response.

1.7 Application Layers

The Application layers are the core of the system. When the IAL API is used, the Application layers access the 88SX50xx /88SX60x1 adapter, SATA channels, and storage devices.

Example of Application Layers:

- A specific operating system to whose SCSI sub-system the IAL connects. (IAL is a SCSI low-level driver under Linux, a SCSI mini-port under Windows, etc.)
- RAID subsystem to which the IAL connects as a translation and command scheduling layer, to perform tasks on the hardware through the CORE driver.

Section 2. System Integration

2.1 Introduction

This section describes methods of integrating the OS-independent components of the Marvell Serial ATA host bus adapter software layers into a user-specific software driver. It is divided into two main sections:

- System integration using only CORE driver (described in Section 2.2).
- System integration using CORE driver, SCSI to ATA translation layer, and Common IAL layer (described in Section 2.3).

An example of system integration—Marvell Windows SCSI-port (described in Section 2.4)—is provided. This example is based on the system integration method described in Section 2.3).

Section 2.5 "Miscellaneous Issues" has miscellaneous issues involving system integration issues.

The OS-independent components discussed in this section are:

- **CORE driver**: Provides low-level access to hardware with queuing interface and interrupt service routine (see Section 6. "Core Driver" on page 42).
- **SCSI to ATA translation layer**: Provides functionality for translating SCSI commands to ATA commands and queuing capability to hardware using the CORE driver (see Section 7. "SCSI to ATA Translation Layer" on page 83). This functionality is usually required when IAL connects to SCSI subsystems.
- **Common IAL layer**: Provides functionality usually required by IALs. The code is written in OS-independent coding style (see Section 8. "IAL Common Layer" on page 94).

Note

This document refers to the software layers that control all the OS-independent components as "high" layers or "higher" layers. These software layers consist of the IAL and additional higher layers.

2.2 System Integration Using Only CORE Driver

This system integration method is suitable for systems that need to access adapters for executing ATA commands. A good example of this is a RAID stack that requests read/write I/Os. These I/Os can be translated into ATA read/write commands (UDMA or PIO commands) and queued using the CORE driver. The CORE driver handles all queuing, command completion, and error handling for the request commands.

In this type of system integration the following components/functionality must exist:

- Coding of system-dependent header file (mvOs.h), which enables CORE driver accessing system resources (described in Section 6.6 "System-Dependent Header File (mvOs.h)" on page 75).
- Hardware detection and CORE driver initialization.
- Storage devices detection and initialization.
- Command queuing, execution and completion.
- Error handling.

2.2.1 System-Dependent Header File (mvOs.h)

This file includes macros and possibly function calls that provide the CORE driver with the capability to access system resources.

This file is user-supplied. For details see Section 6.6 "System-Dependent Header File (mvOs.h)" on page 75.

2.2.2 Hardware Detection, Adapter, and CORE Driver Initialization

Higher layers scan the PCI bus/buses for detection of 88SX50xx /88SX60x1 adapters. For each adapter found the following steps must be performed by higher layers:

1. Initializes the Base Addresses registers (BARs) found in each adapter's PCI configuration space (BAR).
2. The system integrator must decide if register access to 88SX50xx /88SX60x1 adapter is performed through I/O-BAR or Memory-BAR (and accordingly define the register access in the mvOs.h file).
3. Enable memory and I/O accesses to adapter and enable adapter's master capability (bits 0, 1, and 2 in Command register in PCI configuration space).
4. Allocate and initialize (assign zero to all fields) the MV_SATA_ADAPTER data structure and 4/8 MV_SATA_CHANNEL data structures (depending on the amount of serial ATA channels the adapter supports).
5. Allocate 4/8 request and response queues (1 KByte for each request queue and 256 bytes for each response queue). See the 88SX50xx /88SX60x1 datasheets regarding alignment restrictions that each request/response queue must have.

Note

The request and response queues must be cache-coherent. For systems that have hardware cache coherency assist, the allocation for request and response queues is a simple memory allocation that is reachable by the adapter from PCI address space. For systems that don't have cache coherency hardware assist, allocate request and response queues that have non-cacheable attributes when being accessed.

6. Set the MV_SATA_ADAPTER data structure variables:
   a) Set the `adapterId` field for a value unique to that specific 88SX50xx /88SX60x1. (In debug mode this field is used by the CORE driver for log messages.)
   b) Set the adapter `pciConfigDeviceId` to the PCI device ID of the adapter, as reported on the adapter's PCI configuration space.
   c) Set the adapter `pciConfigRevisionId` to the PCI revision ID of the adapter, as reported on the adapter's PCI configuration space.
   d) Set the `adapterIoBaseAddress` field to the CPU address that enables access from the CPU to the specific 88SX50xx /88SX60x1 adapter being initialized. The address can be mapped into either memory-BAR or
   IO-BAR.
   e) Set `intCoalThre` and `intTimeThre` threshold fields to the required values for using the interrupt coalescing mechanism. Setting them both to zero indicates that interrupt coalescing thresholds are set to minimum, which achieves the same results as disabling them.
   f) Set the `mvSataEvetNotify` field to point to the user-implemented function used by the CORE driver as a callback function for event notification.
   g) Set the `sataChannel` pointer to zero.
   h) Set `pciCommand`, `pciSerrMask` and `pciInterruptMask` to the required values. (See the 88SX50xx / 88SX60x1 datasheet for further information about these fields.)

7. Call the `mvSataInitAdapter()` function to start initialization of the adapter.
   See Section 6.5.3.1 "CORE Driver Adapter Management" on page 55) for an explanation of the functionality of `mvSataInitAdapter()`.
8. Set up the adapter's interrupt line to trigger a higher layers interrupt service routine wrapper upon interrupt generation.
9. If the adapter supports staggered spinup, start an OOB sequence by calling either `mvSataEnableStaggeredSpinUp()` per serial ATA channel for performing an OOB sequence on the serial ATA channels one by one, or alternatively call the `mvSataEnableStaggeredSpinupAll()` function to perform an OOB sequence of all serial ATA channels in parallel.
10. Call the `mvSataUnmaskAdapterInterrupt()` function to enable interrupt assertion by the adapter.

2.2.3 Storage Devices Detection and Initialization

After the hardware detection and initialization described above has been completed, there are two possible states per SATA channel—connected or disconnected.

If a specific SATA channel is not connected, then there is no need for further initialization of the specific SATA channel.

Note

To determine whether the SATA channel is connected to/disconnected from a storage device, use the `mvSataIsStorageDeviceConnected()` CORE driver function.

If a SATA channel is connected, then higher layers must perform the following algorithms to detect and initialize the storage device connected to the specific SATA channel:

1. Storage device discovery algorithm: Checks whether the SATA channel is directly connected to a hard drive or to a port multiplier.
2. Initialization of hard drive algorithm: Reads the hard drive's IDENTIFY DEVICE data, parses the data, and accordingly issues SET FEATURES ATA commands.
3. Initialization of port multiplier: This algorithm is a preparation algorithm to perform the "Initialization of hard drive algorithm" for each hard drive connected to the device SATA ports of the port multiplier.
4. Configuring EDMA mode.

2.2.3.1 Storage Device Discovery Algorithm

This is the first algorithm to be executed when a Serial ATA channel is connected.

1. From the MV_SATA_CHANNEL data structures allocated in Section 2.2.2 "Hardware Detection, Adapter, and CORE Driver Initialization" initialize the MV_SATA_CHANNEL data structure corresponding to the SATA channel being initialized as follows:
   a) Set corresponding `sataChannel` pointers in MV_SATA_ADAPTER channel to point to the MV_SATA_CHANNEL chosen.
   b) Set `channelNumber` to the index number of the SATA channel.
   c) Set the `requestQueue`, `requestQueuePciHiAddress`, and `requestQueuePciLowAddress` fields to point to the request queue allocated in Section 2.2.2 "Hardware Detection, Adapter, and CORE Driver Initialization" . The `requestQueue` parameter is the CPU address to the request queue and `requestQueuePciHiAddress` and `requestQueuePciLowAddress` are the DMA addresses to the request queue.

d) Set `responseQueue`, `responseQueuePciHiAddress`, and `responseQueuePciLowAddress` as performed in the previous step, except for the response queue.

2. Call the `mvSataConfigureChannel()` CORE driver function.
3. Initiate software reset protocol using the `mvStorageDevATAStartSoftResetDevice()` CORE driver function. If the adapter supports port multiplier, then the destination port multiplier port must be port 15 (0xF), otherwise the destination port must be 0 (default). Note that after software reset protocol has been completed, the ATA status register equals 0x80, which indicates disk busy.
4. Poll and wait for signature FIS (aka register device to host FIS or FIS 34) to be received. The polling can be achieved by calling `mvStorageIsDeviceBsyBitOff()`, which reads the ATA Status register and returns MV_TRUE if the BSY bit switched from '1' to '0'.
5. If the signature FIS received is a port multiplier signature, then perform the algorithm described in Section 2.2.3.2 "Initialization of the Port Multiplier" .
6. If the signature FIS received is a hard drive signature, then perform the algorithm described in Section 2.2.3.3 "Initialization of Hard Drive Algorithm" .

2.2.3.2 Initialization of the Port Multiplier

The algorithm described in this section is intended for port multiplier initialization.

This is a preparation algorithm, which detects the number of SATA channels the port multiplier has, and for each SATA channel, initializes the hard drive connected to it (if any).

The higher layers must perform the following steps:

1. Query the port multiplier regarding its vendor, features and capabilities. As a result of the query the higher layers know how many device ports are connected to the port multiplier.
   See the `mvGetPMDeviceInfo()` function in the Common IAL layer (Section 8.5.2.1 "Common IAL Helper Functions" on page 103) for reference on how to perform the query.
2. For all device ports on the port multiplier, perform the following:
   a) Trigger an OOB sequence on the device port of the port multiplier. This can be achieved by calling `mvPMDevEnableStaggeredSpinUp()`, or alternatively calling `mvPMDevEnableStaggeredSpinUpAll()`, which triggers an OOB sequence on all device ports of the port multiplier.
   b) Read the Status register of the port multiplier's device port (using the `mvPMDevReadReg()` CORE driver function). If a hard drive is connected to the port multiplier's device port, perform the following steps. Otherwise skip to the next port multiplier's device port.
   - Clear the SError register of the port multiplier's device port (using the `mvPMDevWriteReg()` CORE driver function). This enables the hard drive to send FISes to the adapter's host port.
   - Perform the algorithm described in Section 2.2.3.3 "Initialization of Hard Drive Algorithm" for the specific port multiplier's device port.
   - Continue initialization of the next hard drives connected to the other port multiplier's device ports.

2.2.3.3 Initialization of Hard Drive Algorithm

The algorithm described in this section initializes a hard drive.

The algorithm can be executed when the hard drive is either connected directly to the adapter's SATA channel or through the port multiplier's device port.

For the first option, when the hard drive is connected directly to the adapter, perform the algorithm described in this section, then perform the steps described in Section 2.2.3.4 "Configuring EDMA Mode" .

For the second option, every function call to the CORE driver's functions must have the port multiplier's device port as an input to the function.

The higher layers must perform the following steps:

1. Initiate the software reset protocol using the `mvStorageDevATAStartSoftResetDevice()` CORE driver function. If the adapter supports port multiplier, then the destination port multiplier port must be port 15 (0xF), otherwise the destination port must be 0 (default). Note that after software reset protocol has been completed, the ATA Status register equals 0x80, which indicates disk busy.
2. Poll and wait for signature FIS to arrive (aka register device to host FIS or FIS 34). The polling can be achieved by calling `mvStorageIsDeviceBsyBitOff()`, which reads the ATA Status register and returns MV_TRUE if the BSY bit switched from '1' to '0'.

Note

Steps #1 and #2 above can be automated by calling the `mvStorageDevATASoftResetDevice()` CORE driver function, which initiates a software reset protocol and polls until FIS 34 is received. The problematic issue is that a hard drive may be in its mechanics initialization state, thus it may take a few seconds until it has been completed. Due to this, the higher layer developer is encouraged to have a timer-based polling mechanism that as its first step initiates a software reset protocol, but the polling for reception of FIS 34 can be used with timer based polling methods that releases the CPU for performing other tasks.

3. Execute the IDENTIFY DEVICE ATA command (using the `mvStorageDevATAIdentifyDevice()` CORE driver function).
4. Parse the IDENTIFY DEVICE data buffer and accordingly set the hard drive's parameters such as UDMA speed, write cache, read ahead, etc.

2.2.3.4 Configuring EDMA Mode

Configure each SATA channel's EDMA mode using the information collected from the IDENTIFY DEVICE data buffers (see Section 2.2.3.3 "Initialization of Hard Drive Algorithm").

This can be achieved by calling the `mvSataConfigEdmaMode()` CORE driver function.

2.2.4 Command Queuing, Execution and Completion

After adapter detection and initialization, when the storage devices detection and initialization phases have been completed, the adapter and hard drives connected to it are ready for command queuing and execution.

2.2.4.1 Command Queuing and Execution

The command queuing is performed using the `mvSataQueueCommand()` CORE driver function.

If the command is UDMA read/write then the IAL must provide as input to the `mvSataQueueCommand()` function, a PRD table which is a scatter-gather table (see Section 6. "Core Driver").

The UDMA commands are executed solely by the adapter's EDMA engine, from the point of view of queuing to hardware, data transfer, and completion. The PIO commands are performed by the CORE driver.

When the outstanding commands issued to the CORE driver have mixed PIO and UDMA commands, the CORE driver identifies the command and automatically switches between EDMA enabled mode for UDMA commands execution, and EDMA disabled mode for PIO commands execution.

Note

Higher layers can optionally call the `mvSataNumOfDmaCommands()` CORE driver function, which returns the number of outstanding commands on the specific SATA channel. Using the return value, the higher layers can be signalled as to whether the SATA channel has empty slots for further command queuing.

2.2.4.2 Command Completion

Command completion is done using a callback function called by the CORE driver, which is a higher layers function, to indicate completion of a specific command. The completion can have different status indication— successful completion and erroneous completion.

Usually the command completion scheme is triggered by the adapter's interrupt notifying higher layers of a specific event. The higher layers call the CORE driver `mvSataInterruptServiceRoutine()` function, which interrogates the adapter and response queues, and upon recognition of completion the CORE driver calls a callback function with the statuses.

The callback function is per command and it is defined in the command, when issued to the CORE driver using the `mvSataQueueCommand()` function.

The CORE driver supports three types of command completion schemes. At any time a single scheme is valid.

The higher layers are allowed to change the command completion scheme, but when they do scheme switching, higher layers must make sure that no outstanding commands are queued to the adapter.

Interrupt Driven Driver and Command Completion in ISR

This is the default CORE driver command completion mechanism.

The following scenario is typical for such a scheme:

1. Adapter issues PCI interrupt.
2. Operating system calls higher layers interrupt service routine (this is the ISR routine described in Section 2.2.2 "Hardware Detection, Adapter, and CORE Driver Initialization").
3. The higher layers ISR calls the `mvSataInterruptSerivceRoutine()` CORE driver function.
4. The `mvSataInterruptServiceRoutine()` function interrogates the adapter and response queues, and accordingly calls a callback function for completion.

Interrupt Driven Driver and Command Completion Deferred in Task

This scheme makes it possible to partition the command completion into two steps.

The first step is masking the adapter's interrupt and scheduling a task for interrupt servicing. The second step is the actual interrupt service routine and command completion.

Note

To enable this scheme, the higher layers must call the `mvSataSetInterruptScheme()` function after `mvSataInitAdapter()` has been called.

The following scenario is typical for such scheme:

1. Adapter issues PCI interrupt.
2. Operating system calls higher layers interrupt service routine (this is the ISR routine described in Section 2.2.2 "Hardware Detection, Adapter, and CORE Driver Initialization").
3. The higher layers ISR calls the `mvSataCheckPendingInterrupt()` CORE driver function.
   If the function returns MV_FALSE, then the interrupt is not the adapter's.
   If the return value is MV_TRUE, then higher layers schedules a task in which the actual interrupt handling will be done.
   Note that if the return value is MV_TRUE, then the `mvSataCheckPendingInterrupt()` function has already masked the adapter's interrupts.
4. Higher layers ISR exits.
5. Scheduled task is executed after the higher layers ISR exits.

6. Task calls the `mvSataInterruptServiceRoutine()` CORE driver function.
7. The `mvSataInterruptServiceRoutine()` function interrogates the adapter and response queues and accordingly calls the callback function for completion.
8. The `mvSataInterruptServiceRoutine()` function unmasks the adapter's interrupts before exiting.

Polling Driven Command Completion

This scheme makes it possible to poll for command completion without using the PCI interrupts as a trigger for command completion.

Notes

- To enable this scheme, the higher layers must call the `mvSataSetInterruptScheme()` function after `mvSataInitAdapter()` has been called.
- When this scheme is enabled, higher layers must make sure that `mvSataUnmaskAdapterInterrupts()` has not been called, or if it has been called previously, then `mvSataMaskAdapterInterrupts()` is called before enabling this scheme.

The following scenario is typical for such a scheme:

1. Higher layers queue command(s) using the `mvSataQueueCommand()` CORE driver function.
2. Higher layers call the `mvSataInterruptServiceRoutine()` CORE driver function.
3. The `mvSataInterruptServiceRoutine()` function interrogates the adapter and response queues and accordingly calls the callback function for completion.

Note

Higher layers must not queue new commands in the context of completion callback functions. Higher layers must wait until mvSataInterruptServiceRoutine() exits.Afterwards it is permissible to queue new commands.

2.2.5 Error Handling

When using the CORE driver, the following hardware and software errors may occur:

- PCI bus error
- SATA bus errors
- Hard drives errors
- Command timeout
- Software errors

The following section details how the CORE driver handles such errors, using its API.

2.2.5.1 PCI Bus Error

When a PCI bus error occurs and it is detected by the adapter, a PCI interrupt is generated (depending on the `pciSerrMask` and `pciInterruptMask` settings defined in Section 2.2.2 "Hardware Detection, Adapter, and CORE Driver Initialization"). Higher layers must call the `mvSataInterruptServiceRoutine()` function as part of the interrupt service. Upon PCI bus error the `mvSataInterruptServiceRoutine()` function calls the `mvSataEventNotify()` callback function with corresponding parameters, indicating that a PCI bus error was detected.

It is recommended that upon detection of a PCI bus error, higher layers abort all outstanding commands, re-initialize the adapter and its storage devices, and then retry the aborted commands.

2.2.5.2 SATA Bus Error

Depending on the adapter, a SATA bus error can be identified using different methods. For example, for a 88SX50XX device, if an UDMA command was completed with the ERR bit equal to '1' in ATA status, and the ATA ERROR register equals 0x0C or 0x14, then this means that a SATA bus error occurred in the middle of execution of a command. (The command is completed via the completion callback function of the specific failed command).

The higher layers can also periodically poll the SError registers on all the adapter's SATA channels, to identify serial ATA bus errors.

It is recommended that upon detection of a serial ATA bus error, all outstanding commands be aborted and retried.

2.2.5.3 Hard Drive Errors

Upon completion of a failed command due to a hard drive error (for example UNC error), the CORE driver calls the callback function with error indication.

Depending on the type of failure, higher layers must decide which actions should be done next.

2.2.5.4 Command Timeout

When higher layers issue a command to the CORE driver, it is recommended to allocate a timeout period for each specific command.

When a command's timer expires, it is recommended that the higher layers perform the following for the specific SATA channel on which the command timed out:

1. Call the `mvSataDisableChannelDma()` function.
   This disables the queuing mechanism.
2. Call `mvSataFlushDmaQueue()`.
   This triggers the CORE driver to empty its queue. When each entry is emptied, the CORE driver calls the relevant callback function with abort indication.
3. Call `mvSataChannelHardReset()`.
   This resets the adapter's specific serial ATA bridge and re-issues an OOB sequence.
4. Trigger the Storage Devices Detection and Initialization algorithm for re-initialization of the SATA channel. (See Section 2.2.3 "Storage Devices Detection and Initialization" .)
5. After re-initialization has been completed, retry the abort commands.

2.2.5.5 Software Errors

The CORE driver has many sanity checks with regards to the parameters that are passed to the CORE driver functions. It is recommended that in the initial stages of system integration, all CORE driver logging messages be enabled. At more advanced stages the CORE driver provides the ability to filter only the error messages and disable all logging messages in the production driver, since they increase the CORE driver's footprint and decreases its performance.

2.3 System Integration Using CORE Driver, SCSI to ATA Translation Layer, and Common IAL Layers

This system integration method is typical for systems that have SCSI subsystems that initiate SCSI commands.

The CORE, SAL, and Common IAL layers can be integrated with the user's IAL, thus providing an interface for executing SCSI commands as if the SATA hard drives were SCSI targets.

This is done by different functionality provided by the different layers:

- SAL provides the functionality of translating SCSI commands into ATA commands and an interface for queuing to the CORE driver.
- CORE driver provides hardware access and ATA command queuing.
- Common IAL provides the functionality for initializing SATA hard drives.

This type of system integration has the following components and tasks that must be fulfilled:-

- Coding of a system-dependent header file (mvOs.h) that enables the CORE driver accessing system resources (described in Section 6.6 "System-Dependent Header File (mvOs.h)" on page 75).
- Hardware detection; initialization of adapter, CORE, SAL, and Common IAL drivers.
- Command queuing, execution, and completion.
- Error handling.

2.3.1 System-Dependent Header File (mvOs.h)

See Section 2.2.1 "System-Dependent Header File (mvOs.h)"

2.3.2 Hardware Detection; Initialization of Adapter, CORE, SAL, and Common IAL Drivers

See Section 2.2.2 "Hardware Detection, Adapter, and CORE Driver Initialization" up to and including calling the `mvSataInitAdapter()` CORE driver function.

Afterwards, higher layers must perform the following steps:

1. Call the `mvSataScsiInitAdapterExt()` SAL function to initialize the SAL layer.
2. Set up the adapter's interrupt line to trigger a higher layers interrupt service routine wrapper upon interrupt generation.
3. Call the `mvAdapterStartInitialization()` Common IAL function, which starts the initialization process of storage devices connected to the adapter's SATA channel.
4. Set up the timer function that is called every 0.5 seconds (or any other configurable variable in Common IAL). The timer function must call the `mvIALTimerCallback()` Common IAL function.

2.3.3 Command Queuing, Execution, and Completion

This section describes how SCSI commands are queued, executed, and completed.

2.3.3.1 Command Queuing and Execution

When the IAL receives a SCSI command, the IAL checks if it is a SCSI read/write command. If this is the case, then the IAL must build a PRD table for the SCSI command.

If the SCSI command is read/write or any other SCSI command, the IAL calls the `mvExecuteScsiCommand()` function, which handles all translation from SCSI commands to ATA commands, and queuing to the CORE driver.

2.3.3.2 Command Completion

When using the SAL, there are several types of command completion:

- Immediate command completion. In the `mvExecuteScsiCommand()` function, the SAL calls the callback function. This is indicated by the return value MV_SCSI_COMMAND_STATUS_COMPLETED from `mvExecuteScsiCommand()`.
- Command queued to CORE driver. This is indicated by the return value MV_SCSI_COMMAND_STATUS_QUEUED from `mvExecuteScsiCommand()`.
- Failure of command execution. Usually this is because the SAL does not support the command being queued. This is indicated by the value MV_SCSI_COMMAND_STATUS_FAILED from `mvExecuteScsiCommand()`.
- Queuing in initialization stages. This is usually command queuing due to SATA drives being initialized. When SATA drives initialization has been completed for a specific SATA channel, all SCSI commands that were previously queued with the MV_SCSI_COMMAND_STATUS_QUEUED_BY_IAL return value will be aborted and re-queued by the SCSI subsystem.

The command completion status returned by the SAL is a SCSI-like status. Users must map the completion statuses to their specific SCSI subsystem completion statuses.

2.3.4 Error Handling

When using the CORE driver, SAL, and Common IAL, the following hardware and software errors may occur.

- PCU bus error
- SATA bus error or hard drives errors
- Command timeout
- Software errors

2.3.4.1 PCI Bus Error

See Section 2.2.5.1 "PCI Bus Error"

2.3.4.2 SATA Bus Error or Hard Drives Errors

These errors are reported to the SAL.

When such errors occur, the SAL translates the error codes to SCSI-like error codes.

Depending on the type of the failure, higher layers must decide on the next actions to be done.

2.3.4.3 Command Timeout

See Section 2.3.4.3 "Command Timeout" . Instead of triggering the Storage Devices Detection and Initialization algorithm, (Section 2.2.3) call the `mvRestartChannel()` Common IAL function, which performs all the storage devices re-initialization.

2.3.4.4 Software Errors

See Section 2.2.5.5 "Software Errors" .

2.4 System Integration by Example

This section provides an example of system integration using the methods described in Section 2.3 "System Integration Using CORE Driver, SCSI to ATA Translation Layer, and Common IAL Layers" .

The example is the Marvell Windows SCSI mini-port driver for the 88SX50xx /88SX60x1 adapters.

