Block Media

Driver Design Document

Document #	Author(s)	Approval(s)

TABLE OF CONTENTS

1 Introduction

The purpose of this document is to explain the device driver design for Block Media module using DSP/BIOS operating system. This manual provides details regarding how the Block Media Driver is architected, its composition, its functionality, the requirements it places on the software environment where it can be deployed, how to customize/ configure it to specific requirements, how to leverage the supported run-time interfaces in user's own application etc.

Note: The usage of structure names and field names used throughout this design document is only for indicative purpose. These names shall not necessarily be matched with the names used in source code.

1.1 Terms and Abbreviations

Term	**Description**
API	Application Programmer's Interface
CSL	TI Chip Support Library – primitive h/w abstraction
IP	Intellectual Property
ISR	Interrupt Service Routine
OS	Operating System
MSC	Mass storage client
PSP	Platform support package
ERTFS	File System

1.2 Related Documents

1.	spru403o.pdf	DSP/BIOS Driver Developer's Guide
2.	xxxx_BIOSPSP_Userguide.doc	Block media user guide

2 System Purpose

Block Media Driver module lies below the application and file system layer. The Block Media Driver transfers calls from application/file system to the lower layer storage drivers registered. The driver would use the DSP BIOS™ APIs for OS services. The Driver is supposed to be synchronous driver.

2.1 Context

The block diagram below shows the overall system architecture:

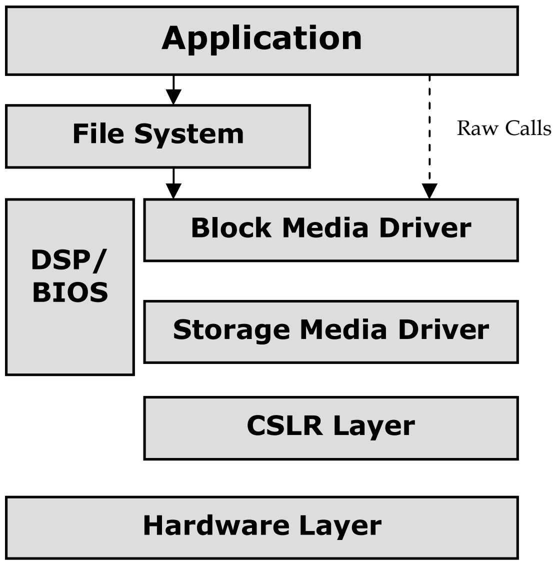


Figure 1: System Architecture

2.2 System Interface

The Block Media Driver provides generic interface for application to access any underlying storage media and also provide generic interface for the underlying storage driver to be interfaced by application.

2.3 Non-functional Requirements

The Block Media Driver does not take care of formatting and partition of the media. This should be taken care by the file system. The driver also not deal with the buffer allocation required for the FAT, File buffers etc. The system also not takes care of the drive letter assignment and no of partitions possible in the media.

3 Structure

Block media driver provides initialization, read, write, IOCTL functions which will be used by file system and RAW applications to read/write, control storage device.

Block media driver is designed such that it gets call from file system and then it performs all file system and OS related operations and diverts it to storage driver. It provides wrapper to storage driver functions. Storage driver need not to bother about file system related operation so that storage media drivers can be designed file system independent.

The block media driver takes care of cache alignment if the application provides cache unaligned buffer for accessing the storage media during read/write. It provides alignment of data to catch line size. User can enable this functionality by enabling `PSP_BUFFER_ALIGNMENT` switch in `psp_blkdev.h`

Block media collects storage device information during registration and store it in Block device information table and provides this information to file system whenever required.

Block media provides all required functionality to the file system on behalf of storage driver. It is a common interface for all storage drivers, so there is no need to implement same code in all storage drivers.

Block media driver provides Sync read/write to or from storage device. It treats file system and RAW read/write call as Sync read/write. Currently block media works as in synchronous mode for its access to the media driver for both the access to media by file system or RAW mode. The RAW mode of access can be extended to asynchronous mode, if enhancement is required.