2.4.1 Hardware Detection

The Windows kernel performs hardware detection in the following manner:

1. Windows calls `DriverEntry()` function.
2. DriverEntry enables auto-flush mechanism (see Section 6. "Core Driver" for further information about auto-flush).
3. DriverEntry initializes a template hwInitializationData data structure that contains all function pointers for hardware initialization, command execution, and interrupt service routine.
4. DriverEntry calls the `ScsiPortInitialize()` SCSI Port function, each time defining a new PCI device ID from the 88SX50xx /88SX60x1 adapters device list. According to this method, DriverEntry requests that the SCSI Port scan the PCI buses for adapters that have the relevant vendor ID and device ID, and accordingly calls `mvFindAdapter()`.

2.4.2 Hardware Initialization

For each adapter found by the SCSI port driver (as requested by DriverEntry using the `ScsiPortInitialize()` function), the SCSI port driver calls the `mvFindAdapter()` function, which performs the following:

1. Initializes the HwDeviceExtension parameters (see Windows DDK).
2. Gets PCI BAR 0 mapping through ScsiPortValidateRange and ScsiPortGetDeviceBase. Note that the Windows kernel has already enabled memory and I/O access to the adapter and has also enabled the adapter's capability to be a PCI master.
3. Reads adapter's PCI device ID and revision ID.
4. Allocates request and response queues.
5. Initializes the MV_SATA_ADAPTER data structure (which is part of HwDeviceExtension as previously requested by DriverEntry).
6. Calls the `mvSataInitAdapter()` function.
7. Calls the `mvSataScsiInitAdapterExt()` function with a pointer to MV_SAL_ADAPTER_EXTENSION, which is also part of HwDeviceExtension, as previously requested by DriverEntry.

2.4.3 Storage Devices Initialization

The SCSI port calls the `mvHwInitialize()` function, which triggers storage devices initialization by calling the `mvAdapterStartInitialization()` Common IAL function.

2.4.4 Command Queuing and Execution

The SCSI port calls `mvStartIO()` for executing SCSI commands (and other tasks).

If the request was I/O-Control, the IAL handles this request.

If the request was executing SCSI command, the IAL checks if the command is a Read or Write SCSI command, in which case the IAL must build a PRD table.

Afterwards, the IAL issues a command to the SAL via the `mvExecuteScsiCommand()` SAL function.

The IAL checks the return value of `mvExecuteScsiCommand()` and accordingly decides wether to call `ScsiPortNotification()` with NextRequest or NextLuRequest.

2.4.5 Interrupt Servicing and Command Completion

Upon PCI interrupt from the adapter (or from another adapter sharing the same PCI IRQ), the SCSI port calls the `mvInterrupt()` function, and `mvInterrupt()` calls `mvSataInterruptServiceRoutine()` for interrupt processing.

Within `mvSataInterruptServiceRoutine()` the command completion callback is called for completing the SCSI commands.

After `mvSataInterruptServiceRoutine()` has been completed and has returned MV_TRUE (indicating that the interrupt was generated by the adapter), `mvInterrupt()` calls the `mvSataScsiPostIntService()` SAL function.

For every SCSI command completed, the `IALCompletion()` IAL function is called. (This is defined as a callback function for every SCSI command issued to the SAL.

The `IALCompletion()` function maps the corresponding SAL completion status to SCSI port driver-specific status codes.

2.4.6 Bus Reset Upon Timeout

Upon timeout caused by a non-completed SCSI command (previously issued by the SCSI port driver) the `mvResetBus()` IAL function is called.

The function performs the following:

1. Calls the `mvSataDisableChannelDma()` CORE driver function to disable queuing.
2. Calls the `mvSataFlushDmaQueue()` CORE driver function. As a result, all callback functions for the outstanding commands are called with abort indication.
3. Calls the `mvSataChannelHardReset()` CORE driver function, to reset the SATA bridge and restart an OOB sequence.
4. Calls the `mvRestartChannel()` Common IAL function, to restart storage device re-initialization.

2.5 Miscellaneous Issues

2.5.1 Hotplug on SATA Channels

Upon hotplug event on SATA channel (either connected directly to the adapter or indirectly, through the port multiplier) the user-implemented `mvSataEventNotify()` function is called with the corresponding event.

The event indication can be one of the following:

- SATA channel connect event on 88SX50xx /88SX60x1 adapter
(indicated by MV_SATA_CABLE_EVENT_CONNECT).
- SATA channel disconnect event on 88SX50xx /88SX60x1 adapter
(indicated by MV_SATA_CABLE_EVENT_DISCONNECT)
- SATA channel connect or disconnect on port multiplier's device side SATA channels 88SX50xx /88SX60x1
(indicated by MV_SATA_CABLE_EVENT_PM_HOT_PLUG).

The hotplug event on the SATA channel connected directly to the adapter is easier to handle than the indirectly connected channel (through port multiplier) for the following reasons:

- When the port multiplier sends SDB FIS with the 'N' bit indicating a hotplug event on the port multiplier, there is no indication whether a device SATA channel was connected or disconnected, or on which device side

SATA channel it occurred. On the other hand, when the hard drive is directly connected to the adapter, the previous events are reported by the adapter.

- When the hard drive is disconnected from the port multiplier while it is in the middle of a transaction (either due to PIO command or UDMA command), the command will not be completed. If the hard drive is directly connected and a disconnect event occurs, then the software driver can simply abort the transactions.

Due to the above reasons, it is recommended that upon receipt of a hotplug event on the port multiplier's device port, the software drive immediately aborts all outstanding commands and starts storage device re-initialization on the specific SATA channel.

Thus at the end of the re-initialization process, the software can determine the reason for the hot plug event and on which device channel it occurred.

2.5.2 Logger Module for Debug Messages Logging

The Logger module is a generic debug messages logging mechanism. It is used by the CORE driver, SAL, and Common IAL independently of one another.

It is recommended that the IAL also register its debug messages in this logging mechanism.

The Logger module produces output messages for debugging, monitoring, and tracking the activity of the SATA adapter driver modules. It uses module identifiers with the required logging level to monitor driver activity. The logging level for each module determines the type of messages being printed for the module. Every driver module may be registered independently to the logger, with the desired logging level.

When the logging filter associated with the module matches the level of the current log message, the logger keeps the module name and log filter. It only prints the messages from registered modules.

The message output format is "<Module name> (<Debug Level>) <Message body>", i.e.

```
"Core Driver (DEBUG) Issue SRST command"
```

Note

The Logger module is implemented as part of the CORE driver. See Section 6. "Core Driver" on page 42 for further information on the API of the Logger module.

2.5.3 Channel-to-Channel Communication (aka Target Mode)

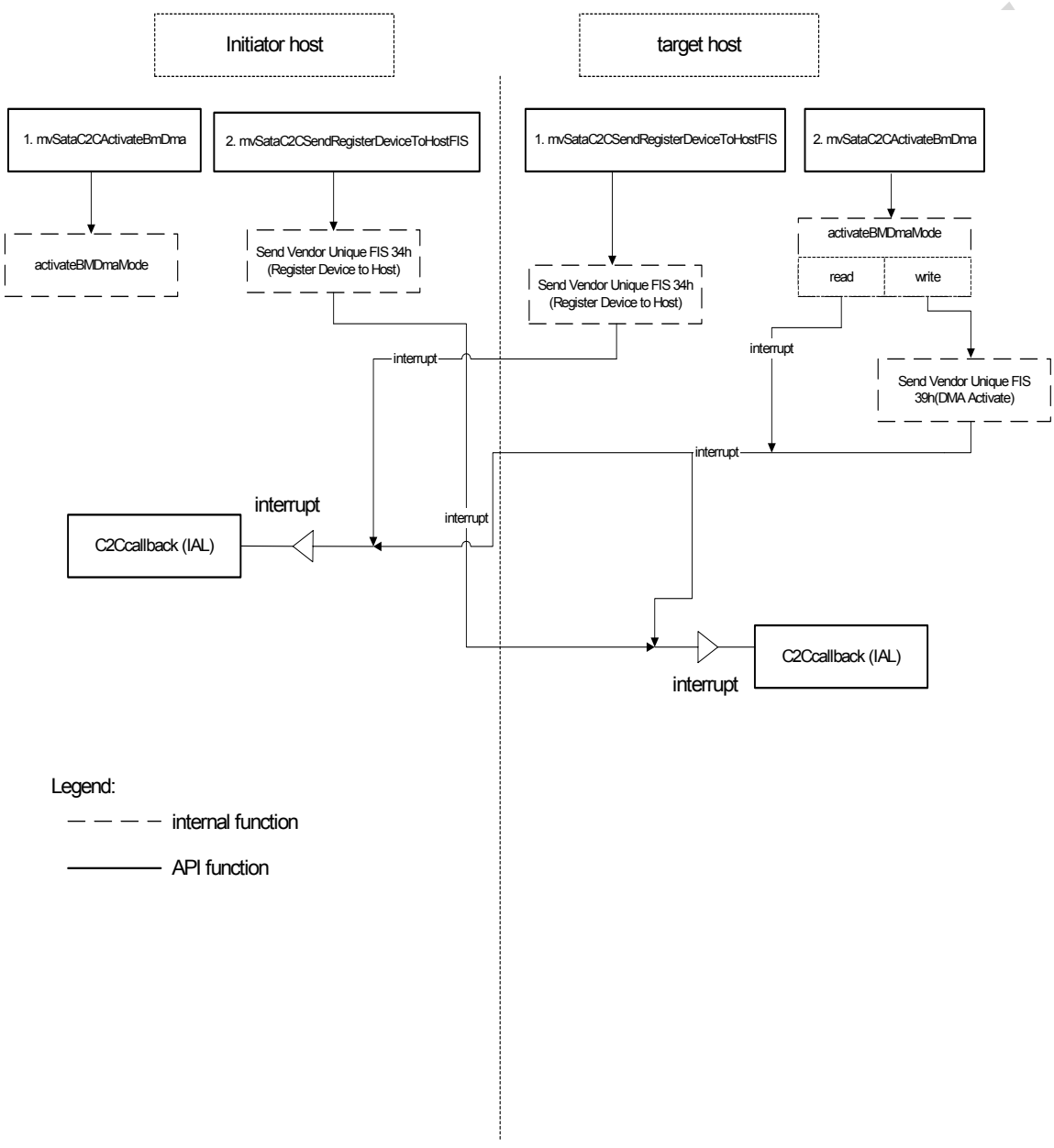
The 88SX60x1 adapter supports a channel-to-channel communication feature (aka Target mode).

In this mode, the Serial-ATA II ports are used for communication between two 88SX60x1 adapters. The communication channels are not symmetric—one side is configured as an initiator while the other side is configured as a target.

The communication is carried out based on sending either a message or a block of data (via DMA). An IAL callback function called by the CORE driver as part of the its interrupt service routine indicates that a message was received and/or the SATA channel's DMA has been completed.

sFigure 2 describes the channel-to-channel communication driver support in the CORE driver.

Figure 2: Channel-to-Channel Communication Driver Support in the Core Driver



2.5.3.1 Channel-to-Channel Communication: Initialization

The channel-to-channel feature is initialized by calling the `mvSataC2CInit()` CORE driver function.

The IAL calls the function with MV_SATA_C2C_MODE_INITIATOR for the adapter with the initiator SATA channel and MV_SATA_C2C_MODE_TARGET for the adapter with the target SATA channel.

When calling the `mvSataC2CInit()` CORE driver function, the call-back function must be valid for both the initiator and the target.

For disabling the channel-to-channel communication feature the IAL can call the `mvSataC2CStop()` CORE driver function.

2.5.3.2 Channel-to-Channel Communication: Sending a Message

Both initiator and target can send a message with a size of 10 bytes.

The hardware sends this messages using FIS 34 SATA FIS.

For sending a message the IAL calls the `mvSataC2CSendRegisterDeviceToHostFIS()` Core driver function.

The receiving SATA channel issues an interrupt and the CORE driver calls the defined call-back function as part of the interrupt handling.

2.5.3.3 Channel-to-Channel Communication: Transferring Blocks

The IAL can initiate a block transfer using DMA transfer SATA protocol between the initiator and target SATA channels. The DMA transfer direction can be either from/to the target SATA channel, but the DMA initiation can be triggered only by the initiator SATA channel.

The algorithm is as follows:

1. Initiator's IAL calls the`mvSataC2CActivateBmDma()` CORE driver function for the initiator SATA channel. This initializes the initiator's DMA and waits for DMA Setup FIS from target SATA channel.
2. Initiator's IAL calls the `mvSataC2CSendRegisterDeviceToHostFIS()` Core driver function to send a message to the target channel. The content of the message depends on the IAL's implementation. It is recommended that message be unique, so that the driver handling the target SATA channel can recognize it and accordingly set the target's DMA.
3. Software driver that handles the target SATA channel (target's IAL) receives a unique message (triggered by the CORE driver calling the defined call-back function as part of the interrupt handling).
4. Target's IAL calls the `mvSataC2CActivateBmDma()` CORE driver function for the target SATA channel. The CORE driver sets up the target's SATA channel DMA and sends DMA ACTIVATE FIS to the initiator.
5. Initiator and target SATA channels perform DMA transfer. When the DMA transfer has been completed, both the initiator's and target's IALs receive a call-back function call from the CORE driver indicating the completion.
6. Both the initiator's and target's IALs call the `mvSataC2CResetBmDma()` CORE driver after receiving the DMA transfer completion indication.

2.5.3.4 Channel-to-Channel Communication: Error Handling

Message Error Handling

When sending a message via the `mvSataC2CSendRegisterDeviceToHostFIS()` Core driver function, if any error occurs, then the function returns MV_FALSE.

The errors can be either parameter errors, initialization errors, or FIS transmission errors.

In the case of a FIS transmission error, the IAL can retry the FIS transmission by calling the `mvSataC2CSendRegisterDeviceToHostFIS()` CORE driver function.

Block Transfer Error Handling

Upon recognition of a block transfer error, an interrupt is issued for either the initiator or the target. As part of the CORE driver's interrupt service routine, the IALs call-back function is called with the proper error indication.

2.5.4 I/O-Granularity

The interrupt coalescing in I/O Granularity enables command completion interrupts to be coupled into a single interrupt. This feature can be used in RAID applications. In such applications a single RAID transaction is divided into EDMA transactions to multiple drives and the completion interrupt can be generated as a single interrupt for the entire RAID transaction.

Interrupt Coalescing in I/O Granularity Design Highlights

- For every I/O transaction, the related I/O transaction counter is updated with the number of SATA commands related to this I/O transaction.
- The 88SX60x1 adapter issues a maskable interrupt, when the number of SATA commands executed with a specific I/O transaction number equals the number of SATA commands related to that specific I/O transaction number.
- The I/O Granularity driver support is capable of switching between two modes of operations—with and without interrupt coalescing in I/O granularity per SATA adapter.
- When interrupt coalescing in I/O granularity is enabled for the controller, all EDMA transactions for this adapter are executed with an interrupt coalescing I/O granularity interrupt scheme.

2.5.4.1 Enabling I/O-Granularity CORE Driver Support

To enable I/O-Granularity CORE driver support, the IAL must call the `mvSataEnableIoGranularity()` CORE driver function after `mvSataInitAdapter()` has been called.

2.5.4.2 I/O-Granularity Command Queuing

The following steps describe the modifications needed to be done for queuing a command when the I/O-Granularity feature is enabled.

1. For the first command in the chain of commands to be queued with I/O-Granularity, the IAL must set the iogCommandType to MV_IOG_COMMAND_TYPE_FIRST and then set the total number of commands in the `transCount` field.
2. Call the `mvSataQueueCommand()` function. Upon exit the CORE driver sets `iogCurrentTransId`. This is a unique code for the chain of command.
3. IAL saves the iogCurrentTransId returned by CORE driver. This field must be used for the next commands in the chain.
4. For the other commands in the chain of commands, the IAL must set the `iogCommandType` to MV_IOG_COMMAND_TYPE_NEXT and then set the transId to the saved value that was previously returned by CORE driver.

2.5.4.3 I/O-Granularity Command Completion

The command completion is the same as that described in.

2.5.4.4 I/O-Granularity Error Handling

When the I/O-Granularity feature is enabled, error handling is the same as that described in Section 2.2.5 "Error Handling" with the following addition:

When any error occurs, the I/O-Granularity feature is automatically disabled by the CORE driver, to enable the IAL to perform error handling and recovery.

When the IAL has completed error handling and recovery, it must re-enable I/O-Granularity feature support by calling the `mvSataEnableIoGranularity()` CORE driver function.

Note

See Section 6. "Core Driver" for further information about the I/O-Granularity functions API.

2.5.5 Restrictions when Using the CORE Driver API

When using the CORE driver API, the following restrictions apply:

- When allocating a request queue, the PCI address should be 1 KByte aligned.
- When allocating a response queue the PCI address should be 256 bytes aligned.
- The request queue and response queue should be cache coherent or in non-cacheable memory regions.
- When the IAL builds a PRD table, each entry must not cross the 4 GByte addressing boundary (for further details, see the 88SX50xx /88SX60x1 datasheet).
- When the IAL calls `mvSataShutdownAdapter()` for deactivating a specific SATA channel, it must make sure that no other task is using the CORE driver API.
- Depending on operating system implementation, the IAL must gracefully remove (or delete) the semaphore from the relevant ***MV_SATA_CHANNEL*** data structure after calling `mvSataRemoveChannel()`.

Section 3. Linux Intermediate Application Layer

3.1 Introduction

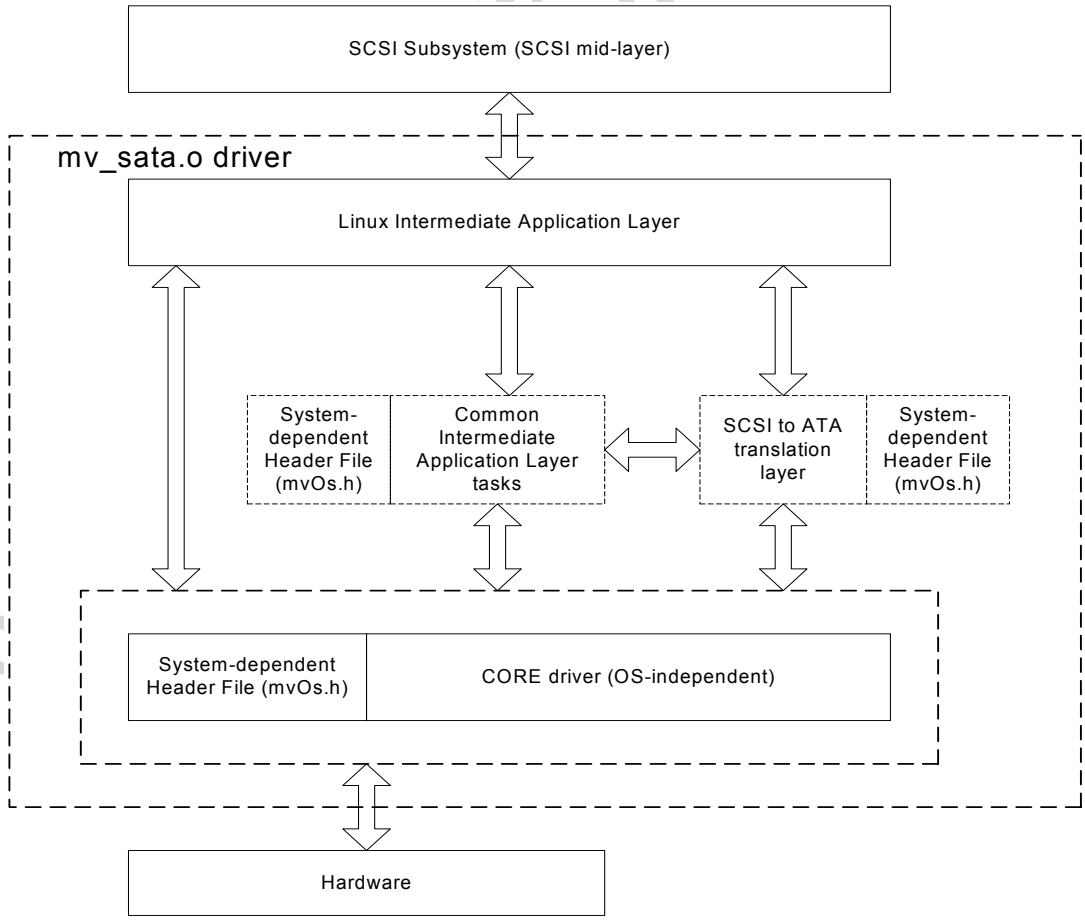
The 88SX50xx /88SX60x1 Linux Intermediate Application Layer (IAL) is a software layer positioned under the Linux SCSI subsystem and above the CORE driver, SAL, and Common IAL.

The Linux IAL implements a Linux SCSI host template (Linux SCSI sub-system low level driver interface) that receives SCSI-3 commands and forwards them to the SAL for SCSI command translation to ATA, and later for queuing to hardware through the CORE driver API.

The Linux IAL consists of the following parts:

- Linux IAL SCSI Host Template Driver
- Linux IAL Extension Library

Figure 3: Linux IAL Driver Architecture



3.1.1 Linux IAL SCSI Host Template Driver

The Linux IAL Host Template driver is the part of the Linux IAL that enables connectivity to the Linux SCSI subsystem. It presents the 88SX50xx /88SX60x1 as a SCSI host adapter, by implementing a SCSI host template.

The Linux IAL Host Template driver implements the following functionality:

- Registers the SCSI host adapter to the Linux SCSI subsystem using a Linux SCSI host template data structure.
- Triggers the 88SX50xx /88SX60x1 adapter initialization sequence.
- Triggers a storage devices initialization process using the Common IAL API. These storage devices are connected to the 88SX50xx /88SX60x1 SATA channels.

3.1.2 Linux IAL Extension Library

The Linux IAL Extension Library provides the Linux IAL SCSI Host Template driver with the ability to perform a set of tasks.

The Extension Library is not implemented from within the SCSI Host Template driver. This is because the Extension Library is more 88SX50xx /88SX60x1-hardware-oriented than the SCSI Host Template Driver, which makes it easier for the user to differentiate between Linux-oriented source code and hardware-oriented source code.

The Linux IAL Extension Library implements the following functionality:

- PRD table generation from SCSI scatter-gather buffers list.
- Sanity checking of vital 88SX50xx /88SX60x1 configuration.
- SCSI commands completion notification, based on call-back function from the SCSI subsystem.
- Triggering of Common IAL storage device initialization sequence upon hot-plug event.
- CORE driver Interrupt Service Routine (ISR) Wrapper.

3.2 Linux IAL SMART (Self-Monitoring, Analysis, and Reporting Technology) Support

The Linux IAL supports SMART commands. The commands are standard SMART commands, but the interface is a Marvell proprietary interface through the SCSI_IOCTL_SEND_COMMAND interface.

The structure of the SCSI command delivered via SCSI_IOCTL_SEND_COMMAND is a 6-byte command that is followed by the buffer containing the ATA register values. The driver uses the same buffer for input and output, thus the minimum buffer allocated by application must be 520 bytes (8 bytes for ATA registers + 512 bytes for the whole sector returned by some of SMART commands).

For an explanation of ATA register values, see the ATA/ATAPI-6 specification.

Marvell has ported a general Linux utility called SMARTMONTOOLS. This utility generally interfaces to ATA hard drives via /dev/hdX block devices.

To communicate with the driver, the ported utility uses the SCSI_IOCTL_SEND_COMMAND IO control function with the vendor-specific command code 0xC as a transport for the SMART commands.

For an explanation of the SMARTMONTOOLS package, see the Readme file and the SMARTMONTOOLS user manual.

The following commands are supported by the Linux IAL as part of the support for SMART:

- IDENTIFY - ATA IDENTIFY (not a SMART command)
- ENABLE SMART
- DISABLE SMART
- ENABLE DISABLE AUTOSAVE
- EXECUTE OFFLINE DIAGS
- RETURN STATUS
- READ SMART THRESHOLDS
- ENABLE DISABLE AUTO OFFLINE
- READ SMART LOG

Table 1: SMART Command Input Buffer

Buffer Offset	Value	Description
0x0	0xC	Vendor-specific SCSI command
0x1	0	Reserved
0x2	0	Reserved
0x3	0	Reserved
0x4	6	Command header length
0x5	0	Reserved
0x6	0xEC for the INDENTIFY 0xB0 for the SMART	ATA command register input value
0x7	Depends on the command	ATA sector number register input value
0x8	Depends on the command	ATA features register input value
0x9	Depends on the command	ATA sector count register input value
0xA	Depends on the command	ATA LBA Mid register input value
0xB	Depends on the command	ATA LBA High input value
0xC	Depends on the command	ATA device register input value
0xD	Depends on the command	ATA error register input value

3.3 Building and Running the Project

3.3.1 Requirements

The requirements are:

- Linux machine with kernel 2.4 series
- 88SX50xx /88SX60x1 software package
- Native GNU toolchain compilers (gcc, ld etc.)
- Kernel header files installed on /**usr/src/linux**

3.3.2 Building and Running the Project

To build and run the project:

1. Log in as root.
2. Change the current directory to **LinuxIAL**.
3. Execute a **make** command on the shell.
4. Add a SCSI subsystem to the running kernel by executing **/sbin/modprobe scsi_mod**.
5. Run the kernel module by executing **/sbin/insmod mv_sata.o**.
   This detects storage devices connected to the SATA adapter and presents them as SCSI storage devices to the Linux kernel.