3.1 Overview

Block media driver is designed as a monolithic block of code in a single file as it is just a generic abstraction layer between storage media drivers and File system/applications. Storage driver gets themselves registered to the block media driver so that application can use their services seamlessly.

3.2 Components

Block media driver is divided in to four components mainly:

- ➢ One part deals with the initialization and de-initialization of block media driver.
- ➢ The other part defines interfaces for RAW application to talk to block media layer.
- ➢ The third part deals with calling the IOCTLS with file system.
- ➢ The last part deals with how to interface with the underlying storage media.

3.2.1 DMA Configuration

Block media Driver uses DMA provided in the SoC (EDMA). EDMA is primarily used to move the data from peripheral to application supplied buffers and vice versa, in case of unaligned buffer being passed from and application and alignment is used in block media layer. In this implementation, we use two separate EDMA channels to move data (for File system and RAW mode). The EDMA is configured to move data only when there is an unaligned read/write request from the application and the macro for alignment is enabled.

3.3 Interfaces

In the following subsections, the interfaces implemented by each of the sub-component are specified. Please refer to BIOSPSP_blkmedia.chm for complete details on APIs. Following interfaces are used for block media layer

3.3.1 Initialization Interfaces

The initialization of block media driver is done by, PSP_blkmediaDrvInit, which takes care of initializing the resources required for block media and allocating all the resources for the storage media device. The block media allocates all the resources required for the each storage media present and therefore needs to be initialize once throughout the system and should not be initialized for every media driver used from the application. The de-initialization of block media, PSP_blkmediaDrvDeInit, takes care of returning or de-allocating all the resources.

Following are the functions defined for this:

- Int32 PSP_blkmediaDrvInit(Ptr hEdma, Uint32 edmaEventQ, Uint32 taskPrio, Uint32 taskSize);
- Int32 PSP_blkmediaDrvDeInit(Void);

3.3.2 RAW application Interfaces

The application when wants to use the underlying storage driver in RAW mode then it uses these interfaces. The application first tells the block media which storage driver will be accessed for RAW mode of operation by calling the function PSP_blkmediaDrvIoctl() and passing appropriate parameters. This device is now set for all RAW calls form application. Then the application registers itself for RAW mode of operations using PSP_blkmediaAppRegister(). The application then can use the function pointer returned to call the read, write and IOCTL operations. Once application is done it can call the un-register function, PSP_blkmediaAppUnRegister(), to un-register itself with block media.

Following are the functions defined for this:

- Int32 PSP_blkmediaDrvIoctl(Ptr *pDevName, PSP_BlkDrvIoctlInfo_t *pIoctl);
- Int32 PSP_blkmediaAppRegister(PSP_blkDevCallback_t appCb, PSP_BlkDevOps_t **pIntOps, Ptr *pHandle);
- Int32 PSP_blkmediaAppUnRegister(Ptr handle);

3.3.3 IOCTL Interfaces

To use the IOCTLS defined for storage driver and block media, the application needs to call the PSP_blkmediaDevIoctl() IOCTL function with the device id for which this IOCTL is meant as a parameter and appropriate IOCTL enum.

Following are the functions defined for this:

- Int32 PSP_blkmediaDevIoctl(PSP_BlkDrvId_t driverId, PSP_BlkDevIoctlInfo_t *pIoctl);

3.3.4 Storage driver Interfaces

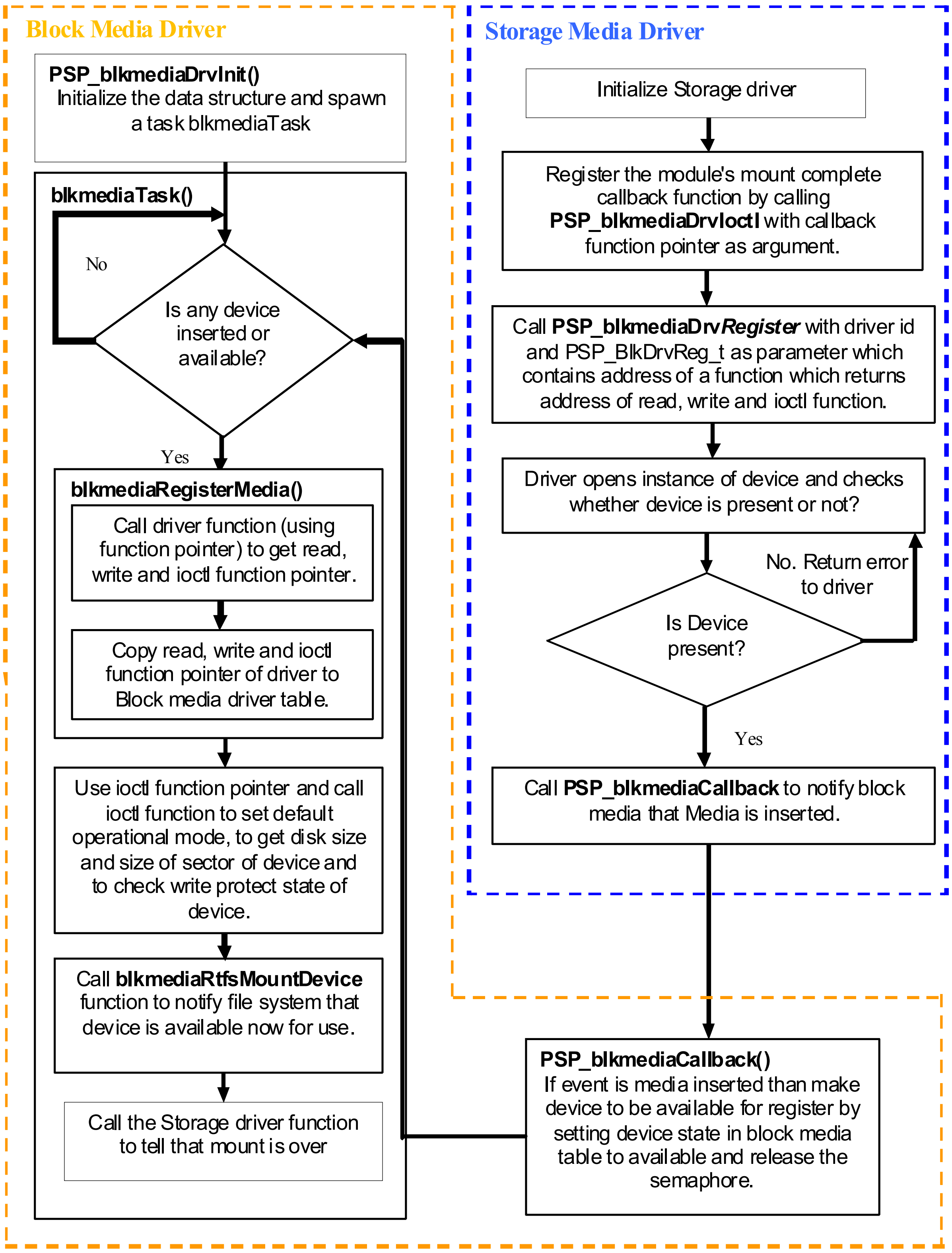
The storage driver needs to call the API, PSP_blkmediaDrvRegister(), of block media so that once the device is available it can be registered with the file system. Also underlying storage media needs to register its read/write/IOCTL function with block media using PSP_blkmediaCallback(), so that the call from application can be diverted to the underlying storage media.

- Int32 PSP_blkmediaDrvRegister(PSP_BlkDrvId_t driverId, PSP_BlkDrvReg_t const *pRegInfo);
- Void PSP_blkmediaCallback(PSP_BlkDrvId_t driverId, PSP_BlkDevEvent_t const *pEventInfo);

4 Dynamic Behavior

This section describes the various scenarios for block media like the storage driver registration, read/write etc.