To build a project with three different levels of log messaging:

1. Log in as root.
2. Change the current directory to **LinuxIAL**.
3. Execute a **sh build.sh** command on the shell.
4. The script creates target directory **build/Linux** in driver root.
   The target directory structure is:
   build/Linux/DebugError/mv_sata.o - Module prints log messages on error.

   build/Linux/DebugFull/mv_sata.o - Module prints all log messages.

   build/Linux/Free/lmv_sata.o - Module prints no messages.
5. Add a SCSI subsystem to the running kernel by executing **/sbin/modprobe scsi_mod**.
6. Change the current directory to the desired target directory, e.g., **cd ./../build/Linux/DebugFull**.
7. Run the kernel module by executing **/sbin/insmod mv_sata.o**.
   This detects storage devices connected to the SATA adapter and presents them as SCSI storage devices to the Linux kernel.

Notes

- If the kernel header files are not located in the **/usr/src/linux** directory, edit the Makefile in the LinuxIAL directory and change the value of the KERN_HEADERS parameter to the desired directory.
- If you need to cross-compile the project (and not compile it with native tools), edit the Makefile and modify the **CROSS_COMPILE** parameter so it has the prefix of the desired cross compiler toolchain (e.g., for PowerPC, CROSS_COMPILE is assigned the value **powerpc-linux-**). Also, modify the **CFLAGS** parameter to the correct values. (See the example in the Makefile.)
- To stop and remove the kernel module from the Linux kernel, execute **rmmod linuxIAL**.

3.3.2.1 Building the Project for Linux RedHat

This section describes how to build the project that makes it possible to install and boot Linux RedHat on an 88SX50xx /88SX60x1 adapter.

1. Log in as root.
2. Download the Redhat kernels to a directory, e.g., **/usr/src/kernels/**. For RedHat 8 you must have a directory **/usr/src/kernels/2.4.18-14** and for RedHat 9 you must have a directory **/usr/src/kernels/2.4.20-8**.
3. Change the current directory to **RedHat** (under LinuxIAL).
4. To make sure the files are in Unix format, run the following command:
   > dos2unix gen_module.sh files/*
5. Execute **sh gen_module.sh /usr/src/kernels**. This generates files in the Files directory.
6. Copy these files to a diskette.
7. Start the RedHat installation on the required computer which has the 88SX50xx /88SX60x1 adapter(s).
8. During installation, select Expert mode and follow the installation instructions.
9. When a driver disk is requested, insert the diskette in Drive A and continue with the installation.

Notes

- The gen_modules.sh script scans all directories under **/usr/src/kernels** and according to the directory names it generates four drivers per directory name. The four drivers are drivers for boot, single CPU, SMP, and bigmem.
- Due to the fact that gen_modules.sh scans directories under **/usr/src/kernels**, it needs the exact version of the kernel for which the drivers are being built. For example, for RedHat 9 kernel, the directory name must be **2.4.20-8**, which is the exact kernel version RedHat 9 installation is shipped with.

3.3.2.2 System Monitoring and Driver/proc Extension

After installing the Linux kernel module mv_sata.o, a new directory called 'mvSata' is added to the **/proc/scsi/** directory.

This directory has one or more files in it. Each file's name is a number (number of files is equal to number of 88SX50xx /88SX60x1 adapters installed in the system). Each file indicates a sequence number of a single 88SX50xx /88SX60x1 adapter.

For example if there is a single 88SX50xx /88SX60x1 adapter and there are no other SCSI adapters in the system, a single file named **0** will be found, and its path will be **/proc/scsi/mvSata/0**.

Two operations can be done on each of the /proc extension files—reading from them and writing to them.

3.3.2.3 Reading from the /proc Extension files

When reading from a /proc extension file (for example by executing **cat /proc/scsi/mvSata/0**) the output is as follows:

```
Version_1_0

TimeStamp:
4069645 512


Number of interrupts generated by the adapter is:
130097
```

```
Adapter     Channel   ID    LUN   TO        TSA    TS          Vendor       QD       LBA48
0           0         0     0     2185609   96433  9110766     IC35L040AV   0        0
0           1         0     0     988860    37376  3262702     IC35L040AV   1        0
0           2         0     0     4         4      8           IC35L040AV   1        0
0           3         0     0     4         4      8           IC35L040AV   1        0
0           4         0     0     4         4      8           IC35L040AV   1        0
0           5         0     0     4         4      8           IC35L040AV   1        0
0           6         0     0     4         4      8           IC35L040AV   1        0
0           7         0     0     4         4      8           IC35L040AV   1        0

TO            - Total Outstanding commands accumulated
TSA           - Total number of IOs accumulated
TS            - Total number of sectors transferred (both read/write)
QD            - Queued DMA feature set enabled
LBA48         - Large Block Address 48 feature set enabled
Can't Remove - If '1' the drive can't be removed
```

The meaning of the output is:

- 'Version_1_0' is the output version of the /proc extension file.
- TimeStamp is the timestamp of the system. The first number is the tick count of the operating system and the second is the number of ticks the operating system counts per second (fixed number).
- Number of interrupts produced by the adapter: Note that even if the same interrupt line on the PCI is shared with other adapters, this number will still count the real number of interrupts generated by the adapter.
- Description of the channels attached to the adapter:

Adapter, Channel, ID, LUN	SCSI ID
TO	Number of outstanding commands accumulated.
TSA	Total number of IO operations.
TS	Total number of sectors transferred through the specific SATA channel.
Vendor	Model number of the storage device connected to the specific SATA channel.
QD	Indication that the storage device has the queued DMA feature set enabled (indicated by value 1).
LBA48	Indication that the storage device has the 48-bit LBA addressing feature set enabled (indicated by value 1).

3.3.2.4 Writing to the /proc Extension Files

It is possible to do three different things while writing to the /proc extension files:

- Change interrupt coalescing parameters: Executing a write to the /proc extension file (e.g., **echo "int_coal 1 30 10000" > /proc/scsi/mvSata/0**) changes the values of the interrupt coalescing parameters of the specific SATA unit (each quad SATA unit contains 4 SATA channels). The format is **int_coal x y z,** where **x** is the

SATA unit (0 or 1), **y** is the number of completed commands needed to generate an interrupt, and **z** is the timeout in 150 MHz ticks, where the adapter starts counting after the first command is completed.

- Shut down a SATA physical interface: Execute **echo "sata_phy_shutdown x" > /proc/scsi/mvSata/0** to shut down a SATA PHY **x**.
- Power up a SATA physical interface: Execute **echo "sata_phy_powerup x" > /proc/scsi/mvSata/0** to power up a SATA PHY **x**.

Notes

- All values indicated by **x**, **y**, and **z** above should be in decimal format.
- After power-up, the SATA physical interfaces are all powered up.

3.3.3 Hot-Swapping Storage Devices

When adding or removing a storage device from an operating system, when the 88SX50xx /88SX60x1 adapters driver is already up and running, the SCSI subsystem must be informed of the changes made. This is done by executing Add or Remove requests to the SCSI subsystem.

3.3.3.1 Adding a Storage Device

To add a storage device to a specific SATA channel on a specific adapter (e.g., channel **y** on adapter **x** or device **z** on channel **y** on adapter **x** in the case of a port multiplier):

1. Poll on the /proc extension file relevant to adapter **x** and wait until channel **y** is valid (and the variable **z** if the hard drive is connected to a port multiplier)
2. Execute ’echo "scsi add-single-device **x y z** 0" > /proc/scsi/scsi’.

Note

If the hard drive is connected directly to the adapter’s SATA channel and not through port multiplier, then **z** parameter equals 0

3.3.3.2 Removing a Storage Device

To remove a storage device that is connected on SATA channel **y** on adapter **x** (and **z** if the hard drive is connected through a port multiplier):

1. Execute **’**echo "scsi remove-single-device **x y z** 0" > /proc/scsi/scsi**’**.
2. Remove the storage device connected to SATA channel **x** on a adapter **y**.

Note

If the hard drive is connected directly to the adapter’s SATA channel and not through the port multiplier, then **z** parameter equals 0

3.4 Linux IAL SCSI Host Template Driver API

SCSI Host Template Functions

`mv_ial_ht_detect`	Emulates SCSI host controller to the Linux SCSI mid-level subsystem. Detects 88SX50xx /88SX60x1 adapters on PCI buses. For the adapter it detects, calls ***mv_ial_lib_init_adapter***.
`mv_ial_ht_proc_info`	Prints into a buffer information with regards to a specific 88SX50xx / 88SX60x1 adapter. This information is intended for the /proc file system.
`mv_ial_ht_queuecommand`	Schedules a translation of SCSI commands to ATA commands and issues execution to a specific 88SX50xx /88SX60x1 adapter using the CORE driver API.
`mv_ial_ht_hl_bus_reset`	Resets a specific SCSI bus. Eventually it is translated to reset a specific serial ATA channel.
`mv_ial_ht_release`	Shuts down and frees all resources for a specific 88SX50xx /88SX60x1 adapter.

3.5 Linux IAL Extension Library

88SX50xx /88SX60x1 Device Initialization

`mv_ial_lib_init_channel`	Initializes a specific 88SX50xx /88SX60x1 SATA channel by setting request/response queues, etc.
`mv_ial_lib_free_channel`	Releases a specific 88SX50xx /88SX60x1 SATA channel.

PRD Table Generation

`mv_ial_lib_prd_init`	Initializes a PRD table pool for a specific 88SX50xx /88SX60x1 adapter.
`mv_ial_lib_prd_destroy`	Destroys a PRD table pool for a specific 88SX50xx /88SX60x1 adapter.
`mv_ial_lib_prd_free`	Frees a PRD table.
`mv_ial_lib_generate_prd`	Generates a PRD table from a Linux SCSI command buffer.

Interrupt Service Routine

`mv_ial_lib_int_handler`	CORE driver ISR wrapper.

Event Notification

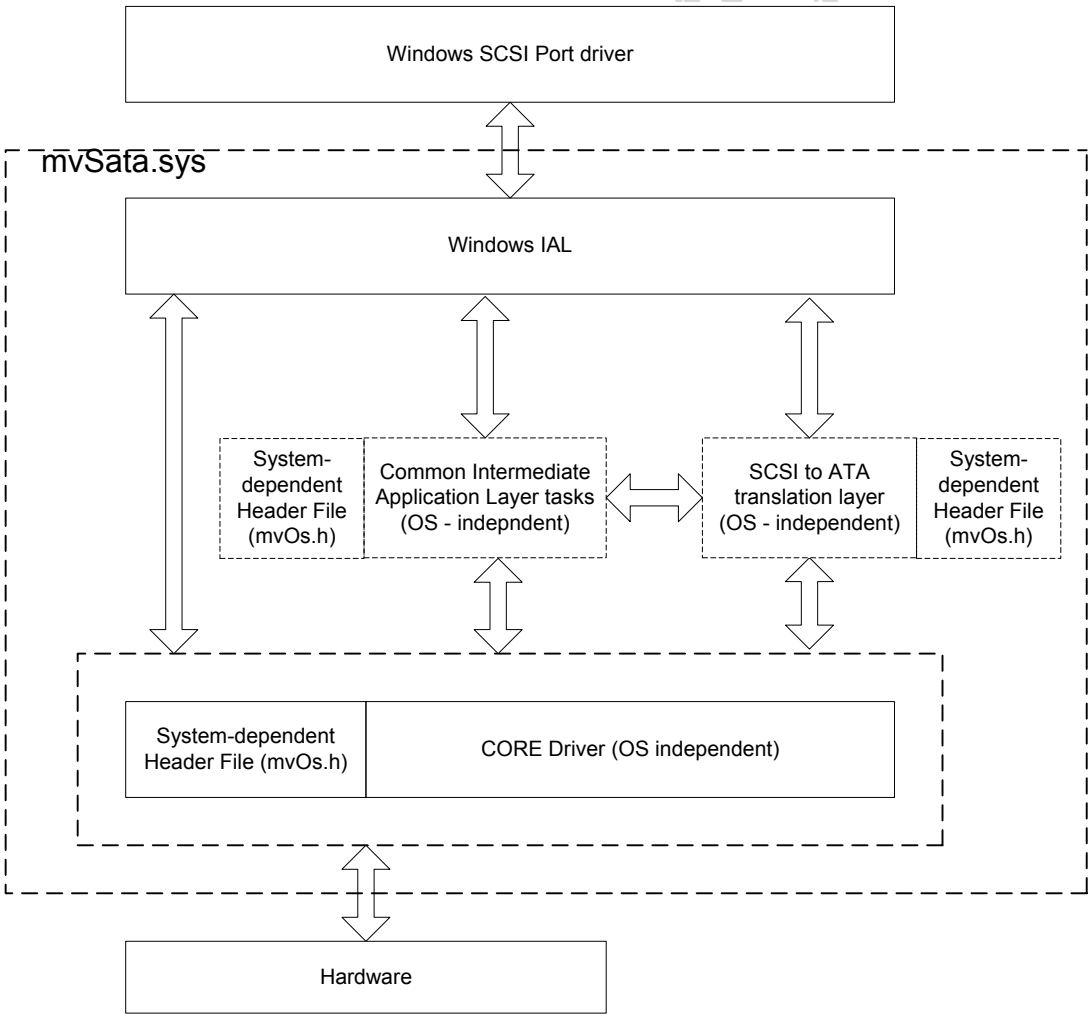
`mv_ial_lib_event_notify`	Event notification upon an event (Interrupt) from the 88SX50xx / 88SX60x1.

Section 4. Windows Intermediate Application Layer

4.1 Introduction

The 88SX50xx /88SX60x1 Windows 2000/XP/2003 Intermediate Application Layer (IAL) is a software layer positioned under the Windows SCSI port driver and above the CORE driver, SAL, and Common IAL.

The Windows 2000/XP/2003 IAL implements a SCSI Miniport driver that receives SCSI-3 commands and forwards them to SCSI to ATA translation layer for further queuing to hardware via CORE driver API.



4.1.1 Windows IAL SCSI Miniport Driver Functionality

The Windows IAL SCSI miniport driver implements the following functionality:

- Represents the 88SX50xx /88SX60x1 as a SCSI host adapter.
- Triggers 88SX50xx /88SX60x1 initialization sequence.
- Triggers storage devices connected to 88SX50xx /88SX60x1 SATA channels via Common IAL functions.
- CORE driver Interrupt Service Routine (ISR) Wrapper.
- Error Handling: Performs mapping of SAL error status to SCSI port error status.
- Handles hot-plug events, and notifies the SCSI port layer about these events.
- Windows IAL SMART (Self-Monitoring, Analysis, and Reporting Technology) support

The Windows IAL SCSI miniport driver fully supports the Windows drive failure prediction (SMART) IOCTL interface.

The application can use the SMART IOCTL commands (all Windows versions, except of Windows.NET) or WMI predictive failure capabilities to execute SMART commands via the SATA SCSI miniport driver.

Notes

- For an explanation of WMI support for SMART drives see http://www.microsoft.com/whdc/hwdev/driver/WMI/smartdrv.mspx
- For an explanation of SMART IOCTL interface support in Windows see the SMARTAPP example at http://support.microsoft.com/default.aspx?scid=kb;en-us;Q208048.

The following SMART commands are supported by the Windows IAL:

- IDENTIFY - ATA IDENTIFY (not a SMART command, but can be sent using this IOCTL interface)
- ENABLE SMART
- DISABLE SMART
- ENABLE DISABLE AUTOSAVE
- EXECUTE OFFLINE DIAGS
- RETURN STATUS
- READ SMART ATTRIBS
- READ SMART THRESHOLDS
- READ SMART LOG
- WRITE SMART LOG

4.2 Building and Installation

4.2.1 Requirements

The requirements are:

- Computer running Windows 2000/XP/2003.
- 88SX50xx /88SX60x1 software package
- Microsoft Driver Development Kit build environment for Windows 2000/XP/2003.

Note

If you use Windows 2000 DDK (which doesn't include built-int toolchains as in Windows 2003 DDK), then you may also need Microsoft Visual C++ Enterprise edition.

4.2.2 Building

To build the driver:

1. Invoke the DDK Free Build Environment under the windows DDK program group. This starts a command prompt window and sets some environment variables).
2. Change the current directory to **Windows IAL**.
3. Run the batch file **mvsata_build.bat** from the current directory. The generated binary file (***mvsata.sys***) will be located under the directory **install**.

To build a driver with three different levels of log messaging

1. Go to the command prompt window.
2. Change the current directory to **Windows IAL**
3. Run the batch **build_all.bat** with the parameter indicating the DDK directory, for example **build_all.bat C:\winddk\3790**.
4. The batch file creates the **build/Windows** target directory in driver root. The target directory structure is the following:
   - **build/Windows/<Platform>/DebugError/mvSata.sys** - Module prints log messages on error.
   - **build/Windows/<Platform>/DebugFull/mvSata.sys** - Module prints all log messages.
   - **build/Windows/<Platform>/Free/mvSata.sys** - Module prints no messages.
5. Where the <Platform> parameter can be either **i386** (used in 32-bit Windows versions) or **amd64** (used in Windows version for AMD 64-bit processor.)

Note

For building to i386 and amd64 platforms, Windows 2003 DDK must be installed.

4.2.3 Installation of the Driver into a Running System

The installation of the driver can be done by using the Windows Device Manager. Note that the driver .**inf** and **.sys** files location depends on the build type chosen in Section 4.2.2 "Building" on page 37.

4.2.4 Installing Windows 2000/XP/2003 on a 88SX50XX-60X1 Adapter

This section describes how to install and boot from a 88SX50xx /88SX60x1.

1. Copy the **mvSata.inf**, **mvSata.sys** and **txtsetup.oem** files to a diskette.
2. Start a Windows installation on a CD-ROM.
3. When prompted, press the **F6** key (at the beginning of the installation).
4. When prompted, press the **S** key, then insert the diskette in Drive A and continue the installation, following the installation instructions.

4.2.5 Using Windows 2000/XP/2003 SCSI Parameters

View the .**reg** files under the **Install** directory and follow the instructions to enter registry values that affect how the Windows 2000/XP/2003 SCSI manager interprets the generic configuration information of SCSI device drivers. These values can affect all the 88SX50xx /88SX60x1 adapters' installed drivers or a specific driver. They will take effect the next time the driver is started. Following are the registry files supplied and a description of the values they set:

- **mv_256k.reg** specifies the maximum I/O length of transactions issued by the Windows SCSI Port.
- **mv_maxReqs** specifies the number of outstanding SCSI requests per SCSI adapter.

4.2.6 Hot-Swapping Storage Devices

This driver supports hard drive and port multiplier hot-swapping. The storage drive can be connected or disconnected on the fly.

On a connect event the system may take a few seconds before it updates the device manager with regards to the new device added. This is due to the disk initialization sequence, which may be stalled due to hard drive initialization (e.g., when initializing the hard drive's mechanics).

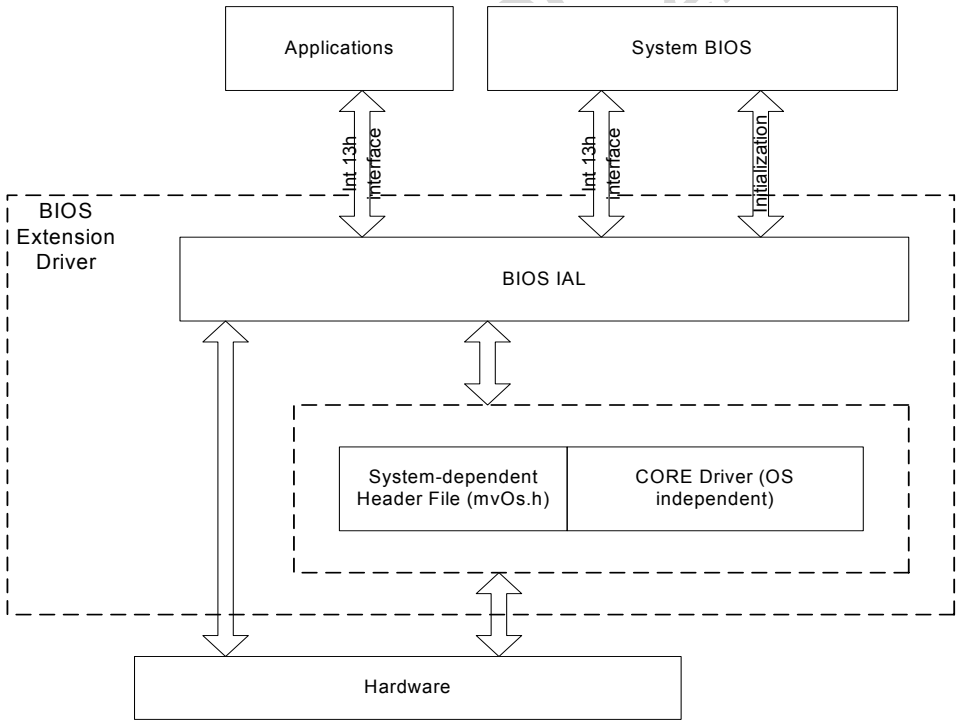
Section 5. Bios Extension Driver Intermediate Application Layer

5.1 Introduction

The main purpose of the 88SX50xx /88SX60x1 BIOS Extension driver is to enable interrupt 13h extensions to hard drives connected to 88SX50xx /88SX60x1 SATA channels.

Figure 4 describes the connectivity of the different software layers.

Figure 4: BIOS IAL Driver Architecture



5.1.1 BIOS Extension Driver Functionality

The BIOS extension driver provides the following functionality:

- Support for PnP and non-PnP system BIOSes
- Initialization of 88SX50xx /88SX60x1 adapters
- Initialization of storage devices connected to 88SX50xx /88SX60x1 SATA channels.
- Hooking up to eight hard drives to interrupt 13h interrupt service routine.
- Servicing interrupt 13h calls

Notes

- BIOS extension driver can hook only eight hard drives to the interrupt 13h service routine chain. If there are more than eight hard drives connected to a specific 88SX50xx /88SX60x1, then only the first eight drives are initialized and hooked to the int13h chain.
- Interrupt 13h servicing is done via source code found in the BIOS IAL. The CORE driver interface for queuing and executing commands is not used for int13h servicing, due to the fact that in run-time the text and data sections of the driver are read-only, and only the caller's stack is read-write. This makes it impossible to call CORE driver functions, which need to maintain data structures and pointers to the status of the queues.
- The BIOS extension driver does not provide hotplug functionality.

5.2 Building and Installation

The build process results in the following files:

- mvFlshUp.com utility - This utility flashes the BIOS extension driver to a specific 88SX50xx /88SX60x1 adapter.
- 5080.IMG, 5081.IMG .... 6041.IMG - These are different BIOS extension drivers, each for a different 88SX50xx /88SX60x1 adapter.

5.2.1 Requirements

The requirements are:

- PC running Windows 2000 or above
- Watcom C/C++ compiler.
- 88SX50xx /88SX60x1 software package.
- DOS for flashing the BIOS extension driver into 88SX50xx /88SX60x1 the adapter's flash.

Notes

- The project build was tested mainly on Watcom C/C++ compiler version 1.0 on a PC running Windows 2000.
- The compiler is an open-source compiler that can be downloaded and installed from http://www.openwatcom.org. The installation should be for DOS 16/32bit and Windows 16/32 bit targets.

5.2.2 Building

To build the driver:

1. Open a command line and change the directory to BiosIAL from the software package root tree.
2. Execute makeBiosDriver.bat.

5.2.3 Installation of the BIOS Extension Driver

To install the BIOS extension driver:

1. Boot the system with DOS (MS-DOS, DR-DOS, or any other variant of DOS).
2. Copy **mvFlshUp.com** and the required BIOS extension driver image to a diskette.
3. Insert the diskette and run **mvFlshUp.com <image name>** (image name is the required BIOS extension driver image).
   The utility provides a list of available adapters to choose from.

5.2.4 Un-installation of the BIOS Extension Driver

To un- install the BIOS extension driver:

1. Boot the system with DOS (MS-DOS, DR-DOS or any other variant of DOS).
2. Copy **mvFlshUp.com** to a diskette.
3. Insert the diskette and run **mvFlshUp.com /erase <adapter Device ID>** (Adapter Device ID is the device ID of the required adapter. This can be 0x5080, 5081 ... 0x6041).
   The utility provides a list of available adapters to choose from.

Section 6. Core Driver

6.1 Introduction

The CORE driver is a software package which is operating system and architecture independent. When a system-dependent header file (mvOs.h file) is attached to the CORE driver, it can access and reserve system resources that it needs for proper functioning.

The CORE driver API and data structures are divided into two main categories:

- **CORE-driver-implemented API and data structures:** Includes most of the functions and data structures being used. These functions are already implemented as part of the CORE driver software suite.
- **User-implemented API and data structure:** Includes several functions and a single data structure which must be implemented by the user in the system-dependent header file (mvOs.h). In Section 6.2 and Section 6.3 the data structure and functions which must be implemented are referred to as "user-implemented".

Note

In this document, references to the "CORE driver API and data structure" mean the combination of CORE-driver-implemented API and data structures plus the user-implemented API and data structure.

The CORE driver provides the following functionality:

- 88SX50xx /88SX60x1 adapter management, initialization, diagnostics and status reporting.
- Execution of UDMA ATA commands.
- Execution Non-UDMA ATA commands.
- Management of software queue of ATA commands per SATA channel. Each queue depth is 31 commands (configurable).
- Management of command completion and events notification, based on call-back functions.
- Interrupt Service.
- I/O Granularity extension for generating a single interrupt on multiple I/Os.
- Channel to Channel communication (aka Target mode): API for communication between two 88SX60X1 adapters.
- Debug messages logging module: Generic debug messages logging module that is used by the CORE driver and can be used by any other software layer.

This section is divided into the following sub-sections:

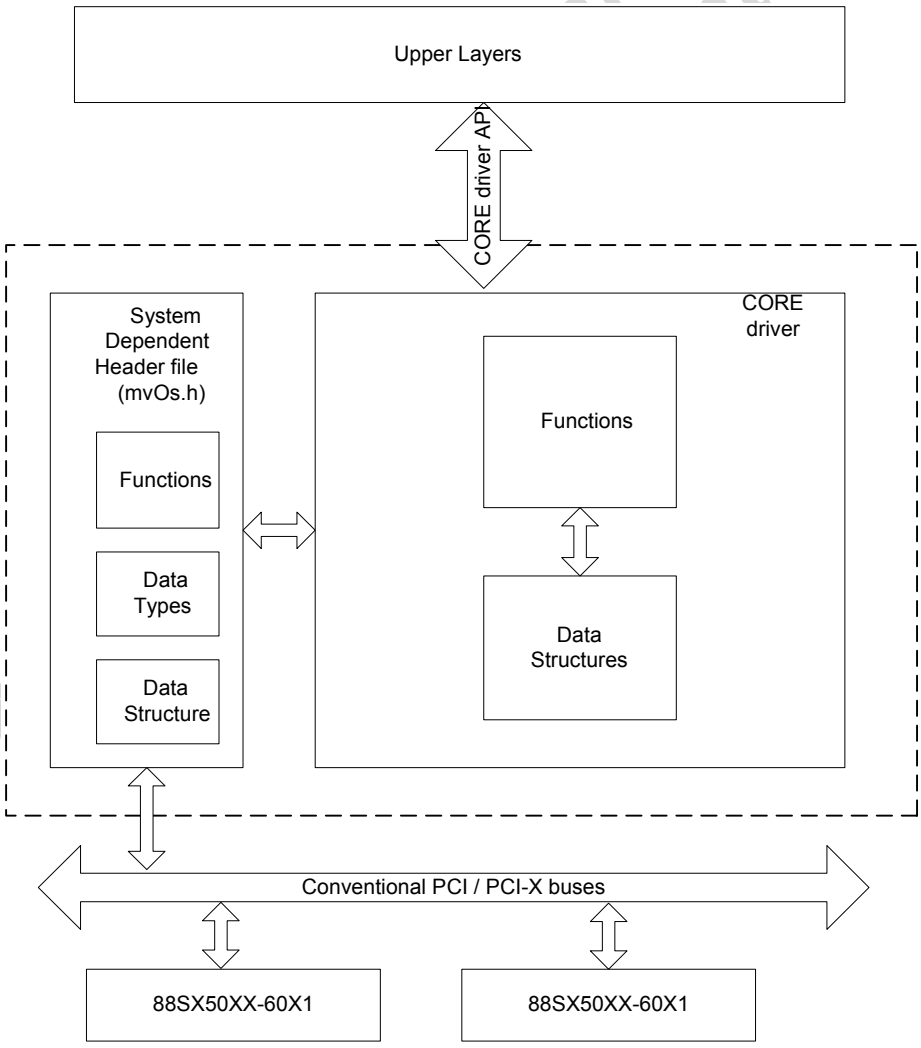
- **"CORE Driver API and Data Structures Summary":** A brief summary that categorizes the CORE driver API into groups of functions and a brief description of the data structure being used. The purpose of this section is to ramp up the user's knowledge of the CORE driver's interface.
- **"Compile-Time CORE Driver Configuration":** Describes the functions, data types and data structures the user must implement for proper integration of the CORE driver in the system. This section also deals with restrictions the user must be aware of when integrating the CORE driver in the system.
- **"CORE Driver API User Implementation Requirements and Restrictions":** Shows how to use the CORE driver from the Intermediate Application Layer (IAL) point of view. It is separated into sub-sections, each describing a specific task the IAL must use.

- **"Detailed CORE Driver Implemented API and Data Structures":** A complete reference to the CORE-driver-implemented API and data structures.
- **"System-Dependent Header File (mvOs.h)":** Details the API, data types, and data structures the user must implement for proper integration of the CORE driver in the system.

6.2 CORE Driver API and Data Structures Summary

The following two sub-sections summarize the CORE driver API and data structures, which are categorized in groups according to their functionality.

Figure 5: CORE Driver API and Data Structures Block Diagrams



6.2.1 CORE Driver API Summary

CORE Driver Adapter Management

mvSataInitAdapter	Initializes 88SX50xx /88SX60x1 adapter.
mvSataShutdownAdapter	Shuts down 88SX50xx /88SX60x1 adapter.
mvSataReadReg	Reads from 88SX50xx /88SX60x1 internal register.
mvSataWriteReg	Writes to 88SX50xx /88SX60x1 internal register.
mvEnableAutoFlush	Enables auto-completion on errors.
mvDisableAutoFlush	Disables auto-completion on errors.

CORE Driver SATA Channel Management

mvSataConfigureChannel	Configures 88SX50xx /88SX60x1-specific SATA channel.
mvSataRemoveChannel	Removes 88SX50xx /88SX60x1-specific SATA channel.
mvSataIsStorageDeviceConnected	Checks if storage device is connected to specific SATA channel.
mvSataChannelHardReset	Causes 88SX50xx /88SX60x1 to reset a specific SATA channel.
mvSataConfigEdmaMode	Configures specific SATA channel's EDMA mode.
mvSataEnableChannelDma	Enables specific SATA channel's EDMA mode.
mvSataDisableChannelDma	Disables specific SATA channel's EDMA mode.
mvSataFlushDmaQueue	Flushes corresponding SATA channel's request queue.
mvSataNumOfDmaCommands	Returns number of posted DMA commands on a specific SATA channel request queue.
mvSataSetIntCoalParams	Sets interrupt coalescing for specific quad SATA channels 88SX50xx /88SX60x1 (or octal SATA channels on 88SX60X1 adapters).
mvSataSetChannelPhyParams	Sets the AMP and PRE values of a specific SATA channel.
mvSataChannelPhyPowerOn	Powers up the physical interface of a specific SATA channel.
mvSataChannelPhyShutdown	Shuts down the physical interface of a specific SATA channel.
mvSataChannelFarLoopbackDiagnostic	Performs an external loopback test of specific SATA channel.
mvSataEnableStaggeredSpinUp	Enables SATA communication on a specific SATA channel.
mvSataEnableStaggeredSpinUpAll	Enables SATA communication on all SATA channels.
mvSataDisableStaggeredSpinUp	Disables SATA communication on a specific SATA channel.
mvSataDisableStaggeredSpinUpAll	Disables SATA communication on all SATA channels.

`mvSataSetInterfaceSpeed`	Modifies SATA speed (Gen I/II) on a specific SATA channel.
`mvSataGetInterfaceSpeed`	Returns SATA speed (Gen I/II) on a specific SATA channel.

Non-UDMA ATA Command Execution Task (Polling driven)

`mvStorageDevATAExecuteNonUDMACommand`	Issues user-specific non-UDMA ATA command to specific storage device.
`mvStorageDevATAIdentifyDevice`	Issues IDENTIFY DEVICE ATA command to specific storage device.
`mvStorageDevATASetFeatures`	Issues SET FEATURES ATA command to specific storage device.
`mvStorageDevATAIdleImmediate`	Issues IDLE IMMEDIATE ATA command to specific storage device
`mvStorageDevATASoftResetDevice`	Issues SRST sequence to specific storage device.
`mvStorageDevATAStarSoftResetDevice`	Issues SRST sequence to specific storage device but does not poll for disk ready status.
`mvStorageIsDeviceBsyBitOff`	Returns MV_TRUE if storage device busy bit is off.
`mvStorageDevExecutePIO`	Issues user-specific PIO ATA command to specific storage device (using ATA registers data structure as input and output for the function).
`mvStorageDevSetDeviceType`	Sets the storage device type connected to a specific SATA channel.
`mvStorageDevGetDeviceType`	Retrieves the storage device type connected to a specific SATA channel

Port multiplier functions (Polling driven)

`mvPMDevReadReg`	Reads a port multiplier's internal register.
`mvPMDevWriteReg`	Writes to a port multiplier's internal register.
`mvPMDevEnableStaggeredSpinUp`	Enables communication on a specific SATA channel on a port multiplier's device ports.
`mvPMDevEnableStaggeredSpinUpAll`	Enables communication on all SATA channels on a port multiplier's device ports.

Queue Asynchronous ATA Commands (Interrupt driven)

`mvSataQueueCommand`	Adds ATA command to an asynchronous commands queue.

Command Completion and Event Notification (User-Implemented)

`mvSataCommandCompletionCallBack`	Callback function called upon specific command completion.
`mvSataEventNotify`	Event notification, upon event (Interrupt) from 88SX50xx /88SX60x1.

Interrupt Service Routine

`mvSataInterruptServiceRoutine`	Interrupt Service Routine
`mvSataMaskAdapterInterrupt`	Masks 88SX50xx /88SX60x1 adapter interrupts.
`mvSataUnmaskAdapterInterrupt`	Unmasks 88SX50xx /88SX60x1 adapter interrupts.
`mvSataSetInterruptScheme`	Modifies CORE driver interrupt scheme.
`mvSataCheckPendingInterrupt`	Used for checking whether an interrupt is pending only in the interrupt scheme, where interrupt handling is performed in a task and not in the ISR.

System Routines (User Implemented)

`mvOsSemInit`	Initializes semaphore.
`mvOsSemTake`	Takes ownership of a semaphore.
`mvOsSemRelease`	Releases ownership of a semaphore.
`mvOsSaveFlagsAndMaskCPUInterrupts`	Saves CPU flags and masks its interrupts.
`mvOsRestoreFlags`	Restore CPU flags.
`mvMicroSecondsDelay`	Delays function in micro-seconds resolution.
`mvLogMsg`	User implemented function that enables logging of CORE driver messages.

Logger functions

mvLogRegisterModule	Associate the module with the logger.
mvLogSetModuleFilter	Set log filter for the module
mvLogGetModuleFilter	Get log filter for the module
mvLogMsg	Prints log message

Interrupt coalescing in I/O granularity functions

mvSataEnableIoGranularity	Enables support for interrupt coalescing in I/O granularity.

Channel to channel communication (aka Target mode) functions

mvSataC2CInit	Initializes channel to channel communication mode for a specific SATA channel.
mvSataC2CStop	Disables channel to channel communication mode for a specific SATA channel.
mvSataC2CSendRegisterDeviceToHostFIS	Sends Register Host to Device FIS.
mvSataC2CActivateBmDma	Activates Bus Master DMA for the channel.
mvSataC2CResetBmDma	Resets Bus Master DMA for the channel.

6.2.2 CORE Driver Data Structure Summary

Data structures modified by IAL and CORE driver.

MV_SATA_ADAPTER	Data structure representing 88SX50xx /88SX60x1 adapter.
MV_SATA_CHANNEL	Data structure representing a specific 88SX50xx /88SX60x1 SATA channel.
MV_STORAGE_DEVICE_REGISTERS	Data structure representing a storage device's registers. Used upon completion of both UDMA and non-UDMA ATA commands.
MV_SATA_EDMA_PRD_ENTRY	Data structure representing a single entry in the 88SX50xx / 88SX60x1 PRD table.
MV_UDMA_COMMAND_PARAMS	Data structure for passing UDMA ATA command parameters to the `mvSataQueueCommand()` function.
MV_NONE_UDMA_COMMAND_PARAMS	Data structure for passing non-UDMA ATA command parameters to the `mvSataQueueCommand()` function.
MV_QUEUE_COMMAND_INFO	Data structure for queuing ATA commands through the `mvSataQueueCommand()` function.
MV_OS_SEMAPHORE	User-implemented data structure, used for locking/unlocking MV_SATA_ADAPTER and MV_SATA_CHANNEL.

6.3 Compile-Time CORE Driver Configuration

Several parameters that can be configured in compile time. These are discussed in this section.

6.3.1 CORE Driver Logging Mechanism

For logging CORE driver debug messages, the following must be added to the user-specific mvOs.h file:

#define MV_LOG_DEBUG - For logging all debug messages.

#define MV_LOG_ERROR - For logging only error messages.

It is recommended to disable all logging mechanisms on a released driver, to minimize the driver's footprint and maximize performance.

6.3.2 CORE Driver Queue Size

The default behavior of the CORE driver in compile time is that its queue size is 31 commands.

It is possible to modify this behavior in compile by adding the following lines to the user-specific mvOs.h file:

#define MV_SATA_OVERRIDE_SW_QUEUE_SIZE

#define MV_SATA_REQUESTED_SW_QUEUE_SIZE <queue size>
where <queue size> is the requested queue size (between 1 and 31).

6.3.3 Channel-to-Channel Communication Support (aka Target Mode)

For enabling channel-to-channel communication support, the following line must be added to the user-specific mvOs.h file:

```
#define MV_SATA_C2C_COMM
```

6.3.4 I/O-Granularity Interrupt Acceleration

To enable I/O-Granularity interrupt acceleration, the following line must be added to the user-specific mvOs.h file:

```
#define MV_SATA_IO_GRANULARITY
```

6.4 CORE Driver API User Implementation Requirements and Restrictions

This section describes the requirements that must be implemented by the user and the restrictions the user must adhere to when using the CORE driver API.

6.4.1 Requirements

For the CORE driver to work properly, user-implemented functions, data types and data structures must be implemented. In this document these requirements are marked as "User-implemented". For clarification purposes they are listed again in this section.

Note

These functions, data types and data structures must all be implemented or declared in the system-dependent header file (mvOs.h).

6.4.1.1 Command Completion and Event Notification

The user must implement the following functions:

```
mvSataCommandCompletionCallBack
mvSataEventNotify
```

6.4.1.2 System Functions

The user must implement the following functions:

```
mvOsSemInit
mvOsSemTake
mvOsSemRelease
mvMicroSecondsDelay
###### ADD Logging mechanism functions ######
```

Note

If a locking mechanism is performed in higher layers above the CORE driver, then the user may consider not using the CORE driver's locking mechanism, by defining the above functions in the user-specific mvOs.h file with a "while (0) {} " statement.

6.4.1.3 Data Types

The user must implement the following data types:

MV_VOID

MV_U32

MV_U16

MV_U8

MV_VOID_PTR

MV_U32_PTR

MV_U16_PTR

MV_U8_PTR

MV_CHAR_PTR

MV_BUS_ADDR_T

MV_CPU_FLAGS

6.5 Detailed CORE Driver Implemented API and Data Structures

6.5.1 Enumerators and Defines

6.5.1.1 Enumerators

MV_BOOLEAN - Enumerator for the value MV_TRUE and MV_FALSE

MV_UDMA_TYPE - Enumerator for either MV_UDMA_TYPE_READ or MV_UDMA_TYPE_WRITE

MV_FLUSH_TYPE - Enumerator for either MV_FLUSH_TYPE_CALLBACK or MV_FLUSH_TYPE_NONE

MV_COMPLETION_TYPE - Enumerator for either MV_COMPLETION_TYPE_NORMAL or MV_COMPLETION_TYPE_ERROR or MV_COMPLETION_TYPE_ABORT

MV_EVENT_TYPE - Enumerator for either MV_EVENT_TYPE_ADAPTER_ERROR or MV_EVENT_TYPE_SATA_CABLE

MV_SATA_CABLE_EVENT - Enumerator for either MV_SATA_CABLE_EVENT_DISCONNECTED, MV_SATA_CABLE_EVENT_CONNECTED or MV_SATA_CABLE_EVENT_PM_HOT_PLUG.

MV_EDMA_MODE - Enumerator for either MV_EDMA_MODE_QUEUED or MV_EDMA_MODE_NOT_QUEUED

MV_QUEUE_COMMAND_RESULT - Enumerator for either MV_QUEUE_COMMAND_RESULT_OK, MV_QUEUE_COMMAND_RESULT_QUEUED_MODE_DISABLED, MV_QUEUE_COMMAND_RESULT_FULL, MV_QUEUE_COMMAND_RESULT_BAD_LBA_ADDRESS or MV_QUEUE_COMMAND_RESULT_BAD_PARAMS

MV_NON_UDMA_PROTOCOL - Enumerator for either MV_NON_UDMA_PROTOCOL_NON_DATA, MV_NON_UDMA_PROTOCOL_PIO_DATA_IN, or MV_NON_UDMA_PROTOCOL_PIO_DATA_OUT.

MV_QUEUED_COMMAND_TYPE - Enumerator for either MV_QUEUED_COMMAND_TYPE_UDMA, or

MV_QUEUED_COMMAND_TYPE_NONE_UDMA.

MV_SATA_C2C_MODE - Enumerator for channel-to-channel feature. It determines the channel's role in the channel-to-channel communication mode of either MV_SATA_C2C_MODE_INITIATOR or MV_SATA_C2C_MODE_TARGET

MV_C2C_EVENT_TYPE - Enumerator for channel-to-channel feature. Can be either MV_C2C_REGISTER_DEVICE_TO_HOST_FIS_DONE, or MV_C2C_REGISTER_DEVICE_TO_HOST_FIS_ERROR, or MV_C2C_BM_DMA_DONE, or MV_C2C_BM_DMA_ERROR.

MV_IOG_COMMAND_TYPE - Enumerator for I/O Granularity feature. Can be either MV_IOG_COMMAND_TYPE_FIRST or MV_IOG_COMMMAND_TYPE_NEXT.

MV_SATA_INTERRUPT_SCHEME - Enumerator for either MV_SATA_INTERRUPT_HANDLING_IN_ISR, MV_SATA_INTERRUPT_HANDLING_IN_TASK, or MV_SATA_INTERRUPTS_DISABLED.

MV_SATA_IF_SPEED - Enumerator for either MV_SATA_IF_SPEED_1_5_GBPS, MV_SATA_IF_SPEED_3_GBPS, MV_SATA_IF_SPEED_NO_LIMIT and MV_SATA_IF_SPEED_INVALID.

MV_SATA_DEVICE_TYPE - Enumerator for either MV_SATA_DEVICE_TYPE_UNKOWN, MV_SATA_DEVICE_TYPE_ATA_DISK, MV_SATA_DEVICE_TYPE_ATAPI_DISK, or MV_SATA_DEVICE_TYPE_PM.

6.5.1.2 Defines

MV_SATA_DEVICE_ID_5080,MV_SATA_DEVICE_ID_5081, MV_SATA_DEVICE_ID_5040, MV_SATA_DEVICE_ID_5041, MV_SATA_DEVICE_ID_6081, and MV_SATA_DEVICE_ID_6041 - The different device ID per 88SX50xx /88SX60x1 adapter's PCI configuration space.

MV_SATA_VENDOR_ID - The vendor ID for the 88SX50xx /88SX60x1 adapter PCI configuration space (equals 0x11AB).

MV_SATA_CHANNELS_NUM - Number of maximum serial ATA channels in a single 88SX50xx /88SX60x1 (equals 8).

MV_SATA_UNITS_NUM - Number of maximum serial ATA units in a single 88SX50xx /88SX60x1 (equals 2).

MV_EDMA_QUEUE_LENGTH - Maximum number of outstanding UDMA ATA commands (equals 32).

MV_EDMA_REQUEST_ENTRY_SIZE - Size of a single entry in a request queue (equals 32).

MV_EDMA_RESPONSE_ENTRY_SIZE - Size of a single entry in a response queue (equals 8).

MV_EDMA_REQUEST_QUEUE_SIZE - Size of an entire request queue (equals 32 * 32 = 1024 bytes).

MV_EDMA_RESPONSE_QUEUE_SIZE - Size of an entire response queue (equals 32 * 8 = 256 bytes).

MV_EDMA_PRD_ENTRY_SIZE - Size of a single entry in a PRD table (equals 16).

MV_EDMA_PRD_SNOOP_FLAG - Flag indicating a snoop operation to be performed on the relevant PRD entry.

MV_EDMA_PRD_EOT_FLAG - Flag indicating the end of a PRD table.

MV_ATA_IDENTIFY_DEV_DATA_LENGTH - Number of fields in the data array returned from a storage device as a response for an IDENTIFY DEVICE ATA command (equals 256).

MV_ATA_MODEL_NUMBER_LEN - Length of the model number, as extracted from the IDENTIFY DEVICE ATA command (equals 0x40).

MV_C2C_MESSAGE_SIZE - Definition for channel-to-channel communication feature. The maximum size of user data transmitted with Register Device to Host FIS - equals to 10

Logger Defines

MV_LOG_DEBUG - The preprocessor variable. When one is set, all log debug messages are printed.

MV_LOG_ERROR - The preprocessor variable. When one is set, all log error messages are printed.

Log Levels Filter Mask

MV_DEBUG_MASK - Filter value for general debug messages (0x1).

MV_DEBUG_INIT_MASK - Filter value for debug messages during initialization (0x2).

MV_DEBUG_INTERRUPTS_MASK - Filter value for debug messages from interrupt service routine (0x4).

MV_DEBUG_SATA_LINK_MASK - Filter value for debug messages related to SATA link layer (0x8).

MV_DEBUG_UDMA_COMMAND_MASK - Filter value for general debug messages related to UDMA commands (0x10).

MV_DEBUG_NON_UDMA_COMMAND_MASK - Filter value for debug messages related to non-UDMA commands (0x20).

MV_DEBUG_ERROR_MASK - Filter value for error messages (0x40).

MV_DEBUG_PM_MASK - Filter value for debug messages related to port multiplier (0x80).

MV_DEBUG_ALL_MASK - Filter value to print all log messages.

Log Message Type

MV_DEBUG - General debug message (0x0).

MV_DEBUG_INIT - Debug messages during initialization (0x1).

MV_DEBUG_INTERRUPTS - Debug messages from interrupt service routine (0x2).

MV_DEBUG_SATA_LINK - Debug messages related to SATA link layer(0x3).

MV_DEBUG_UDMA_COMMAND - Debug messages related to UDMA commands (0x4).

MV_DEBUG_NON_UDMA_COMMAND - Debug messages related to non-UDMA commands (0x5).

MV_DEBUG_ERROR - Error message (0x6).

MV_DEBUG_PM - Debug messages related to port multiplier (0x7).

MV_RAW_MSG_ID - Specified instead of module ID in `mvLogMsg()` function call.
The message is printed without specifying module ID and debug level.

MV_LOG_PRINT - Macro that defines the OS-dependent print function used by the logger.

6.5.2 Data Structures

MV_SATA_ADAPTER

Fields set by the Intermediate Application Layer

MV_U32 adapterId - A unique number for each 88SX50xx /88SX60x1 adapter. This number is only used for indexing multiple 88SX50xx /88SX60x1s for the purpose of log messages.

MV_VOID_PTR IALData - is scratchpad data used only by IAL.

MV_U8 pciConfigRevisionId - Device revision number of the adapter, as reported from the revision number in the adapter's PCI configuration space.

MV_U16 pciConfigDeviceId - Device Id number of the adapter, as reported from the revision number in the adapter's PCI configuration space.

MV_BUS_ADDR_T adapterIoBaseAddress - A CPU address for accessing an 88SX50xx /88SX60x1 adapter's function0, BAR0 base address.

MV_U32 intCoalThre[MV_SATA_UNITS_NUM] - Array of two fields indicating the Interrupt Coalescing Threshold for each quad SATA channel.

MV_U32 intTimeThre[MV_SATA_UNITS_NUM] - Array of two fields indicating the Interrupt Time Threshold in each quad SATA channel.

mvSataEventNotify - Pointer to a call-back function, used by CORE driver to indicate error or status change in an 88SX50xx /88SX60x1 adapter.

MV_SATA_CHANNEL *sataChannel [MV_SATA_CHANNEL_NUM] - Array of MV_SATA_CHANNEL_NUM count in fields (equals 8). Each is a pointer to MV_SATA_CHANNEL data structures.

MV_U32 pciCommand - A 32-bit field describing the 88SX50xx /88SX60x1 PCI command register.

MV_U32 pciSerrMask - A 32-bit field describing the bit masking register of the SERR# signal in the 88SX50xx /88SX60x1 adapter.