4.1 Storage driver registration sequence


Block Media Driver
PSP_blkmediaDrvInit()
Initialize the data structure and spawn a task blkmediaTask
blkmediaTask()
No
Is any device inserted or available?
Yes
blkmediaRegisterMedia()
Call driver function (using function pointer) to get read, write and ioctl function pointer.
Copy read, write and ioctl function pointer of driver to Block media driver table.
Use ioctl function pointer and call ioctl function to set default operational mode, to get disk size and size of sector of device and to check write protect state of device.
Call blkmediaRtfsMountDevice function to notify file system that device is available now for use.
Call the Storage driver function to tell that mount is over
Storage Media Driver
Initialize Storage driver
Register the module's mount complete callback function by calling PSP_blkmediaDrvIoctl with callback function pointer as argument.
Call PSP_blkmediaDrvRegister with driver id and PSP_BlkDrvReg_t as parameter which contains address of a function which returns address of read, write and ioctl function.
Driver opens instance of device and checks whether device is present or not?
No. Return error to driver
Is Device present?
Yes
Call PSP_blkmediaCallback to notify block media that Media is inserted.
PSP_blkmediaCallback()
If event is media inserted than make device to be available for register by setting device state in block media table to available and release the semaphore.

Block media driver initializes all data structures and Block device table and then spawn a task blkmediaTask(), which will executes infinitely with high priority (priority of 2). blkmediaTask() task will continuously check for device to be available for register.

Storage media driver will initialize all data structures and table. Storage driver will call PSP_blkmediaDrvRegister() function by passing driver id as a first parameter and address of PSP_BlkDrvReg_t structure variable which contains address of driver register function. This driver register function returns address of Media_Read, Media_Write and Media_Ioclt function of driver.

Storage Driver will open instance of storage device and check whether device is available/inserted or not. If it is available/inserted than driver will call PSP_blkmediaCallback() function by passing (PSP_BlkDrvId_t) driver Id, and address of PSP_BlkDevEvent_t structure variable which contains information about event (Media inserted or removed event).

PSP_blkmediaCallback() function will set state of device to available in block device information table. blkmediaTask() task will now call blkmediaRegisterMedia() function which will call a function of storage driver which returns function pointer of read, write and ioctl function of storage driver. Block media copy these function pointers to Block driver register (BLK_DEV_Info_t) table.

Block media will use function pointer of ioctl function of Storage driver and set operational mode, get disk size and sector size and checks whether device is write protected or not.

Block media will notify to File system (ERTFS) that device is available for read and write by calling pc_rtfs_media_insert() function of ERTFS through blkmediaRtfsMountDevice(). At successful return from this function block media calls driver call back function to notify driver that driver is registered with block media and file system and now user can perform read/write on device. Driver must set this call back function using BLK_DRV_SET_INIT_COMP_CALLBACK IOCTL.

4.1.1 Event notification interface

There are two callback functions available in Block media driver which can be used by driver to inform block media about lower level event like media inserted or media removal and read write completion in case of asynchronous read/write function call.