MV_U32 pciInterruptMask - A 32-bit field describing the bit masking register of the PCI unit Interrupt Cause register.

MV_SATA_CHANNEL

Fields set by the Intermediate Application Layer

MV_U8 channelNumber - Logical number from 0.. MV_SATA_CHANNEL_NUM-1 (equals 7).

MV_DMA_REQUEST_QUEUE_ENTRY *requestQueue - CPU address (pointer) to the request queue of the channel.

MV_DMA_RESPONSE_QUEUE_ENTRY *responseQueue - CPU address (pointer) to the response queue of the channel.

MV_U32 requestQueuePCIHiAddress - High 32-bit PCI address of the SATA channel request queue.

MV_U32 requestQueuePCILowAddress - Low 32-bit PCI address of the SATA channel request queue.

MV_U32 responseQueuePCIHiAddress - High 32-bit PCI address of the SATA channel response queue.

MV_U32 responseQueuePCILowAddress - Low 32-bit PCI address of the SATA channel response queue.

MV_QUEUE_COMMAND_INFO

MV_QUEUED_COMMAND_TYPE type - Type of ATA command—UDMA or non-UDMA

MV_U8 PMPort - Destination port multiplier's port number.

union

{

MV_UDMA_COMMAND_PARAMS udmaCommand;

MV_NONE_UDMA_COMMAND_PARAMS NoneUdmaCommand;

} commandParams - parameters of the ATA command.

MV_UDMA_COMMAND_PARAMS

MV_UDMA_TYPE readWrite - Whether the command is read or write.

MV_BOOLEAN isEXT - Indicates if the command is LBA48 or legacy command.

MV_U32 lowLBAAddress -The LSB 32 bits of the LBA sector address.

MV_U16 highLBAAddress - The MSB 16 bits of the sector LBA address.

MV_U16 numOfSectors - Number of sectors to transfer.

MV_U32 prdLowAddr - A low 32-bit PCI address to the PRD table of the command.

MV_U32 prdHighAddr- A high 32-bit PCI address to the PRD table of the command.

mvSataCommandCompletionCallBack_t callBack A callback function that is called when command execution has been completed.

MV_VOID_PTR commandId - An arbitrary ID that uniquely identifies the command.

If I/O Granularity is enabled, then the following fields are valid:

MV_BOOLEAN ioGranularityEnabled - MV_TRUE if I/O Granularity feature is enabled.

MV_IOG_COMMAND_TYPE iogCommandType - Equals MV_IOG_COMMAND_TYPE_FIRST if this is the first command in the I/O granularity command chain. Otherwise equals MV_IOG_COMMAND_TYPE_NEXT.

union {MV_U8 transId; MV_U8 transCount} iogGranularityCommandParam - If iogCommandType indicates first command, then union refers to transCount, which indicates the number of I/Os the chain holds. Otherwise the union refers to transId, which refers to the transaction chain ID of which this command is part.

MV_U8 iogCurrentTransId - If iogCommandType indicates first command, then iogCurrentTransId is output from the CORE driver and it indicates the transaction ID allocated to the chain of commands. Otherwise, iogCurrentTransId holds the transaction ID to which this command refers.

Notes

- Only in the case of a 48-bit LBA-feature-set-compliant storage device, `numOfSectors` can be more than 256 sectors and LBA address is 48-bit. Otherwise, `numOfSectors` can be a maximum of 256 sectors and LBA address consists of a 28-bit address.
- If `numOfSectors` is zero and the isEXT field is MV_FALSE (48-bit address feature set is not in use), then the UDMA command transfer size is 256 sectors.
- If `numOfSectors` is zero and the isEXT field is MV_TRUE (48-bit address feature set is in use), then the UDMA command transfer size is 65,536 sectors.

MV_NONE_UDMA_COMMAND_PARAMS

MV_NON_UDMA_PROTOCOL protocolType - Protocol of the requested ATA command to perform.

MV_BOOLEAN isEXT- Equals MV_TRUE if the command is an LBA 48-bit extended command.

MV_U16_PTR bufPtr - Pointer to a buffer that the PIO data-out/in ATA command transfers from/to (must be word (16-bit) byte aligned).

MV_U32 count - Number of words to transfer from/to buffer.

MV_U16 features- The value to be written to the FEATURES register.

MV_U16 sectorCount - The value to be written to the SECTOR COUNT register.

MV_U16 lbaLow - The value to be written to the LBA LOW register.

MV_U16 lbaMid - The value to be written to the LBA MID register

MV_U16 lbaHigh - The value to be written to the LBA HIGH register.

MV_U8 device - The value to be written to the DEVICE register.

MV_U8 command - The value to be written to the COMMAND register.

mvSataCommandCompletionCallBack_t callBack - A callback function that is called when command execution has been completed

MV_VOID_PTR commandId -An arbitrary ID that uniquely identifies the command.

MV_STORAGE_DEVICE_REGISTERS

Fields set either by the IAL or CORE driver (depending on the function used with this data structure).

MV_U8 errorRegister - Storage device's error register

MV_U16 sectorCountRegister - Storage device's sector count register.

MV_U16 lbaLowRegister - Storage device's low LBA register

MV_U16 lbaMidRegister - Storage device's mid LBA register

MV_U16 lbaHighRegister - Storage device's high LBA register

MV_U8 deviceRegister - Storage device's device register

MV_U8 statusRegister - Storage device's status register

Note

The 16-bit fields in the MV_STORAGE_DEVICE_REGISTERS data structure are all used only in 48-bit LBA storage devices. When using legacy 28-bit LBA addressing, only the 8-bit LSB of these fields are used.

MV_SATA_EDMA_PRD_ENTRY

Fields set by the Intermediate Application Layer:

MV_U32 lowBaseAddr - Low 32-bit address of the buffer the IAL needs to read/write.

MV_U16 byteCount - Length of the buffer (maximum 64 KByte).

MV_U16 flags - Flags indicating how the 88SX50xx /88SX60x1 adapter must handle this buffer.

MV_U32 highBaseAddr - High 32-bit address of the buffer the IAL needs to read/write.

MV_U32 reserved - Reserved field that IAL must not use.

Note

See the 88SX50xx /88SX60x1 adapter datasheet for further information about the fields described above.

6.5.3 CORE Driver API

6.5.3.1 CORE Driver Adapter Management

The following CORE driver adapter management functions initialize and configure the 88SX50xx /88SX60x1 adapter and its SATA channels.

*MV_BOOLEAN mvSataInitAdapter (MV_SATA_ADAPTER *pAdapter)*

DESCRIPTION

Initializes 88SX50xx /88SX60x1 adapter by writing the values passed in the MV_SATA_ADAPTER to the 88SX50xx /88SX60x1 hardware.

IAL allocates an MV_SATA_ADAPTER in the system memory then updates fields that are marked as "Fields set by IAL" in the MV_SATA_ADAPTER documentation above. The IAL then calls `mvSataInitAdapter`, passing to it a pointer to the allocated MV_SATA_ADAPTER data structure.

The mvSataInitAdapter performs the following:

1. Masks the adapter's interrupts
2. Using the pciConfigDeviceId and pciConfigRevisionId, identifies the adapter and accordingly sets the internal workarounds needed for the adapter. If the revision ID is greater than the latest supported, the workarounds of the latest revision ID are implemented.
3. Reads the pre-emphasis and amplitude parameters of all SATA channels from the adapter and saves them in the adapter data structure. This is done to preserve user-specific pre-emphasis and amplitude that were previously configured (for example the configuration can be either by POST or via an on-board TWSI EEPROM). Note that at every reset performed by software to the SATA channel, the saved pre-emphasis and amplitude values are written to the SATA channel.
4. Performs internal software reset of the adapter. If the adapter is an 88SX50XX, then the function writes its default values to the 88SX50XX internal registers. If the adapter is an 88SX60X1, then the function uses the global soft reset feature, which reverts all 88SX60X1 internal registers to their default value (except PCI configuration space). See the 88SX60X1 datasheet for further information about global soft reset.
5. Enables adapter's LEDs.
6. Configures PCI registers with the user's required parameters.

INPUT

pAdapter - A pointer to the MV_SATA_ADAPTER data structure that holds all information to access the corresponding 88SX50xx /88SX60x1 adapter.

RETURN

MV_TRUE on success

MV_FALSE on failure

Note

The `mvSataInitAdapter` function does not initialize the SATA channels in the 88SX50xx /88SX60x1 adapter.

*MV_BOOLEAN mvSataShutdownAdapter (MV_SATA_ADAPTER *pAdapter)*

DESCRIPTION

Shuts down a specific 88SX50xx /88SX60x1 adapter.

IAL uses this function either before booting the system, or for shutting down a non-usable 88SX50xx / 88SX60x1 adapter.

INPUT

pAdapter - A pointer to the MV_SATA_ADAPTER data structure that holds all information to access the corresponding 88SX50xx /88SX60x1 adapter.

RETURN

MV_TRUE on success

MV_FALSE on failure

*MV_U32 mvSataReadReg (MV_SATA_ADAPTER *pAdapter, MV_U32 regOffset)*

DESCRIPTION

Returns the value of the register that has the offset regOffset, in an 88SX50xx /88SX60x1 adapter pointed to by pAdapter.

INPUT

pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

regOffset - An offset to the register to read.

RETURN

32 bits that hold the value of the register.

Note

Several 88SX50xx /88SX60x1 internal registers are implemented as 8-bit or 16-bit registers. The above function can still be used for reading these registers. The read operation result in an 8-bit register will be in the first 8 least significant bits of the return value. The reading operating in a 16-bit register will be in the first 16 least significant bits of the return value.

Note

Accessing a drive's registers when EDMA is enabled is an unpredictable action.

*MV_VOID mvSataWriteReg (MV_SATA_ADAPTER *pAdapter, MV_U32 regOffset, MV_U32 regValue)*

DESCRIPTION

Writes a value regValue to a register with offset regOffset in an 88SX50xx /88SX60x1 adapter pointed to by pAdapter.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

regOffset - An offset to the register to write.

regValue - The value to write to the register.

RETURN
N/A

Note

Several 88SX50xx /88SX60x1 internal registers are implemented as 8-bit or 16-bit registers. The above function can still be used for writing to these registers. The write operation uses the 8 least significant bits in the regValue parameter for the 8-bit registers, and uses the 16 least significant bits in the regValue parameter for 16-bit registers.

MV_VOID mvEnableAutoFlush(MV_VOID)

DESCRIPTION

If an error occurs on a specific SATA channel, the first command is completed with the error, and the following commands are completed with software abort. This function enables this auto-completion on error feature.

INPUT

N/A

RETURN

N/A

MV_VOID mvDisableAutoFlush(MV_VOID)

DESCRIPTION

If an error occurs on a specific SATA channel, the first command is completed with the error, and the following commands are completed with software abort. This function disables this auto-completion on error feature.

INPUT

N/A

RETURN

N/A

6.5.3.2 CORE Driver SATA Channel Management

MV_BOOLEAN mvSataConfigureChannel (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex)

DESCRIPTION
Configures SATA channel whose data structure is pointed to by pAdapter and channelIndex.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds information to access the 88SX50xx /88SX60x1 device.

channelIndex - An index to a specific 88SX50xx /88SX60x1 channel.

RETURN
MV_TRUE on success

MV_FALSE on failure

Note

Refer to the MV_SATA_CHANNEL data structure regarding parameters that must be filled in by the IAL before calling the `mvSataConfigureChannel()` function.

MV_BOOLEAN mvSataRemoveChannel (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex)

DESCRIPTION
Removes data structures and other parameters used for the specific SATA channel that is indicated by pAdapter and channelIndex.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds information to access the 88SX50xx /88SX60x1 device.

channelIndex - An index to a specific 88SX50xx /88SX60x1 channel.

RETURN
MV_TRUE on success

MV_FALSE on failure

MV_BOOLEAN mvSataIsStorageDeviceConnected (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex)

DESCRIPTION
Checks the 88SX50xx /88SX60x1 adapter to determine if a storage device is connected to a specific SATA channel, indexed by channelIndex.

INPUT
pAdapter - A pointer to a MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - An index to a specific 88SX50xx /88SX60x1 channel.

RETURN
MV_TRUE if a SATA storage device is connected

MV_FALSE if a SATA storage device is not connected

Note

This function is used to check if a storage device is connected directly to a 88SX50xx /88SX60x1 adapter. It can't be used for checking if a storage device is connected to a port multiplier's device port.

*MV_BOOLEAN mvSataChannelHardReset (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex)*

DESCRIPTION
Hardware resets a specific SATA channel.

If the adapter doesn't support staggered spinup, as in the case of 88SX50XX, then an OOB sequence is automatically triggered when the SATA channel reset sequence finished.

If the adapter does support staggered spinup, and it was previously enabled by the `mvSataEnableStaggeredSpinUp()` function, then an OOB sequence is triggered by software after the SATA channel reset sequence has been completed. Otherwise an OOB sequence is not triggered.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - An index to a specific 88SX50xx /88SX60x1 channel.

RETURN
MV_TRUE on success

MV_FALSE on failure

Note

This function does not poll for disk ready status (ATA status BSY bit transits to '0') as in Release 3.1.2 and older releases. To do this, the IAL must use the `mvSataChannelHardReset()` function to perform the reset and OOB sequence, and afterwards use the `mvStorageIsDeviceBsyBitOff()` function for polling on disk ready status.

*MV_BOOLEAN mvSataConfigEdmaMode (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex, MV_EDMA_MODE dmaMode, MV_U8 queueDepth)*

DESCRIPTION
Configures a specific SATA channel EDMA mode (queued commands features set or non-queued commands feature set).

If the queued commands features set is selected, it configures the queue depth the SATA channel may use.

This function also sets EDMA burst sizes to maximum supported by the adapter.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - An index to a specific 88SX50xx /88SX60x1 channel.

dmaMode - Can be either MV_EDMA_MODE_QUEUED, indicating queued commands feature set, or MV_EDMA_MODE_NOT_QUEUED, indicating non-queued feature set.

queueDepth - Valid only if queued commands feature set is selected. This parameter indicates the queue depth the SATA channel may use.

RETURN
MV_TRUE on success

MV_FALSE on failure

*MV_BOOLEAN mvSataEnableChannelDma (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex)*

DESCRIPTION
Enables the software ATA commands queuing engine in a specific SATA channel.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - An index to a specific 88SX50xx /88SX60x1 channel.

RETURN
MV_TRUE on success

MV_FALSE on failure

*MV_BOOLEAN mvSataDisableChannelDma (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex)*

DESCRIPTION
Disables the software ATA commands queuing engine in a specific SATA channel.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - An index to a specific 88SX50xx /88SX60x1 channel.

RETURN
MV_TRUE on success

MV_FALSE on failure

*MV_BOOLEAN mvSataFlushDmaQueue (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex, MV_FLUSH_TYPE flushType)*

DESCRIPTION
Flush posted ATA commands on a specific 88SX50xx /88SX60x1 SATA channel software commands queue. If the flushType parameter is MV_FLUSH_TYPE_CALLBACK, then all callback functions of the posted and still not completed commands are called with a flush indication.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - An index to a specific 88SX50xx /88SX60x1 channel.

execCallBack - If this parameter equals MV_FLUSH_TYPE_CALLBACK, then all callback functions for all posted and not completed ATA commands are called. If it equals MV_FLUSH_TYPE_NONE then the queue is flushed without calling the callback functions.

RETURN
MV_TRUE on success

MV_FALSE on failure

*MV_U8 mvSataNumOfDmaCommands (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex)*

DESCRIPTION
Returns the number of posted ATA commands on the software ATA commands queue in a specific SATA channel.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - An index to a specific 88SX50xx /88SX60x1 channel.

RETURN
In the case of success, the return value is number between 0.. MV_MAX_CMD - Number of UDMA ATA commands posted.

In the case of failure, the return value is 0xFF.

*MV_BOOLEAN mvSataSetIntCoalParams (MV_SATA_ADAPTER *pAdapter, MV_U8 sataUnit, MV_U32 intCoalThre, MV_U32 intTimeThre)*

DESCRIPTION
Sets the interrupt coalescing for a specific SATA unit (each SATA unit contains quad SATA channels).

If the adapter supports all units interrupt coalescing (as in the case of the 88SX60X1 adapter) then sataUnit equals 0xFF, which signals the CORE driver to enable this feature.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

sataUnit - An index to a specific 88SX50xx /88SX60x1 SATA unit. When sataUnit equals 0xFF, this is an indication for the CORE to enable the all units interrupt coalescing feature.

intCoalThre - Parameter indicating the Interrupt Coalescing Threshold to be set.

intTimeThre - Parameter indicating the Interrupt Time Threshold to be set.

RETURN
MV_TRUE on success

MV_FALSE on failure

Notes

- For further information about the Interrupt Time Threshold and Interrupt Coalescing Threshold registers, see the 88SX50xx /88SX60x1 datasheet.
- This function can be called in runtime without deactivating any SATA channel.
- When this function is called, intCoalThre and intTimeThre in the MV_SATA_ADAPTER data structure are automatically updated.

*MV_BOOLEAN mvSataSetChannelPhyParams(MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex, MV_U8 signalAmps,M_U8 pre)*

DESCRIPTION
Modifies the AMP and PRE of a specific SATA channel.

INPUT
pAdapter - Pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - Index to a specific 88SX50xx /88SX60x1 channel.

signalAmps - New value of the AMP (values can be from 0–7).

pre - New value of the PRE (values can be from 0–3).

RETURN
MV_TRUE on success

MV_FALSE on failure

Note

CORE driver modifies the pre-emphasis and amplitude that were saved previously in the MV_SATA_ADAPTER data structure by `mvSataInitAdapter()`. This is done to preserve the new pre-emphasis and amplitude values after each SATA channel reset.

*MV_BOOLEAN mvSataChannelPhyPowerOn(MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex)*

DESCRIPTION
Powers up the physical interface of a specific SATA channel.

INPUT
pAdapter - Pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - Index to a specific 88SX50xx /88SX60x1 channel.

RETURN
MV_TRUE on success

MV_FALSE on failure

Notes

- The physical interfaces of all SATA channels are powered on by default after reset.

*MV_BOOLEAN mvSataChannelPhyShutdown(MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex)*

DESCRIPTION
Shuts down the physical interface of a specific SATA channel.

INPUT
pAdapter - Pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - Index to a specific 88SX50xx /88SX60x1 channel.

RETURN
MV_TRUE on success

MV_FALSE on failure

Note

No disconnect interrupt will be asserted by the 88SX50xx /88SX60x1 after the physical interface is shut down.

*MV_BOOLEAN mvSataChannelFarLoopbackDiagnostic (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex)*

DESCRIPTION
Performs a an external loopback (far end loopback) diagnostic of a specific SATA channel. The diagnostic test can be only performed when a storage device is connected to the SATA channel.

The diagnostic test runs on the specific SATA channel PHY and the SATA PHY in the storage device connected through the SATA cable.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - An index to a specific 88SX50xx /88SX60x1 SATA channel.

RETURN
MV_TRUE on success of diagnostic

MV_FALSE on failure of diagnostic

Note

See the 88SX50xx /88SX60x1 datasheet for an explanation of the external loopback diagnostics.

*MV_BOOLEAN mvSataEnableStaggeredSpinUp (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex)*

DESCRIPTION
Enables SATA channel communication and triggers an OOB sequence on the specific SATA channel.

This function is relevant for adapters that support staggered spinup (for example the 88SX60X1 adapters).

INPUT
pAdapter - Pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - Index to a specific 88SX50xx /88SX60x1 channel.

RETURN
MV_TRUE on success

MV_FALSE on failure

*MV_BOOLEAN mvSataEnableStaggeredSpinUpAll (MV_SATA_ADAPTER *pAdapter)*

DESCRIPTION
Enables SATA channel communication and triggers an OOB sequence on all SATA channels for the specific adapter.

This function is relevant for adapters that support staggered spinup (for example the 88SX60X1 adapters).

INPUT
pAdapter - Pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - Index to a specific 88SX50xx /88SX60x1 channel.

RETURN
MV_TRUE on success

MV_FALSE on failure

Note

The IAL can have a function loop calling `mvSataEnableStaggeredSpinUp()` for all SATA channels on the adapter; but the `mvSataEnableStaggeredSpinUpAll()` function will be faster, since it does the OOB sequence negotiation in parallel for all SATA channels.

*MV_BOOLEAN mvSataDisableStaggeredSpinUp (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex)*

DESCRIPTION
Disables SATA channel communication on a specific SATA channel.

This function is relevant for adapters that support staggered spinup (for example the 88SX60X1 adapters).

INPUT
pAdapter - Pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - Index to a specific 88SX50xx /88SX60x1 channel.

RETURN
MV_TRUE on success

MV_FALSE on failure

*MV_BOOLEAN mvSataDisableStaggeredSpinUpAll (MV_SATA_ADAPTER *pAdapter)*

DESCRIPTION
Disables SATA channel communication on all SATA channels.

This function is relevant for adapters that support staggered spinup (for example the 88SX60X1 adapters).

INPUT
pAdapter - Pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - Index to a specific 88SX50xx /88SX60x1 channel.

RETURN
MV_TRUE on success

MV_FALSE on failure

*MV_BOOLEAN mvSataSetInterfaceSpeed (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex, MV_SATA_IF_SPEED ifSpeed)*

DESCRIPTION
This function sets (and can limit) interface speed negotiation for Gen I (1.5 Gbps) or Gen II (3 Gbps).

This function is relevant for adapters that support Gen I and Gen II (for example the 88SX60X1 adapters).

INPUT
pAdapter - Pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - Index to a specific 88SX50xx /88SX60x1 channel.

ifSpeed - The required setting (or limitation) for the specific SATA channel.

RETURN
MV_TRUE on success

MV_FALSE on failure

MV_SATA_IF_SPEED mvSataGetInterfaceSpeed (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex)

DESCRIPTION

This function returns the currently negotiated interface speed.

This function is relevant for adapters that support Gen I and Gen II (for example the 88SX60X1 adapters).

INPUT

pAdapter - Pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - Index to a specific 88SX50xx /88SX60x1 channel.

RETURN

MV_SATA_IF_SPEED_INVALID - If SATA channel staggered spinup is disabled or if an error occurred on the function parameters.

MV_SATA_IF_SPEED_1_5_GBPS - If currently negotiated interface speed is Gen I.

MV_SATA_IF_SPEED_3_GBPS - If currently negotiated interface speed is Gen II

6.5.3.3 Execute Synchronous Non-UDMA ATA Commands (Polling Driven)

MV_BOOLEAN mvStorageDevATAExecuteNonUDMACommand (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex, MV_U8 PMPort, MV_NON_UDMA_PROTOCOL protocolType, MV_BOOLEAN isEXT, MV_U16_PTR bufPtr, MV_U32 count, MV_U16 features, MV_U16 sectorCount, MV_U16 lbaLow, MV_U16 lbaMid, MV_U16 lbaHigh, MV_U8 device, MV_U8 command);

DESCRIPTION

Performs a user-defined non-UDMA command.

Possible commands must belong to a protocol of either non-data, PIO data-in or PIO data-out ATA command.

INPUT

pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - An index to a specific 88SX50xx /88SX60x1 channel.

PMPort - An index to the port multiplier's destination port (equals 0 if no port multiplier available).

protocolType - Protocol of the requested ATA command to perform.

isEXT - MV_TRUE if the command is an LBA 48-bit extended command.

bufPtr - Pointer to a buffer that the PIO data-out/in ATA command transfers from/to (must be word (16-bit) byte aligned).

count - Number of words to transfer from/to buffer.

features - The value to be written to the FEATURES register.

sectorCount - The value to be written to the SECTOR COUNT register.

lbaLow - The value to be written to the LBA LOW register.

lbaMid - The value to be written to the LBA MID register.

lbaHigh - The value to be written to the LBA HIGH register.

device - The value to be written to the DEVICE register.

command - The value to be written to the COMMAND register.

RETURN
MV_TRUE on success

MV_FALSE on failure

Note

When isEXT is MV_TRUE (extended command) All the 16-bit fields of the MV_U16 parameters to the function are used. Otherwise only the 8 LSBs are used.

*MV_BOOLEAN mvStorageDevATAIdentifyDevice (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex, MV_U8 PMPort, MV_U16_PTR identifyDeviceResult)*

DESCRIPTION
Performs an IDENTIFY DEVICE ATA command to the storage device connected to the SATA channel indexed by channelIndex. The resulting command's data is stored in identifyDeviceResult.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - An index to a specific 88SX50xx /88SX60x1 channel.

PMPort - An index to the port multiplier's destination port (equals 0 if no port multiplier available).

identifyDeviceResult - Holds a pointer to a 512 bytes data buffer that holds the IDENTIFY DEVICE ATA command result.

RETURN
MV_TRUE on success

MV_FALSE on failure

*MV_BOOLEAN mvStorageDevATASetFeatures (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex, MV_U8 PMPort, MV_U8 subCommand, MV_U8 subCommandSpecific1, MV_U8 subCommandSpecific2, MV_U8 subCommandSpecific3, MV_U8 subCommandSpecific4)*

DESCRIPTION
Performs a SET FEATURES ATA command to the storage device connected to the SATA channel indexed by channelIndex.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - An index to a specific 88SX50xx /88SX60x1 channel.

PMPort - An index to port multiplier's destination port (equals 0 if no port multiplier available).

subCommand - Sub-command for the SET FEATURES ATA command.

subCommandSpecific1 - First parameter to the sub-command.

subCommandSpecific2 - Second parameter to the sub-command.

subCommandSpecific3 - Third parameter to the sub-command.

subCommandSpecific4 - Fourth parameter to the sub-command.

RETURN
MV_TRUE on success

MV_FALSE on failure

*MV_BOOLEAN mvStorageDevATAIdleImmediate(MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex)*

DESCRIPTION
Performs the IDLE IMMEDIATE ATA command to the storage device connected to the SATA channel indexed by channelIndex.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - An index to a specific 88SX50xx /88SX60x1 channel.

RETURN
MV_TRUE on success

MV_FALSE on failure

*MV_BOOLEAN mvStorageDevATASoftResetDevice (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex, MV_U8 PMPort, MV_STORAGE_DEVICE_REGISTERS *registerStruct)*

DESCRIPTION
Performs a software reset sequence on the storage device connected to the SATA channel indexed by channelIndex.

The software reset sequence is performed according the software reset sequence defined in the ATA/ATAPI-6 specification

This function waits for the BSY bit to be ’0’, which occurs when FIS 34 is sent from the device to the host upon software reset completion status.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - An index to a specific 88SX50xx /88SX60x1 SATA channel.

PMPort - An index to the port multiplier’s destination port (equals 0 if no port multiplier available).

registerStruct - Holds a pointer to the ATA register data structure, which contains a dump of ATA registers upon completion of software reset protocol. If this parameter equals ’0’, ATA registers are not dumped.

RETURN
MV_TRUE on success

MV_FALSE on failure

*MV_BOOLEAN mvStorageDevATAStartSoftResetDevice (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex, MV_U8 PMPort)*

DESCRIPTION
Performs a software reset sequence on the storage device connected to the SATA channel indexed by channelIndex.

The software reset sequence is performed according the software reset sequence defined in the ATA/ATAPI-6 specification

This function does not poll for software reset completion status. To perform the polling, use the `mvStorageIsDeviceBsyBitOff()` function.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - An index to a specific 88SX50xx /88SX60x1 SATA channel.

PMPort - An index to the port multiplier's destination port (equals 0 if no port multiplier available).

RETURN
MV_TRUE on success

MV_FALSE on failure

*MV_BOOLEAN mvStorageIsDeviceBsyBitOff (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex, MV_STORAGE_DEVICE_REGISTERS *registerStruct)*

DESCRIPTION
Checks if the BSY bit in ATA status is on/off.

If registerStruct is non-zero, then the ATA registers are dumped into the data structure that is pointed to by registerStruct.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - An index to a specific 88SX50xx /88SX60x1 SATA channel.

registerStruct - A pointer to the ATA registers data structure. If non-zero, then upon exit the function dumps the ATA registers to it.

RETURN
MV_TRUE when BSY bit is off

MV_FALSE when BSY bit is on (or on failure)

*MV_BOOLEAN mvStorageDevExecutePIO (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex, MV_U8 PMPort, MV_NON_UDMA_PROTOCOL protocolType, MV_BOOLEAN isEXT, MV_U16_PTR bufPtr, MV_U32 count, MV_STORAGE_DEVICE_REGISTERS *plnATARegs, MV_STORAGE_DEVICE_REGISTERS *pOutATARegs)*

DESCRIPTION
Performs a user-defined non-UDMA command.

Possible commands must belong to a protocol of either non-data, PIO data-in or PIO data-out ATA command.

INPUT

pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - An index to a specific 88SX50xx /88SX60x1 channel.

PMPort - An index to the port multiplier's destination port (equals 0 if no port multiplier available).

protocolType - Protocol of the requested ATA command to perform.

isEXT - MV_TRUE if the command is an LBA 48-bit extended command.

bufPtr - Pointer to a buffer that the PIO data-out/in ATA command transfers from/to (must be word (16-bit) byte aligned).

count - Number of words to transfer from/to buffer.

pInATARegs - ATA registers to be written (includes the command).

pOutATARegs - Holds the result of the PIO command.

RETURN

MV_TRUE on success

MV_FALSE on failure

Note

When isEXT is MV_TRUE (extended command), then all the 16-bit fields of the ATA registers data structure are used. Otherwise only the 8 LSB bits are used.

*MV_BOOLEAN mvStorageDevSetDeviceType (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex, MV_SATA_DEVICE_TYPE deviceType)*

DESCRIPTION

Sets the device type of the storage device connected directly to the adapter's SATA channel.

INPUT

pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - An index to a specific 88SX50xx /88SX60x1 SATA channel.

deviceType - The deviceType connected.

RETURN

MV_TRUE on success

MV_FALSE on failure

Note

This function doesn't query the hardware for the type of storage device, but sets the deviceType field for the appropriate channel's data structure.

MV_SATA_DEVICE_TYPE mvStorageDevGetDeviceType (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex)

DESCRIPTION
Gets the device type of the storage device connected directly to the adapter's SATA channel.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - An index to a specific 88SX50xx /88SX60x1 SATA channel.

RETURN
MV_SATA_DEVICE_TYPE_UNKOWN if no storage device connected (or upon failure).

MV_SATA_DEVICE_TYPE_ATA_DISK if a hard drive is connected.

MV_SATA_DEVICE_TYPE_ATAPI_DISK if an ATAPI device connected.

MV_SATA_DEVICE_TYPE_PM if a port multiplier is connected.

Note

This function doesn't query the hardware for the type of storage device connected, but returns the value previously set by the `mvStorageDevSetDeviceType()` function.

6.5.3.4 Port Multiplier Functions (Polling Driven)

MV_BOOLEAN mvPMDevReadReg (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex, MV_U8 PMPort, MV_U32_PTR pValue, MV_STORAGE_DEVICE_REGISTERS *registerStruct)

DESCRIPTION
Reads from a port multiplier's internal register using PIO non-data protocol.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - An index to a specific 88SX50xx /88SX60x1 SATA channel.

PMPort - Must be 0xF (indicated port multiplier's control port).

pValue - Holds the result of the read (32 bit).

registerStruct - Holds a pointer to the ATA register data structure that contains a dump of ATA registers upon completion of register read. If this parameter equals '0', then ATA registers are not dumped.

RETURN
MV_TRUE on success

MV_FALSE on failure

*MV_BOOLEAN mvPMDevWriteReg (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex, MV_U8 PMPort, MV_U32 value, MV_STORAGE_DEVICE_REGISTERS *registerStruct)*

DESCRIPTION
Writes a value to the port multiplier's internal register using PIO non-data protocol.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - An index to a specific 88SX50xx /88SX60x1 SATA channel.

PMPort - Must be 0xF (indicated port multiplier's control port).

value - Holds the value to be written to the register (32 bit).

registerStruct - Holds a pointer to the ATA register data structure that contains a dump of ATA registers upon completion of register write. If this parameter equals '0', then ATA registers are not dumped.

RETURN
MV_TRUE on success

MV_FALSE on failure

*MV_BOOLEAN mvPMDevEnableStaggeredSpinUp (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex, MV_U8 PMPort)*

DESCRIPTION
Enables SATA channel communication and triggers an OOB sequence on a port multiplier's specific SATA channel.

INPUT
pAdapter - Pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - Index to a specific 88SX50xx /88SX60x1 channel.

PMPort - Holds the required port multiplier's SATA channel number.

RETURN
MV_TRUE on success

MV_FALSE on failure

*MV_BOOLEAN mvPMDevEnableStaggeredSpinUpAll (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex, MV_U8 PMNumOfPorts, MV_U16_PTR bitmask)*

DESCRIPTION
Enables SATA channel communication and triggers an OOB sequence on all port multiplier's SATA channels.

INPUT
pAdapter - Pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - Index to a specific 88SX50xx /88SX60x1 channel.

PMNumOfPorts - Holds the number of device side SATA channels that the port multiplier supports.

bitmask - Pointer to 16-bit data container that holds a bitmask of '1' when the relevant port multiplier's device port staggered spinup operation was successful.

RETURN
MV_TRUE on success

MV_FALSE on failure

6.5.3.5 Queuing Asynchronous ATA Commands

MV_QUEUE_COMMAND_RESULT mvSataQueueCommand(MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex, MV_QUEUE_COMMAND_INFO *pCommandInfo)

DESCRIPTION

Queue UDMA and non-UDMA commands to Core Driver commands queue for a given channel.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - An index to a specific 88SX50xx /88SX60x1 SATA channel.

pCommandInfo - A pointer to an MV_QUEUE_COMMAND_INFO data structure that holds the parameters of the ATA command to add to the commands queue.

RETURN
MV_QUEUE_COMMAND_RESULT_OK - ATA command is queued successfully.

MV_QUEUE_COMMAND_RESULT_QUEUED_MODE_DISABLED - ATA command queueing failed because queuing mode was not enabled. (the API function `mvSataEnableChannelDma` was not called successfully) or this mode was disabled due to an error.)

MV_QUEUE_COMMAND_RESULT_FULL - Command queueing failed because the commands queue is full.

MV_QUEUE_COMMAND_RESULT_BAD_LBA_ADDRESS - Command queueing failed because it tried to queue a 48-bit LBA-feature-set-compliant ATA command on a 28-bit LBA-feature-set-compliant storage device.

MV_QUEUE_COMMAND_RESULT_BAD_PARAMS - UDMA command queueing failed due to bad parameters passed to function.

Note

It is recommended that the IAL assign the value '0' on the MV_QUEUE_COMMAND_INFO data structure (memset operation) before filling in the required parameters.

6.5.4 Interrupt Service Routine

*MV_BOOLEAN mvSataInterruptServiceRoutine (MV_SATA_ADAPTER *pAdapter)*

DESCRIPTION
This function is an interrupt service routine that is called upon receipt of an interrupt from a 88SX50xx / 88SX60x1 adapter.

This routine reads Status registers from the 88SX50xx /88SX60x1 adapter and performs the appropriate interrupt service routine function by calling the `mvSataCommandCompletionCallBack` and `mvSataEventNotify` functions.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

RETURN
MV_TRUE - If there was a real interrupt for the adapter.

MV_FALSE - If there was no real interrupt for the adapter.

*MV_BOOLEAN mvSataMaskAdapterInterrupt (MV_SATA_ADAPTER *pAdapter)*

DESCRIPTION
Masks all interrupts generated from an 88SX50xx /88SX60x1 adapter.

Before masking the interrupts, this functions stores the value of the interrupt mask register in the `interruptMaskReg` field in the pAdapter data structure.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

RETURN
MV_TRUE on success

MV_FALSE on failure

*MV_BOOLEAN mvSataUnmaskAdapterInterrupt (MV_SATA_ADAPTER *pAdapter)*

DESCRIPTION
Unmasks all interrupts generated from an 88SX50xx /88SX60x1 adapter.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

RETURN
MV_TRUE on success

MV_FALSE on failure

MV_BOOLEAN mvSataSetInterruptScheme (MV_SATA_ADAPTER *pAdapter, MV_SATA_INTERRUPT_SCHEME interruptScheme)

DESCRIPTION
Modifies interrupt scheme.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

interruptScheme - A parameter containing the required interrupt scheme.

RETURN
MV_TRUE on success

MV_FALSE on failure

MV_BOOLEAN mvSataCheckPendingInterrupt (MV_SATA_ADAPTER *pAdapter)

DESCRIPTION
Checks if an interrupt is pending. If there is a pending interrupt, then this function masks adapter interrupts and returns MV_TRUE.

This function must be used only when the interrupt scheme is set to MV_SATA_INTERRUPT_IN_TASK.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

RETURN
MV_TRUE if there a pending interrupt

MV_FALSE if there is no pending interrupt

6.6 System-Dependent Header File (mvOs.h)

6.6.1 Types and Defines

6.6.1.1 Data Types (User-Implemented)

MV_VOID - Void

MV_U32 - Unsigned 32-bit

MV_U16 - Unsigned 16-bit

MV_U8 - Unsigned 8-bit

MV_VOID_PTR - Pointer to void

MV_U32_PTR - Pointer to unsigned 32-bit

MV_U16_PTR - Pointer to unsigned 16-bit

MV_U8_PTR - Pointer to unsigned 8-bit

MV_CHAR_PTR - Pointer to string

MV_BUS_ADDR_T - Type that makes it possible for CPU to access PCI addresses

MV_CPU_FLAGS - Type that makes it possible for CORE driver to save and restore CPU flags

6.6.1.2 Defines (User-Implemented)

MV_CPU_WRITE_BUFFER_FLUSH() - Macro for flushing CPU write buffer.

MV_CPU_TO_LE16 (x) - Macro for converting 16-bit from CPU endianess to Little Endian.

MV_CPU_TO_LE32 (x) - Macro for converting 32-bit from CPU endianess to Little Endian.

MV_LE16_TO_CPU (x) - Macro for converting 16-bit from Little Endian to CPU endianess.

MV_LE32_TO_CPU (x) - Macro for converting 32-bit from Little Endian to CPU endianess.

MV_REG_WRITE_BYTE (base, offset, value) - Macro for writing byte to 88SX50xx /88SX60x1 internal register.

MV_REG_WRITE_WORD (base, offset, value) - Macro for writing word (16-bit) to 88SX50xx /88SX60x1 internal
register.

MV_REG_WRITE_DWORD (base, offset, value) - Macro for writing dword (32-bit) to 88SX50xx /88SX60x1 internal
register.

MV_REG_READ_BYTE (base, offset) - Macro for reading a byte from 88SX50xx /88SX60x1 internal register.

MV_REG_READ_WORD (base, offset) - Macro for reading a word (16-bit) from 88SX50xx /88SX60x1 internal register.

MV_REG_READ_DWORD (base, offset) - Macro for reading a dword (32-bit) from 88SX50xx /88SX60x1 internal
register.

6.6.2 Data Structures

6.6.2.1 MV_OS_SEMAPHORE (User-Implemented)

System-dependent implementation defined in the system-dependent header file (mvOs.h).

6.6.2.2 Command Completion and Event Notification (User-Implemented)

MV_BOOLEAN mvSataCommandCompletionCallBack (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex, MV_COMPLETION_TYPE completionType, MV_VOID_PTR *commandID, MV_U16 errorCause, MV_U32 timeStamp, MV_STORAGE_DEVICE_REGISTERS *registerStruct)

DESCRIPTION

This callback function is used by the CORE driver to indicate a command completion event.

INPUT

pAdapter - Pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

channelIndex - An index to a specific 88SX50xx /88SX60x1 channel.

completionType - Equals MV_COMPLETION_TYPE_NORMAL if completion is normal with no errors, MV_COMPLETION_TYPE_ERROR if completion is erroneous, or MV_COMPLETION_TYPE_ABORT if completion is due to an abort request (due to previously called `mvSataFlushDmaQueue` function with flushType equals FLUSH_TYPE_CALLBACK as a parameter to it).

commandID - The command identifier as it is passed to the functions `mvSataQueueUdmaCommand`/ `mvSataQueueCommand` when the command has been issued.

errorCause - Value indicating the Error Cause register in the relevant SATA channel (relevant only for UDMA commands).

timeStamp - The elapsed time it took the hardware to execute the command. (Field is taken from the response queue entry, relevant only for UDMA commands).

registerStruct - If the completion equals MV_COMPLETION_TYPE_ERROR (command completed with errors), this structure holds the register values from the storage device's command block registers.

RETURN
MV_TRUE on success

MV_FALSE on failure

Note

See the 88SX50xx /88SX60x1 datasheet for further information on a specific SATA channel Error Cause register.

*MV_BOOLEAN mvSataEventNotify (MV_SATA_ADAPTER *pAdapter, MV_EVENT_TYPE eventType, MV_U32 param1, MV_U32 param2)*

DESCRIPTION
This callback function is called when the CORE driver needs to notify the IAL of a specific event.

INPUT
pAdapter - A pointer to an MV_SATA_ADAPTER data structure that holds all information to access the 88SX50xx /88SX60x1 adapter.

eventType - Equals MV_EVENT_TYPE_ADPT_ERR in the case of an error in the 88SX50xx /88SX60x1 adapter or equals MV_EVENT_TYPE_SATA_CABLE in the case of a disconnect/connect of a storage device to a SATA channel.

param1 - If eventType is MV_EVENT_TYPE_ADPT_ERR, this parameter holds the PCI Cause register read from the 88SX50xx /88SX60x1 adapter. If eventType is MV_EVENT_TYPE_SATA_CABLE, this parameter equals '0' in the case of a disconnect event notification, or equals '1' in the case of a connect event notification.

param2 - If eventType is MV_EVENT_TYPE_ADPT_ERR, this parameter has no use. In the case of eventType equals MV_EVENT_TYPE_SATA_CABLE, this parameter holds the index of the SATA channel on which the connect/disconnect event.

RETURN
MV_TRUE on success

MV_FALSE on failure

6.6.2.3 System Routines (User-implemented)

MV_BOOLEAN mvOsSemInit (MV_OS_SEMAPHORE *semaphore)

DESCRIPTION
Initializes a semaphore.

INPUT
semaphore - A pointer to an MV_OS_SEMAPHORE data structure that holds semaphore information to be initialized.

RETURN
MV_TRUE on success

MV_FALSE on failure

MV_BOOLEAN mvOsSemTake (MV_OS_SEMAPHORE *semaphore)

DESCRIPTION
Locks a semaphore.

INPUT
semaphore - A pointer to an MV_OS_SEMAPHORE data structure.

RETURN
MV_TRUE on success

MV_FALSE on failure

MV_BOOLEAN mvOsSemRelease (MV_OS_SEMAPHORE *semaphore)

DESCRIPTION
Unlocks a semaphore.

INPUT
semaphore - A pointer to an MV_OS_SEMAPHORE data structure.

RETURN
MV_TRUE on success

MV_FALSE on failure

void mvMicroSecondsDelay (MV_SATA_ADAPTER *pAdapter, MV_U32 delay)

DESCRIPTION
Delay function in micro-seconds resolution.

INPUT
pAdapter - A pointer to the adapter's data structure.

delay - Number of micro-seconds to delay.

RETURN
N/A

6.6.2.4 Logger API

MV_BOOLEAN mvLogRegisterModule(MV_U8 moduleId, MV_U8 filterMask, char* name);

DESCRIPTION
This functions registers the module with the logger. Each module in the driver must provide a unique module identifier for the registration. It is recommended to call the function prior to module initialization, otherwise the log messages issued prior to calling this function won't be seen.

INPUT
moduleId - The module identifier.

filterMask - The filter mask for logging. See "Log Levels Filter Mask" on page 51

name- Pointer to the module name. The module name string is not copied by the logger so the caller ***must not*** free the memory containing the string.

RETURN
MV_TRUE on success

MV_FALSE on failure

MV_BOOLEAN mvLogSetModuleFilter(MV_U8 moduleId, MV_U8 filterMask)

DESCRIPTION
This function defines the new log filter for the registered module.

INPUT
moduleId - The module identifier. See "Log Levels Filter Mask" on page 51.

filterMask - The new filter mask for logging.

RETURN
MV_TRUE on success

MV_FALSE on failure

MV_U8 mvLogGetModuleFilter(MV_U8 moduleId)

DESCRIPTION
This function returns log filter settings for the registered module.

INPUT
moduleId - The module identifier.

RETURN
Module logging filter value. For unregistered module, this function returns 0.

void mvLogMsg(MV_U8 moduleId, MV_U8 type, char* format,...)

DESCRIPTION
This function prints the log message of a specified type and format. The message is printed only if the module is registered in the logger and the message type matches the log filter settings for the current module.

INPUT
moduleId - The module identifier.

type - Log message type (see "Log Message Type" on page 51)

format - Formatted string

RETURN
None

6.6.2.5 Interrupt Coalescing in I/O Granularity API

MV_BOOLEAN mvSataEnableIoGranularity (MV_SATA_ADAPTER* pAdapter, MV_BOOLEAN enable)

DESCRIPTION
Enables and disables interrupt coalescing in I/O granularity for the specific SATA adapter. If it is enabled, the function masks all channel interrupts and enables I/O granularity coalescing interrupts.

INPUT
pAdapter - Pointer to the adapter data structure.

enable - MV_TRUE to enable interrupt coalescing in I/O granularity, MV_FALSE to disable interrupt coalescing in I/O granularity

RETURN
MV_TRUE on success

MV_FALSE on failure

6.6.2.6 Channel-to-Channel Communication Mode Functions

MV_BOOLEAN mvSataC2CInit(MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex, MV_SATA_C2C_MODE mvSataC2CMode, MV_VOID_PTR mvSataC2CCallBack)

DESCRIPTION
Initializes SATA channel for Target Mode communication.

INPUT
pAdapter - pointer to the adapter data structure.

channelIndex - index of the specific SATA channel.

mvSataC2CMode - the channel role in Target Mode communication: target or initiator.

mvSataC2CCallBack - callback function to call on target mode communication event.

RETURN
MV_TRUE on success

MV_FALSE on failure

MV_BOOLEAN mvSataC2CStop (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex)

DESCRIPTION
Disables target mode for SATA channel.

INPUT
pAdapter - pointer to the adapter data structure.

channelIndex - the index of the specific SATA channel

RETURN
MV_TRUE on success

MV_FALSE on failure

MV_BOOLEAN mvSataC2CSendRegisterDeviceToHostFIS (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex, MV_U8 pmPort, MV_BOOLEAN bInterrupt, MV_U8 msg[MV_C2C_MESSAGE_SIZE])

DESCRIPTION
Sends Register Device to Host FIS on specific SATA channel.

INPUT
pAdapter - pointer to the adapter data structure.

channelIndex - the index of the specific SATA channel.

pmPort - Port multiplier port number.

bInterrupt - determines whether to generate the interrupt on the receiver side.

msg - message containing 10 bytes of user data, which is reflected in ATA register on the receiver channel.

RETURN
MV_TRUE on success

MV_FALSE on failure

MV_BOOLEAN mvSataC2CActivateBmDma (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex, MV_U8 pmPort, MV_U32 prdTableHigh, MV_U32 prdTableLow, MV_UDMA_TYPE dmaType)

DESCRIPTION
Activates Bus Master DMA for the specific SATA channel.

INPUT
pAdapter - pointer to the adapter data structure.

channelIndex - the index of the specific SATA channel.

pmPort - Port Multiplier port number.

prdTableHigh - the upper 32-bit of PRD table address.

prdTableLow - the lower 32-bit of PRD table address.

dmaType - DMA operation type (read or write).

RETURN
MV_TRUE on success

MV_FALSE on failure

MV_BOOLEAN mvSataC2CResetBmDma (MV_SATA_ADAPTER *pAdapter, MV_U8 channelIndex)

DESCRIPTION

Resets Bus-Master DMA for the specific SATA channel.

INPUT

pAdapter - pointer to the adapter data structure.

channelIndex - the index of the specific SATA channel.

RETURN

MV_TRUE on success

MV_FALSE on failure

typedef MV_BOOLEAN (*C2CCallBack_t)(struct mvSataAdapter * pAdapter, struct mvSataChannel * pChannel, MV_C2C_EVENT_TYPE event, MV_U32 msgSize, MV_U8* msg);

DESCRIPTION

This is a user-implemented callback function, which is executed upon target mode Register device to host FIS reception, Bus Master DMA transfer completion, or communication error.

INPUT

pAdapter - pointer to the adapter data structure.

pChannel - pointer to the SATA channel data structure.

event - target mode communication event which could be one of the following:

- MV_C2C_REGISTER_DEVICE_TO_HOST_FIS_DONE: Register device to host FIS has been successfully received.
- MV_C2C_REGISTER_DEVICE_TO_HOST_FIS_ERROR: Register device to host FIS transfer error.
- MV_C2C_BM_DMA_DONE: Bus Master DMA transaction succeeded.
- MV_C2C_BM_DMA_ERROR: Bus Master DMA data transfer error succeeded.

msgSize - message buffer size. If message buffer is unavailable, equals ' 0'.

msg - message buffer which contains either 10 bytes user data in the case of MV_C2C_REGISTER_DEVICE_TO_HOST_FIS, otherwise equal to NULL.

RETURN

MV_TRUE on success

MV_FALSE on failure

Section 7. SCSI to ATA Translation Layer

7.1 Introduction

The SCSI to ATA Translation Layer (SAL) driver is a software layer that is operating system and architecture independent. The functionality of this layer is to translate SCSI commands into ATA commands and vice versa. It can be used as a sub-component of an OS-specific driver that is layered under the OS SCSI sub-system.

7.2 Architecture

This layer is built on the CORE driver layer and the Common Intermediate Application layer. From the upper side, it provides an entry point that handles SCSI commands. The upper IAL components are responsible for the adapter management, which involves initializing the adapter, SATA channels, SATA drives, handling hot-plug events etc. Also, several OS services are needed by this layer mainly for messages logging. These services are provided by the OS layer that is integrated with the CORE driver.

7.3 SAL API Summary

`mvSataScsiInitAdapterExt()`	Initializes the SAL adapter data structure extension.
`mvSataExecuteScsiCommand()`	Executes SCSI commands.
`mvSataScsiPostIntService()`	Handles split commands.
`mvSataScsiSetDriveReady()`	Notifies the SAL about connected/disconnected drives.
`mvSataScsiNotifyUA()`	Notifies the SAL about Unit Attention conditions.

7.4 SAL SCSI Characteristics

7.4.1 Implementation Standards

The SAL complies with the following SCSI standards:

1. SCSI-3 Architecture Model (X3.270-199x) (SAM).
2. SCSI-3 Primary Commands (X3.301 - 199x) (SPC).
3. SCSI-3 Block Commands (ANSI NCITS 306-199X)(SBC).

7.4.2 Device Addressing

The following SCSI to ATA device addressing translation scheme holds:

- SATA channel translated to SCSI bus.

- Port multiplier device port translated to SCSI target.
- SATA drive contains one lun (lun 0).

7.4.3 SCSI Features

7.4.3.1 Unit Attention Condition Reporting

The following events reported separately in the following order:

1. Bus reset occurred - Additional sense 29h and additional sense qualifier 2h.
2. Parameters changed - Additional sense 2Ah and additional sense qualifier 1h.

upon power-on or hardware reset the IAL notifies the SAL about unit attention condition, the SAL in turn report the first event for the first command it receives, and the second event is reported for the consecutive command (unless the command is INQUIRY or REQUEST SENSE, see SAM for details).

7.4.3.2 Auto-Sense

when command completes with CHECK CONDITION status the SAL reports the sense data automatically, actually this the only mode supported for reporting sense data, the SAL doesn't store this data for future reporting using the request sense command

7.4.3.3 Parameters Mode Pages

The following mode pages is supported:

1. Caching mode page - write cache enable and disable read-ahead fields supported, values types supported: current values, changeable and default.
2. Control mode page - Qerr and unrestricted reordering allowed fields supported, values types supported: current and default.

7.4.3.4 Relative Addressing

Relative addressing is not supported.

7.4.4 Supported SCSI Commands

Table 2: Supported SCSI Commands

Command	SCSI Operation Code
READ(6)	08h
READ(10)	28h
WRITE(6)	0Ah
WRITE(10)	2Ah
INQUIRY	12h
TEST UNIT READY	00h
MODE SELECT(6)	15h
MODE SENSE(6)	1Ah

Table 2: Supported SCSI Commands (Continued)

Command	SCSI Operation Code
READ CAPACITY(10)	25h
REQUEST SENSE(6)	03h
VERIFY(6)	13h
VERIFY(10)	2fh
SYNCHRONIZE CACHE(10)	35h
SEEK(10)	2Bh
REASSIGN BLOCKS	07h
WRITE LONG(10)	3Fh

7.5 Internal Implementation

7.5.1 Command Splitting

In some cases a SCSI command is translated into several ATA commands, e.g., when a VERIFY (10) command with 512 sectors is received for a drive that doesn't support the ATA command READ VERIFY SECTORS EXT, it is translated into READ VERIFY SECTORS. But this ATA command is limited to 256 sectors only, so the solution is to send two commands, each with 256 sectors. When command splitting is required, the SAL sends these ATA commands one by one in a serial manner, i.e., one command is sent, and when it has been completed by the CORE drive, the next command is sent.

7.5.2 Control Synchronization

The SAL doesn't implement any synchronization mechanism to protect its internal data. The IAL is responsible for doing this. The SAL API functions are not re-entrant and must not be called simultaneously unless they are called for different adapters.

7.5.3 Buffer Synchronization

A typical SCSI command involves two buffers—data buffer and sense data buffer. The sense data buffer is modified by the SAL for all SCSI commands. The data buffer is also modified by the SAL for all SCSI commands except the READ(6), READ(10), WRITE(6), and WRITE(10) SCSI commands. For these commands, data buffers are always accessed by the hardware DMA. The IAL must take this into consideration when synchronizing buffers between the CPU's cache and the system memory.

7.6 SCSI to ATA Commands Translation Table

Table 3: SCSI to ATA Commands Translation

SCSI COMMAND	SCSI OPCODE	ATA COMMAND(S)	NOTES
READ(6) READ(10)	08h 28h	READ DMA or READ DMA EXT if the ATA drive supports lba48 addressing feature set.	CORE drive decides which ATA READ version to send—FPDMA, QUEUED, or regular READ DMA—depending on the EDMA configuration.
WRITE(6) WRITE(10)	0Ah 2Ah	WRITE DMA or WRITE DMA EXT if the ATA drive supports lba48 addressing feature set.	CORE drive decides which ATA WRITE version to send—FPDMA, QUEUED, or regular WRITE DMA—depending on the EDMA configuration.
INQUIRY	12h	No commands	Returns information based on ATA IDENTIFY data cached by the SAL.
TEST UNIT READY	00h	No commands	
MODE SELECT(6)	15h	ATA SET FEATURES	May be split into multiple commands.
MODE SENSE(6)	1Ah	IDENTIFY DEVICE	
READ CAPACITY(10)	25h	No commands	Returns information based on ATA IDENTIFY data cached by the SAL.
REQUEST SENSE(6)	03h	No commands	
VERIFY(6) VERIFY(10)	13h 3Fh	READ VERIFY SECTORS or READ VERIFY SECTORS EXT if the ATA drive supports lba48 addressing feature set.	VERIFY(10) may be split into multiple READ VERIFY SECTORS.
SYNCHRONIZE CACHE(10)	35h	FLUSH CACHE	
SEEK(10)	2Bh	No Commands	
REASSIGN BLOCKS	07h	No Commands	
WRITE LONG(10)	3Fh	WRITE LONG	

7.7 ATA to SCSI Error Translation

When an ATA command is completed with an ATA error, the SAL translates this error into a SCSI error by completing the SCSI command with MV_SCSI_COMPLETION_ATA_FAILED and setting ScsiStatus to CHECK condition, then setting the sense buffer with values according to the ATA error type. The ATA drive reports the type of error by setting a corresponding bit in the ATA Error register when the command is completed.

Table 4: ATA to SCSI Error Translation

ATA error code Abbreviation	ATA Error Code Name	Bit in the ATA Error Register	SCSI Translation
NM	No Media	1	If the ATA command is READ VERIFY SECTORS(EXT) set the sense key to Unit Attention. Else if the ATA command is DMA command (READ/WRITE DMA [QUEUED][EXT]..) set the sense key to Unit Attention, and set the Additional Sense Code to 3Ah (No Media in Device).
MC	Media Changed	5	If the ATA command is READ VERIFY SECTORS(EXT) set the sense key to Unit Attention. Else if the ATA command is DMA command (READ/WRITE DMA [QUEUED][EXT]..) set the sense key to Unit Attention, and set the Additional Sense Code to 3Ah (No Media in Device).
MCR	Media Change Request	3	If the ATA command is READ VERIFY SECTORS(EXT) set the sense key to Unit Attention. Else if the ATA command is DMA command (READ/WRITE DMA [QUEUED][EXT]..) set the sense key to Unit Attention, and set the Additional Sense Code to 3Ah (No Media in Device).
ABRT	Command Aborted	2	If the ATA command is READ VERIFY SECTORS(EXT) or SET FEATURES set the sense key to Aborted Command and set the Additional Sense Code to No Sense Code. Else if the ATA command is DMA command (READ/WRITE DMA [QUEUED][EXT]..) then check the IDNF. If IDNF is not set, then set the SCSI sense key to ILLEGAL REQUEST and set the Additional Sense Code to ILLEGAL BLOCK. If the IDNF is also set, then set the sense key to Aborted Command and set the Additional Sense Core to No Sense.
IDNF	Address could not be found	4	If the ATA command is READ VERIFY SECTORS(EXT) set the sense key to Aborted Command and set the Additional Sense Code to No Sense Code. Else if the ATA command is DMA command (READ/WRITE DMA [QUEUED][EXT]..) then check the ABRT. If ABRT is not set, then set the SCSI sense key to ILLEGAL REQUEST and set the Additional Sense Code to ILLEGAL BLOCK. If the ABRT is also set, then set the sense key to Aborted Command and set the Additional Sense Core to No Sense.
UNC	Uncorrectable data	6	If the ATA command is READ VERIFY SECTORS(EXT) or READ DMA, set the sense key to Medium Error, set the Valid bit to 1, and set the Information bytes with the LBA Address in the Sense buffer of the relevant SCSI command. Else if the ATA command is Write DMA command (WRITE DMA [QUEUED][EXT]..), this error is called **WP(Write Protect).** Set the sense key to Data Protect and set the Additional Sense Code set to No Sense.

Table 4: ATA to SCSI Error Translation (Continued)

ATA error code Abbreviation	ATA Error Code Name	Bit in the ATA Error Register	SCSI Translation
ICRC	CRC error during transfer	6	If the ATA command is DMA command (READ/WRITE DMA [QUEUED][EXT]..) set the sense key to Aborted Command and set the Additional Sense Code to No Sense Code.

Note

If the ATA error received doesn't match the above-mentioned cases, set the sense key to **Aborted Command** and set the Additional Sense Code to **No Sense Code**.

7.8 SAL Integration

To integrate the SAL with the IAL, perform the following steps:

1. Register the SAL's logging module (see CORE driver logging module).
2. Allocate the MV_SAL_ADAPTER_EXTENSION structure per adapter. This structure is used by the SAL to hold its context information for a given adapter.
3. Call `mvSataScsiInitAdapterExt()` to initialize the SAL's data structure for a given adapter.
4. Hook the function `mvSataScsiPostIntService()` right after calling the CORE driver `mvSataInterruptServiceRoutine()` function.
5. For each drive, the MV_SAL_ADAPTER_EXTENSION structure contains ATA information gathered from the ATA Identify Device Command data. Before notifying the SAL that a drive is ready, the IAL should fill in this information for each drive present. The data prototype for this information is defined in the Common Intermediate Application Layer.
6. Call `mvSataScsiSetDriveReady()` to notify the SAL that a given drive is available and ready to receive ATA commands or to notify which drive(s) are not available.
7. Call `mvSataScsiNotifyUA()` to notify the SAL that the SCSI Unit Attention condition is pending for a given drive. This function should be called after performing power-on or a hardware reset to the drive.
8. Call `mvSataExecuteScsiCommand()` to queue SCSI commands.

7.9 SAL API

7.9.1 Enumerators

MV_SCSI_COMPLETION_TYPE - This enumerator defines the values used to describe how the SCSI command **is completed** by the SAL. For some values the SAL also sets the SCSI command status.

Table 5: Enumerators

Completion Status	Description	SCSI Status
INVALID_STATUS	Initial value set by SAL. Should not be returned unless unexpected error occurred.	N/A
SUCCESS	Command completed successfully with no errors.	GOOD
BAD_SCB	Bad SCSI Command Block. Something is wrong with the parameters passed in this structure (e.g, the data length to transfer doesn't match the number of sectors).	N/A
BAD_SCSI_COMMAND	Bad parameter in the SCSI CDB.	CHECK CONDITION
ATA FAILED	Translated ATA command completer with error.	CHECK CONDITION
QUEUE_FULL	No place in the CORE drive commands queue for the translated command.	CHECK CONDITION
NOT READY	CORE drive not ready for queuing ATA commands.	N/A
ABORTED	Command was aborted by the CORE driver.	N/A
OVERRUN	Returned data less than the data buffer length.	GOOD/CHECK CONDITION
UNDERRUN	Returned data more than the data buffer length.	GOOD/CHECK CONDITION
PARITY_ERROR	Command failed due to parity error.	CHECK CONDITION
DISCONNECT	Drive was disconnected while processing the command.	N/A
NO DEVICE	Targeted device not available.	N/A
INVALID BUS	SCSI bus ID not valid.	N/A
BUS_RESET	Not Used.	N/A
BUSY	Not Used.	N/A
UA_RESET	Command failed with unit attention condition due to bus reset.	CHECK CONDITION
UA_PARAMS_CHANGED	Command failed with unit attention condition and parameters changed.	CHECK CONDITION

MV_SCSI_COMMAND_STATUS_TYPE - Returned by `mvSataExecuteScsiCommand()` to describe the flow of the SCSI command. The values of this enumerator are:

Table 6: Enumerator Values

COMMAND STATUS	Description
COMPLETED	SCSI command was completed when it was processed by `mvSataExecuteScsiCommand()`, and the completion callback function was called.
QUEUED	Translated to ATA command(s) that are queued in the CORE driver's commands queue.
FAILED	`mvSataExecuteScsiCommand()` failed to handle this command due to unexpected error.
QUEUED_BY_IAL	Not used by SAL.

7.9.2 Data Structures

7.9.2.1 MV_SATA_SCSI_CMD_BLOCK

This structure contains the input/output and context information of SCS command to be processed by `mvSataExecuteScsiCommand()`.

Description of Fields:

IN MV_U8* **ScsiCdb -** SCSI command data block buffer

IN MV_U32 **ScsiCdbLength** - Length in bytes of the CDB (6,10,12,16)

IN MV_U8 **bus -** SCSI bus

IN MV_U8 **target** - Target device ID

IN MV_U8 **lun** - SCSI lun number of the device

IN MV_BOOLEAN useSingleBuffer - True when the data located in the buffer pointed by pDataBuffer (virtual address). False when the command is READ/WRITE. In this case the dates located in a PRD table.

IN MV_U8 ***pDataBuffer** - Pointer to the command data buffer

IN MV_U32 **dataBufferLength** - Length in bytes of the command data buffer

IN MV_U32 **PRDTableEntries** - Number of entries in the PRD table

IN MV_U32 **PRDTableLowPhyAddress** - Low 32 bits of the PRD table physical address

IN MV_U32 **PRDTableHighPhyAddress** - High 32 bits of the PRD table physical address

OUT MV_U8 **ScsiStatus** - SCSI status will be written to this field

IN MV_U8* **pSenseBuffer** - Pointer to the SCSI sense buffer

IN MV_U32 **senseBufferLength** - Length in bytes of the SCSI sense buffer

OUT MV_U32 **senseDataLength** - Length in bytes of the generated sense data

OUT MV_U32 **dataTransfered** - Length in bytes of the data transferred to the data buffer/s

OUT MV_SCSI_COMPLETION_TYPE ScsiCommandCompletion - Translation layer status of the completed SCSI command callback function called by the translation layer when the SCSI completed.

IN mvScsiCommandCompletionCallBack completionCallBack - Callback function to be invoked when the command is completed.

IN struct mvSalAdapterExtension * pSalAdapterExtension - Pointer to the SAL extension of the adapter.

IN struct mvIALCommonAdapterExtension* pIalAdapterExtension - Not used by SAL

MV_VOID_PTR IALData - This field is for the IAL use only.

The following are fields for internal use by the translation layer:

MV_UDMA_TYPE udmaType

MV_QUEUED_COMMAND_TYPE commandType - Used for sense buffer

MV_U32 LowLbaAddress - Used for non-UDMA and for sense buffer

MV_BOOLEAN isExtended

MV_U16 splitCount

MV_U16 sequenceNumber - Used to create a list for commands that need post interrupt service

struct _mvSataScsiCmdBlock *pNext

MV_STORAGE_DEVICE_REGISTERS ATAregStruct

7.9.2.2 MV_SATA_SCSI_CHANNEL_STATS

This structure is used for collecting statistics of the I/Os sent by the SAL.

The fields of this structure are:

MV_U32 totalIOs

MV_U32 totalAccumulatedOutstanding

MV_U32 totalSectorsTransferred

7.9.2.3 MV_SATA_SCSI_DRIVE_DATA

This structure contains the information used by the SAL per SATA drive fields:

MV_BOOLEAN driveReady - Indicates that the drive is ready for receiving ATA commands

ATA_IDENTIFY_INFO identifyInfo - This structure contains information based on the IDENTIFY DATA that should be set by the IAL

MV_U16_PTR identifyBuffer - Pointer to the buffer to which to write IDENTIFY data

MV_SATA_SCSI_CHANNEL_STATS stats - I/Os statistics

MV_BOOLEAN UAConditionPending - If Unit Attention Condition is pending

MV_U32 UAEvents - Unit Attention events to report

7.9.2.4 MV_SAL_ADAPTER_EXTENSION

This structure contains the information used by the SAL for a given adapter.

The fields of this structure are:

MV_SATA_ADAPTER *pSataAdapter - Pointer to the CORE driver data structure

MV_SATA_SCSI_CMD_BLOCK *pHead - Head of a linked list of the commands that need service when `mvSataScsiPostIntService()` is invoked

MV_SATA_SCSI_DRIVE_DATA ataDriveData [MV_SATA_CHANNELS_NUM][MV_SATA_PM_MAX_PORTS] - SATA drive information for each drive

MV_U16 identifyBuffer[MV_SATA_CHANNELS_NUM][MV_ATA_IDENTIFY_DEV_DATA_LENGTH] - Data buffer for the IDENTIFY command. One buffer is used for each channel, so multiple devices on the same channel share this buffer.

7.9.3 API Functions

MV_VOID mvSataScsiInitAdapterExt(MV_SAL_ADAPTER_EXTENSION *pAdapterExt,MV_SATA_ADAPTER* pSataAdapter)

DESCRIPTION
Initializes the SAL data structure.

INPUT
pAdapterExt - Pointer to SAL structure extension allocated for a given adapter.

pSataAdapter - Pointer to MV_SATA_ADAPTER data structure, which holds information to access the 88SX50xx /88SX60x1 device.

MV_VOID mvSataScsiPostIntService(MV_SAL_ADAPTER_EXTENSION *pAdapterExt)

DESCRIPTION
This function should be called after calling the CORE driver ISR. When the SCSI command is split into multiple ATA commands, this function sends the next ATA command when one has been completed.

INPUT
pAdapterExt - Pointer to SAL structure extension allocated for given adapter.

MV_SCSI_COMMAND_STATUS_TYPE mvSataExecuteScsiCommand(MV_SATA_SCSI_CMD_BLOCK *pMvSataScsiCmdBlock)

DESCRIPTION
This function handles execution of a given SCSI command.

INPUT
pMvSataCmdBlockAdapterExt - Pointer to SCSI COMMAND BLOCK structure, which contains the information of the SCSI command to translate.

RETURN
MV_SCSI_COMMAND_STATUS_TYPE - How the command was processed

MV_VOID mvSataScsiSetDriveReady(MV_SAL_ADAPTER_EXTENSION *pAdapterExt, MV_U8 channelIndex, MV_U8 PMPort, MV_BOOLEAN isReady)

DESCRIPTION
Informs the SAL that SATA drive(s) are ready/not ready.

INPUT
pAdapterExt - Pointer to SAL structure extension allocated for a given adapter.

channelIndex - Channel number where the drive is connected.

PMPort - Port number where the drive is connected. When isReady is MV_FALSE a failure of FFh indicates that all the drives on the given channel are not ready.

isReady - If equals MV_TRUE then the drive is ready. When MV_FALSE then drive(s) are not ready.

MV_VOID mvSataScsiNotifyUA(MV_SAL_ADAPTER_EXTENSION *pAdapterExt, MV_U8 channelIndex, MV_U8 PMPort)

DESCRIPTION
Informs the SAL that SATA drives should have Unit Attention due to power-on or hardware reset.

INPUT
pAdapterExt - Pointer to SAL structure extension allocated for a given adapter.

channelIndex - Channel number where the drive is connected.

PMPort - Port number where the drive is connected.

Section 8. IAL Common Layer

8.1 Introduction

The Common IAL component is a software package which is operating system and architecture -independent. It has two main functionalities—it provides the set of helper API functions to the system-dependent IAL layer and manages state machines of the adapter and SATA channel.

The Common IAL component API and data structures are divided into three categories:

- **Common IAL implemented helper API and data structures:** Includes the helper routines for the system-dependent IAL.
- **Common IAL implemented state machine API and data structures:** Includes the functions and data structures used in the adapter and channels' state machine.
- **Common IAL User-implemented API:** Includes several functions and a single data structure which must be implemented by the user in the system-dependent IAL layer.

The Common IAL layer provides the following functionality:

- Triggers SATA adapter initialization sequence through the Core Driver API.
- Manages a state machine for the adapter and its channels to provide an asynchronous SATA adapter and SATA channels initialization process that is completely transparent to OS.
- Interacts with SAL to provide the representation of SATA connected equipment in SAL.
- Interacts with system-dependent IAL layer to represent the SATA adapters and SATA equipment connected to their channel to user application or OS SCSI subsystem.
- Notifies system-dependent IAL about changes in the status of SATA connected equipment and serves as a wrapper between system-dependent IAL and SAL component.
- Access to TWSI devices connected to the 88SX60X1 adapter TWSI bus. (Relevant only for certain revisions of 88SX60X1 adapters).

This document is divided into the following sections:

- Common IAL basic design and integration guidelines: Describes the basic design guidelines for Common IAL.
- Common IAL state machine for adapter and channels: Describes a Common IAL state machine for the adapter and its channels.
- Common IAL API Summary: Categorizes the Common IAL data structure and API being used.

8.2 Common IAL Basic Design and Integration Guidelines

The purpose of this section is to describe basic design and integration guidelines for users who integrate part or all of the Common IAL into their software drivers.

8.2.1 Initialization Latency and State Machines

Serial ATA devices such as disk drives and Port Multipliers may not initialize immediately. Usually Port Multipliers initialize faster than disk drives, since they don't have any mechanical parts. It usually takes few seconds for a disk drive to initialize, until it reports about the device's malfunctions. On other hand, operating systems such as Linux

and Windows require the device driver to avoid any delays longer than ~10 milliseconds, since longer delays affect OS system performance and smoothness, and occasionally may cause certain management processes to timeout.

To resolve this issue, the Common IAL component provides a mechanism of asynchronous SATA adapter and SATA channels initialization that is completely transparent to the OS. The initialization sequence is state-machine-based and timer-driven, so channels initialization is done in the background and enables the other OS components to continue running.

8.2.2 System Timer

The state machines of the adapter and channel are timer-driven, thus every adapter instance requires a single, periodic, low-resolution OS timer. The timer period default value must be set to 0.5 seconds, unless modified by the user.

When a channel state requires a longer timeout period, timer events are accumulated until the timer expiration threshold is reached.

8.2.3 Common IAL Software Command Queue

The Common IAL optionally maintains a SCSI command software queue, which contains the SCSI commands submitted by the OS before channel initialization has been completed.

When channel initialization has been completed, all SCSI commands in the queue are returned to the OS with BUSY status, which causes the OS to retry the commands.

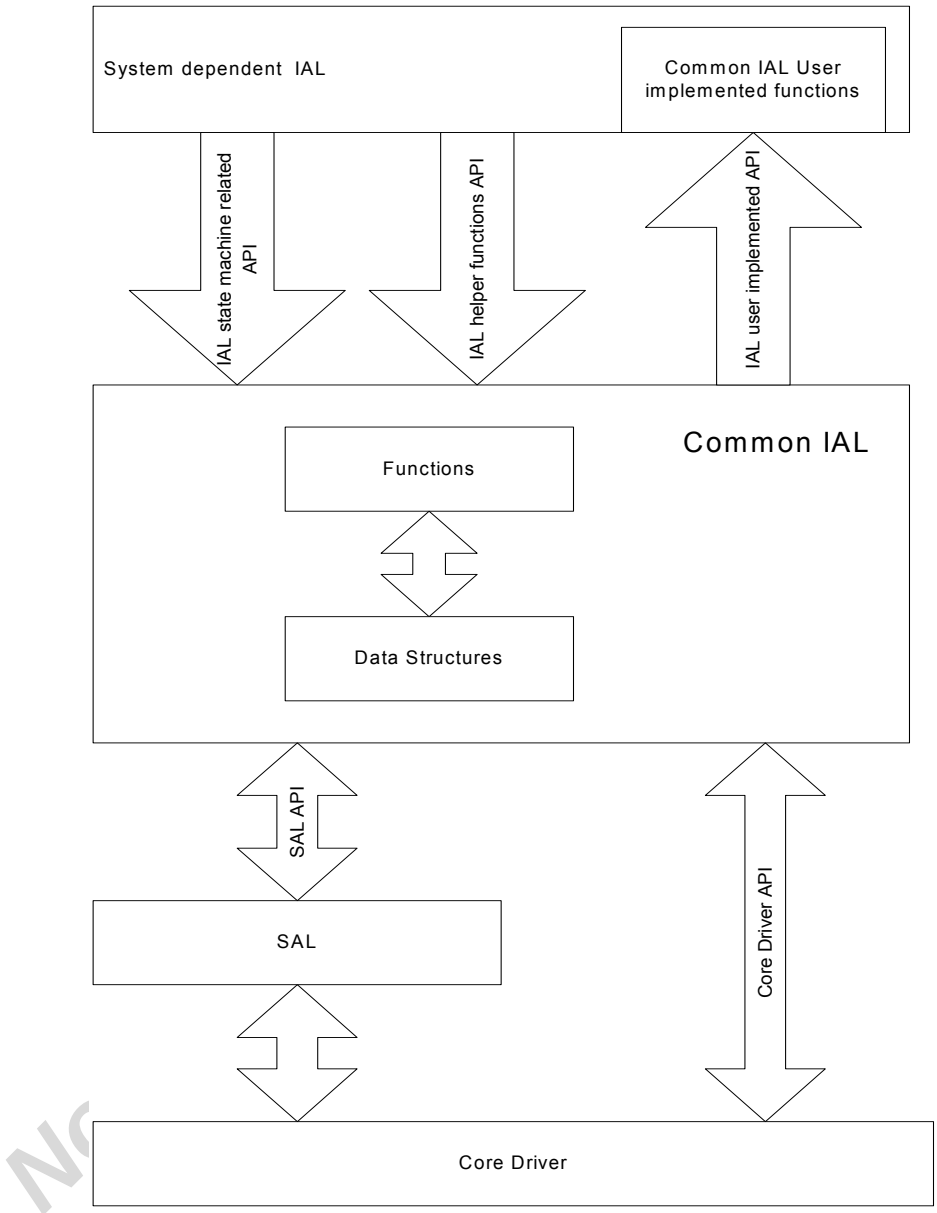
Note

For Windows OS the command queue is not required. In this case when channel initialization sequence has not completed, the SCSI command is completed immediately and returned to OS with BUSY status.

8.3 Common IAL Function API and Data Structures Summary

The following two sub-sections summarize the CORE driver API and data structures, which are categorized in groups according to their functionality.

Figure 6: Common IAL API and Data Structures Block Diagram



8.3.1 API Summary of Common IAL Functions

Common IAL helper functions

mvParseIdentifyResult	Parses IDENTIFY command result.
mvGetSataDeviceType	Determines SATA device type.
mvInitSataDisk	Initializes SATA disk drive.
mvGetPMDeviceInfo	Retrieves Port Multiplier information.

Common IAL state-machine-related functions

mvAdapterStartInitialization	Begins adapter and state machine initialization.
mvRestartChannel	Restarts SATA channel.
mvStopChannel	Stops SATA channel.
mvPMHotPlugDetected	Notifies Common IAL about PM hot plug.
mvIALTimerCallback	Common IAL system timer callback.
mvCommandCompletionErrorHandler	Notifies Common IAL about command completion error.
mvExecuteScsiCommand	Wrapper for SAL mvSataExecuteScsiCommand.

Common IAL user-supplied routines

IALInitChannel	Allocates and initializes SATA channel data structures.
IALReleaseChannel	Frees SATA channel data structure.
IALBusChangeNotify	Notifies OS about changes in channel status.
IALConfigQueuingMode	Configures SATA channel EDMA queuing mode.

Common IAL TWSI devices access

mvSataTWSIMasterInit	Initializes 88SX60X1 TWSI master.
mvSataTWSIMasterReadByte	Reads a byte from a TWSI slave connected to the 88SX60X1 TWSI bus.
mvSataTWSIMasterWriteByte	Writes a byte to a TWSI slave connected to the 88SX60X1 TWSI bus.

8.3.2 Common IAL Data Structure Summary

Data structures modified by IAL and CORE drive:

MV_IAL_COMMON_CHANNEL_EXTENSION	Data structure presenting Common IAL channel related data.
MV_IAL_COMMON_ADAPTER_EXTENSION	Data structure presenting Common IAL adapter related data.

8.4 Common IAL Internal State Diagrams

State Diagram Conventions

Table 7 shows the general layout for each entry of the adapter and channel state diagrams described in Section 8.4.1 and Section 8.4.2.

Table 7: State Table Description

State name or identifier		Explanation of the state
Branch condition 0	->	Next State 0
Branch condition 1	->	Next State 1
Branch condition 2	->	Next State 2
Branch condition 3	->	Next State 2

8.4.1 Common IAL Adapter State Diagram

ADAPTER_INITIALIZING		Initial adapter state
Staggered spin-up for SATA II adapter done	->	ADAPTER_READY
SATA I adapter	->	ADAPTER_READY
Error: Adapter data structure not initialized	->	ADAPTER_FATAL_ERROR

ADAPTER_READY		Working adapter state. When the adapter changes its state from ADAPTER_INITIALIZING to ADAPTER_READY, all channels' states are set to CHANNEL_DISCONNECTED and the channel initialization algorithm is triggered.
No branch to other states		

ADAPTER_FATAL_ERROR		The fatal error occurred during the adapter initialization.
No branch to other states		

8.4.2 Common IAL Channel State Diagram

CHANNEL_NOT_CONNECTED			Channel is not connected or channel initialization failed. The driver does not maintain SCSI command queue in this state.
	Storage device connection detected.	->	CHANNEL_CONNECTED

CHANNEL_CONNECTED			Channel is connected to adapter. Start SRST for channel in Gen. II and set polling timer to 31 seconds.
	1. SRST write register failed.	->	CHANNEL_NOT_CONNECTED
	2. Channel connected. Start SRST. Set timeout to 31 seconds.	->	CHANEL_IN_SRST

CHANNEL_IN_SRST			Check every 0.5 seconds if the device BSY bit is cleared. If so, start device detection.
	1. Device BSY bit is set and 31 sec. timeout has expired.	->	CHANNEL_NOT_CONNECTED
	2. Device BSY bit is set but timeout has not expired.	->	CHANNEL_IN_SRST
	3. Device BSY bit is clear and Port Multiplier (PM) device detected. Configure PM. Proceed with PM staggered spin-up for all PM ports.	->	CHANNEL_PM_STAGGERED_SPIN_UP
	4. Device BSY bit is clear and SATA disk drive is detected. Initialize SATA disk drive. Enable channel EDMA.	->	CHANNEL_READY
	5. Any failure in 3 or 4.	->	CHANNEL_NOT_CONNECTED

CHANNEL_PM_STAGGERED_SPIN_UP			PM device is detected on channel. Initiate staggered spin-up for all PM ports
	1. Staggered spin-up failed.	->	CHANNEL_NOT_CONNECTED
	2. Staggered spin-up successfully completed. Detect PM ports on which the devices are present. Set index of first PM connected device being initialized. Start device SRST. Set channel timer to 31 sec.	->	CHANNEL_PM_SRST_DEVICE

CHANNEL_PM_SRST_DEVICE			PM device is detected and staggered spin-up is completed. The channel remains in this state until all devices connected to PM are initialized.
	1. BSY bit is clear on PM connected device (disk drive) found in SRST. Initialize disk drive. There are more devices connected to PM. Start SRST on next device. Set channel timer to 31 sec.	->	CHANNEL_PM_SRST_DEVICE
	2. BSY bit is clear on PM connected device (disk drive) found in SRST. Initialize disk drive. There are no more devices connected to PM. Enable channel EDMA	->	CHANNEL_READY
	3. Failed to communicate with PM device.	->	CHANNEL_NOT_CONNECTED

CHANNEL_READY			Channel and all attached devices are initialized. Flush SCSI command queue for this channel if one is being maintained. All commands are returned with BUSY status and OS resubmits them later. If the channel has a PM connected to it and asynchronous notification is not supported by PM, start polling the PM Error information GSCR[33] register every 0.5 sec.
	1. Channel disconnect detected.	->	CHANNEL_NOT_CONNECTED
	2. PM hot plug is detected, either by polling or by receiving the event from Core Driver.	->	CHANNEL_PM_HOT_PLUG

CHANNEL_PM_HOT_PLUG			The channel remains in this state until the channel's EDMA command queue on this channel is empty. If the queue is empty, restart the channel by setting its state to CHANNEL_CONNECTED. The channel is being restarted.
	1. EDMA command queue is not empty.	->	CHANNEL_PM_HOT_PLUG
	2. EDMA command queue is empty.	->	CHANNEL_CONNECTED

8.5 Detailed IAL Function API and Data Structures

The following sections describe the function API and data structures in detail.

8.5.1 Detailed Common IAL Data Structures

8.5.1.1 Enumerators and Defines

MV_ADAPTER_STATE: Adapter state enumerator for either ADAPTER_INITIALIZING, ADAPTER_READY, or ADAPTER_FATAL_ERROR

MV_CHANNEL_STATE: Channel state enumerator for either CHANNEL_NOT_CONNECTED, CHANNEL_CONNECTED, CHANNEL_IN_SRST, CHANNEL_PM_STAGGERED_SPIN_UP, CHANNEL_PM_SRST_DEVICE, CHANNEL_READY, or CHANNEL_PM_HOT_PLUG.

MV_IAL_ASYNC_TIMER_PERIOD: Timer period define in milliseconds (equals to 500).

MV_IAL_SRST_TIMEOUT: Software reset timeout expiration value in milliseconds (equals to 31000).

8.5.1.2 Data Structures

MV_IAL_COMMON_CHANNEL_EXTENSION

MV_U8 PMnumberOfPorts

Number of ports in Port Multiplier if one is connected to current channel.

MV_U16 PMdevsToInit

Bit mask to enumerate the Port Multiplier connected devices that need further initialization: Bit value of 1 means that a device needs to be initialized.

MV_U8 devInSRST

Indicates a Port Multiplier connected device on which the software reset is in progress.

MV_BOOLEAN completionError

Indicates that SCSI command completion error has occurred on current channel. When the Port Multiplier is connected to the channel, completionError is equal to MV_TRUE and asynchronous notification is not supported by the Port Multiplier. The driver checks the change in status of Port Multiplier connected devices before submitting the next SCSI command.

MV_U8 pmAccessType

Used for Port Multiplier polling mechanism. Indicates the access type to the Port Multiplier registers: Equals either MV_ATA_COMMAND_PM_READ_REG or MV_ATA_COMMAND_PM_WRITE_REG.

MV_U8 pmReg

Used for Port Multiplier polling mechanism. Stores the Port Multiplier register number to access.

MV_BOOLEAN pmRegAccessInProgress

Indicates that a Port Multiplier register access is pending. This is used to prevent the polling routine from re-entering the register access. Equal to MV_TRUE if the register access is in progress.

MV_BOOLEAN pmAsyncNotifyEnabled

Indicates that a asynchronous notification is supported by the Port Multiplier connected to this channel.

MV_U32 SRSTTimerThreshold

Stores the timer expiration value of the software reset timer.

MV_U32 SRSTTimerValue

Stores the current timer value of the software reset timer.

MV_VOID_PTR IALChannelPendingCmdQueue

Head of the command queue managed by the Common IAL.

MV_IAL_COMMON_ADAPTER_EXTENSION

MV_SATA_ADAPTER *pSataAdapter

Pointer to the Core Driver adapter data structure.

MV_ADAPTER_STATE adapterState

Current state of the adapter.

MV_CHANNEL_STATE channelState[MV_SATA_CHANNELS_NUM]

Current state of adapter's channels.

MV_IAL_COMMON_CHANNEL_EXTENSION IALChannelExt[MV_SATA_CHANNELS_NUM]

Channels' extension of Common IAL.

8.5.2 Common IAL API

8.5.2.1 Common IAL Helper Functions

MV_BOOLEAN mvParseIdentifyResult (MV_U16_PTR iden,ATA_IDENTIFY_INFO *pIdentifyInfo)

DESCRIPTION
Parses the identify command results, checks that the connected devices can be accessed by the device EDMA, and updates the ATA drive parameters structure accordingly.

INPUT
iden - pointer to the buffer returned by the ATA IDENTIFY command

pIdentifyInfo- pointer to the ATA parameters structure

RETURN
MV_TRUE on success

MV_FALSE on failure

MV_SATA_DEVICE_TYPE mvGetSataDeviceType (MV_STORAGE_DEVICE_REGISTERS *mvRegs)

DESCRIPTION
Determines the SATA device type according to the values of ATA registers.

INPUT
mvRegs - ATA registers structure

RETURN
MV_SATA_DEVICE_TYPE_UNKNOWN - unknown device type

MV_SATA_DEVICE_TYPE_ATA_DISK - ATA disk drive

MV_SATA_DEVICE_TYPE_ATAPI_DISK - ATAPI disk drive

MV_SATA_DEVICE_TYPE_PM - Port Multiplier

MV_BOOLEAN mvInitSataDisk(MV_SATA_ADAPTER *pSataAdapter, MV_U8 channelIndex, MV_U8 PMPort, ATA_IDENTIFY_INFO *pIdentifyInfo, MV_U16_PTR identifyBuffer)

DESCRIPTION
Retrieves drive information using ATA IDENTIFY command and initialized disk drive using SET FEATURES ATA command.

INPUT
pSataAdapter - pointer to Core Driver adapter data structure

channelIndex - index of the current channel

PMPort - Port Multiplier port number

OUTPUT
pIdentifyInfo - pointer to IDENTIFY information structure

identifyBuffer - pointer to IDENTIFY buffer

RETURN
MV_TRUE on success

MV_FALSE on failure

MV_BOOLEAN mvGetPMDeviceInfo(MV_SATA_ADAPTER *pSataAdapter, MV_U8 channelIndex, MV_SATA_PM_DEVICE_INFO *pPMDeviceInfo)

DESCRIPTION
Retrieves Port Multiplier information such as vendor ID, device ID, product revision, specification revision and number of ports.

INPUT
pSataAdapter - pointer to Core Driver adapter data structure

channelIndex - index of the current channel

OUTPUT
pPMDeviceInfo - pointer to Port Multiplier information structure

RETURN
MV_TRUE on success

MV_FALSE on failure

8.5.2.2 Common IAL API State-Machine-Related Functions

MV_BOOLEAN mvAdapterStartInitialization(MV_SATA_ADAPTER *pSataAdapter, MV_IAL_COMMON_ADAPTER_EXTENSION *ialExt, MV_SAL_ADAPTER_EXTENSION *scsiAdapterExt)

DESCRIPTION
Adapter found in ADAPTER_INITIALIZING state. Starts adapter initialization: State-machine-related data structures are initialized for the adapter and its channels. After successful staggered spin-up operation the adapter state is changed to ADAPTER_READY.

INPUT
pSataAdapter - pointer to Core Driver adapter data structure

ialExt - Common IAL adapter data structure allocated by system-dependent IAL

scsiAdapterExt - pointer to SAL extension

RETURN
MV_TRUE on success

MV_FALSE on failure

void mvRestartChanne(MV_IAL_COMMON_ADAPTER_EXTENSION *ialExt, MV_U8 channelIndex, MV_SAL_ADAPTER_EXTENSION *scsiAdapterExt, MV_BOOLEAN bBusReset)

DESCRIPTION
Restarts the initialization sequence for current channel. The channel state is changed to CHANNEL_CONNECTED, the OS is notified about the change in the bus and the channel initialization is started.

INPUT
ialExt - Common IAL adapter data structure allocated by system-dependent IAL

channelIndex - channel number

scsiAdapterExt - pointer to SAL extension

bBusReset - Equals MV_TRUE if the function was called upon bus reset.

RETURN
None

void mvStopChanne(MV_IAL_COMMON_ADAPTER_EXTENSION *ialExt, MV_U8 channelIndex, MV_SAL_ADAPTER_EXTENSION *scsiAdapterExt)

DESCRIPTION
Stops current channel. Channel state is changed to CHANNEL_NOT_CONNECTED and the OS is notified about the change in the bus. All channel data structures are released.

INPUT
ialExt - Common IAL adapter data structure allocated by system-dependent IAL

channelIndex - channel number

scsiAdapterExt - pointer to SAL extension

RETURN
None

void mvPMHotPlugDetected(MV_IAL_COMMON_ADAPTER_EXTENSION *ialExt, MV_U8 channelIndex, MV_SAL_ADAPTER_EXTENSION *scsiAdapterExt)

DESCRIPTION
Called when Port Multiplier device hot plug is detected. If the channel has no outstanding EDMA commands, the channel state is changed to CHANNEL_PM_HOT_PLUG. Otherwise, the channel is restarted.

INPUT
ialExt - Common IAL adapter data structure allocated by system-dependent IAL

channelIndex - channel number

scsiAdapterExt - pointer to SAL extension

RETURN
None

*MV_BOOLEAN mvIALTimerCallback(MV_IAL_COMMON_ADAPTER_EXTENSION *ialExt, MV_SAL_ADAPTER_EXTENSION *scsiAdapterExt)*

DESCRIPTION

The system-dependent IAL must call this function from its timer callback routine. The functions of the adapter and channels' state machine are executed in every call of this function.

INPUT

ialExt - Common IAL adapter data structure allocated by system-dependent IAL

scsiAdapterExt - pointer to SAL extension

RETURN

MV_TRUE on success

MV_FALSE on failure

*void mvCommandCompletionErrorHandler(MV_IAL_COMMON_ADAPTER_EXTENSION *ialExt, MV_U8 channelIndex)*

DESCRIPTION

Called either by the SAL completion or the SMART completion function. Checks whether command failed because of PM hot plug.

INPUT

ialExt - Common IAL adapter data structure allocated by system-dependent IAL

channelIndex - channel number

RETURN

None

*MV_SCSI_COMMAND_STATUS_TYPE mvExecuteScsiCommand(MV_SATA_SCSI_CMD_BLOCK *pScb, MV_BOOLEAN canQueue)*

DESCRIPTION

IAL common layer wrapper of Core Driver *mvSataExecuteScsiCommand()* function. If the adapter state is not ADAPTER_READY or the channel is connected but the channel state is not CHANNEL_READY, the current SCSI command can be either queued in the channel's SCSI commands software queue until the channel initialization sequence is completed or immediately returned to the OS with adapter BUSY status. If the channel is in CHANNEL_READY state the SCSI command is passed to the SAL layer.

INPUT

pScb - SCSI command block structure

canQueue - determines if the IAL can queue this command

RETURN

Return MV_SCSI_COMMAND_STATUS_COMPLETED if the command has been returned to the OS.

Return MV_SCSI_COMMAND_STATUS_QUEUED_BY_IAL if the command has been queued to the channel software queue.

Otherwise return the result of *mvSataExecuteScsiCommand()* function call.

8.5.3 Common IAL API User-Supplied Routines

The system-dependent IAL layer must supply the following functions to the Common IAL:

MV_BOOLEAN IALInitChannel(MV_SATA_ADAPTER *pSataAdapter, MV_U8 channelIndex)

DESCRIPTION
Allocate and initialize all system-dependent IAL channel data structures.

INPUT
pSataAdapter - pointer to Core Driver adapter data structure

channelIndex - channel number

RETURN
MV_TRUE on success

MV_FALSE on failure

MV_BOOLEAN IALReleaseChannel(MV_SATA_ADAPTER *pSataAdapter, MV_U8 channelIndex)

DESCRIPTION
Release all system-dependent IAL channel data structures.

INPUT
pSataAdapter - pointer to Core Driver adapter data structure

channelIndex - channel number

RETURN
None

MV_BOOLEAN IALBusChangeNotify(MV_SATA_ADAPTER *pSataAdapter, MV_U8 channelIndex)

DESCRIPTION
Notify the OS about the change in the status of the current channel.

INPUT
pSataAdapter - pointer to Core Driver adapter data structure

channelIndex - channel number

RETURN
MV_TRUE on success

MV_FALSE on failure

MV_BOOLEAN IALConfigQueuingMode(MV_SATA_ADAPTER *pSataAdapter, MV_U8 channelIndex, MV_EDMA_MODE mode, MV_U8 queueDepth)

DESCRIPTION
Performs all system-dependent IAL operations for queuing mode. Then it must call *mvSataConfigEdmaMode().*

INPUT
pSataAdapter - pointer to Core Driver adapter data structure

channelIndex - channel number

mode - EDMA mode to configure for the channel. Can be either MV_EDMA_MODE_QUEUED, MV_EDMA_MODE_NOT_QUEUED or MV_EDMA_MODE_NATIVE_QUEUING

queueDepth - maximum number of outstanding EDMA commands in queue

RETURN
MV_TRUE on success
MV_FALSE on failure

8.5.4 Common IAL TWSI Devices Access

The following functions enable access to TWSI devices attached to the 88SX60X1 TWSI bus.

MV_BOOLEAN mvSataTWSIMasterInit(MV_SATA_ADAPTER *pSataAdapter)

DESCRIPTION
Initializes 88SX60X1 adapter's TWSI bus.

INPUT
pSataAdapter - pointer to Core Driver adapter data structure

RETURN
MV_TRUE on success

MV_FALSE on failure

MV_BOOLEAN mvSataTWSIMasterReadByte(MV_SATA_ADAPTER *pSataAdapter, MV_U8 twsiDevAddr, MV_U16 address, MV_U8_PTR data, MV_BOOLEAN addressRange)

DESCRIPTION

Reads a byte from TWSI device.

INPUT
pSataAdapter - pointer to Core Driver adapter data structure

twsiDevAddr - Address of the slave device in the TWSI address space

address - The required address within the TWSI device (can be either 8-bit or 16-bit)

data - A pointer to an 8-bit data container that holds the data of the result

addressRange - If MV_TRUE then the TWSI device must be accessed by 16-bit addressing.
If MV_FALSE then the device is accessed by 8-bit addressing (only the 8 LSB bits of the above address parameter are valid).

RETURN
MV_TRUE on success

MV_FALSE on failure

MV_BOOLEAN mvSataTWSIMasterWriteByte(MV_SATA_ADAPTER *pSataAdapter, MV_U8 twsiDevAddr, MV_U16 address, MV_U8 data, MV_BOOLEAN addressRange)

DESCRIPTION

Writes a byte to a TWSI device.

INPUT

pSataAdapter - pointer to Core Driver adapter data structure

twsiDevAddr - Address of the slave device in the TWSI address space

address - The required address within the TWSI device (can be either 8-bit or 16-bit)

data - The data to be written

addressRange - If MV_TRUE then the TWSI device must be accessed in 16-bit addressing.

If MV_FALSE then the device is accessed by 8-bit addressing (only the 8 LSB bits of the above address parameter above are valid).

RETURN

MV_TRUE on success

MV_FALSE on failure

Section 9. Revision History

Table 8: Revision History

Document Type	Software Version #	Document Revision	Date
Pre-release draft	Version 3.2.0	Revision 0.9	

This page is intentionally left blank.

MOVING FORWARD FASTER®

Marvell Semiconductor, Inc.

700 First Avenue
Sunnyvale, CA 94089

Phone 408.222.2500
Fax 408.752.9028

www.marvell.com

US and Worldwide Offices

Marvell Semiconductor, Inc.
700 First Avenue
Sunnyvale, CA 94089
Tel: 1.408.222.2500
Fax: 1.408.752.9028

Marvell Asia Pte, Ltd.
151 Lorong Chuan, #02-05
New Tech Park
Singapore 556741
Tel: 65.6756.1600
Fax: 65.6756.7600

Marvell Japan K.K.
Shinjuku Center Bldg. 50F
1-25-1, Nishi-Shinjuku, Shinjuku-ku
Tokyo 163-0650
Tel: 81.(0).3.5324.0355
Fax: 81.(0).3.5324.0354

Marvell Semiconductor Israel, Ltd.
Moshav Manof
D.N. Misgav 20184
Israel
Tel: 972.4.995.1000
Fax: 972.4.995.1001

Worldwide Sales Offices

Western US Sales Office
Marvell
700 First Avenue
Sunnyvale, CA 94089
Tel: 1.408.222.2500
Fax: 1.408.752.9028
Sales Fax: 1.408.752.9029

Central US Sales Office
Marvell
11709 Boulder Lane, Ste. #220
Austin, TX 78726
Tel: 1.512.336.1551
Fax: 1.512.336.1552

Eastern US/Canada Sales Office
Marvell
Parlee Office Park
1 Meeting House Road, Suite 1
Chelmsford, MA 01824
Tel: 978 250-0588
Fax: 978 250-0589

Europe Sales Offices
Marvell
3 Clifton Court
Corner Hall
Hemel Hempstead
Hertfordshire, HP3 9XY
United Kingdom
Tel: 44.(0).1442.211668
Fax: 44.(0).1442.211543

Marvell
Fagerstagatan 4
163 08 Spanga
Stockholm, Sweden
Tel: 46.16.146348
Fax: 46.16.482425

Marvell
5 Rue Poincare
56400 Le Bono
France
Tel: 33.297.579697
Fax: 33.297.578933

Israel Sales Office
Marvell
Ofek Center Bldg. 2, Floor 2
Northern Industrial Zone
LOD 71293
Israel
Tel: 972.8.914.1300
Fax: 972.8.914.1301

China Sales Office
Marvell
5J, 1800 Zhong Shan West Road
Shanghai, China 200233
Tel: 86.21.6440.1350
Fax: 86.21.6440.0799

Taiwan Sales Office
Marvell
2Fl., No. 1, Alley 20, Lane 407
Ti-Ding Blvd., Nei Hu District
Taipei, Taiwan 114, R. O. C
Tel: (886-2).7720.5700
FAX: (886-2).7720.5707

For more information, visit our website at: www.marvell.com