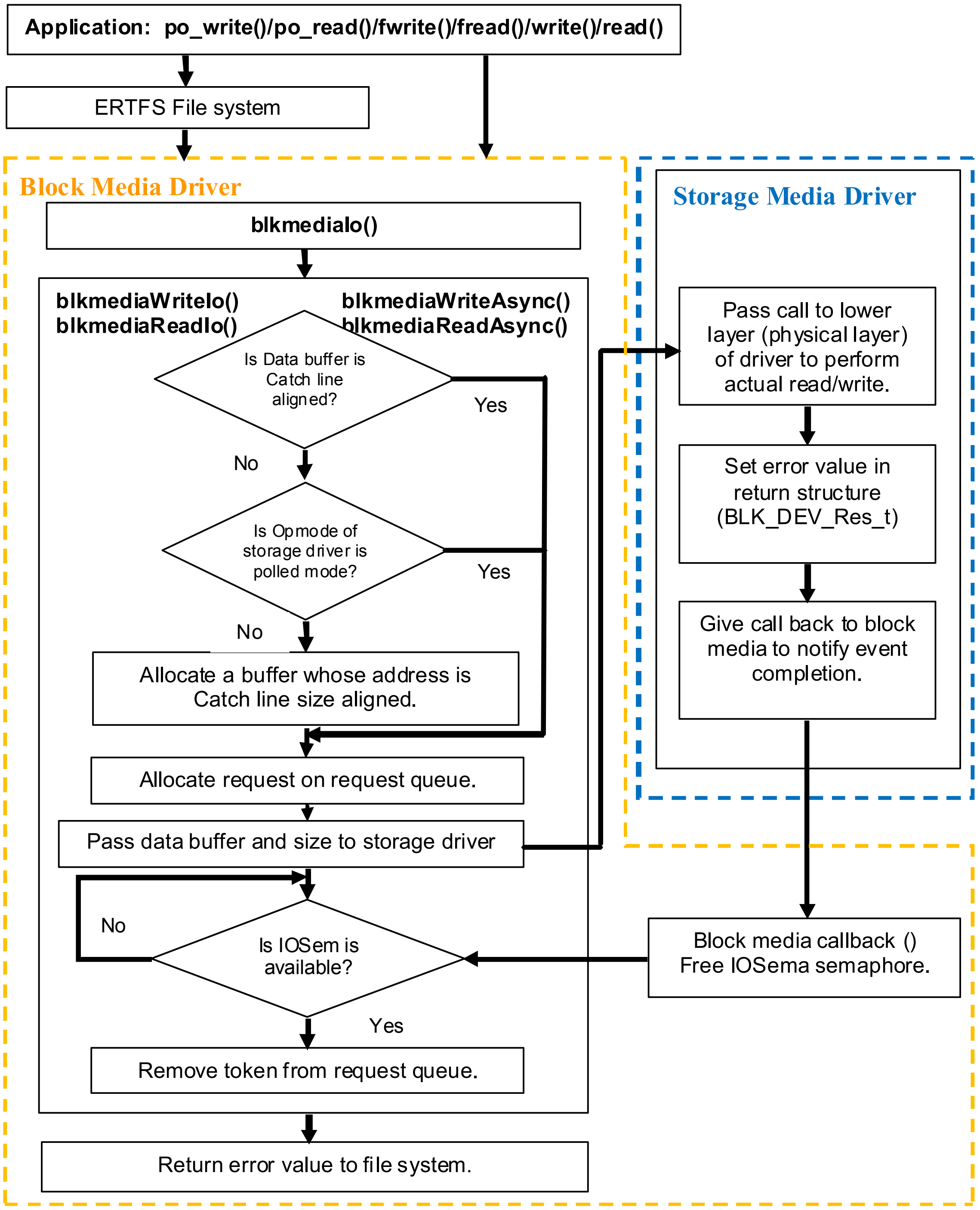
1. Void PSP_blkmediaCallback(PSP_BlkDrvId_t driverId, PSP_BlkDevEvent_t const *pEventInfo);

This callback is used, when storage driver detects that device is available or inserted. This function is used for propagating events from the underlying storage drivers to the block driver (Ex. Device insertion/removal, media write protected). Storage driver calls PSP_blkmediaCallback() function with device id as a first parameter and address of variable of PSP_BlkDevEvent_t structure as a second parameter declared in `psp_blkdev.h` file. PSP_BlkDevEvent_t structure contains two elements EventId and EvtData. EventId is type of event generated and EvtData is a void pointer. Storage driver will pass address of DevHandle as an event data.

2. static Bool blkmediaCallback(Ptr handle, PSP_BlkDevRes_t const *pRes)

After completion of every read/write call driver calls blkmediaCallback() function to return error value and to notify block media driver and in turn file system that read/write has been completed. Storage driver has to pass Media Driver Handle of device, which was received in read/write call from Block media driver as a first parameter and pointer to error information structure (PSP_BlkDevRes_t *) as a second parameter by setting error value which was generated during read/write function call.

4.2 Read/write sequence


Application: po_write()/po_read()/fwrite()/fread()/write()/read()
ERTFS File system
Block Media Driver
blkmediaIo()
blkmediaWriteIo()
blkmediaReadIo()
blkmediaWriteAsync()
blkmediaReadAsync()
Is Data buffer is Catch line aligned?
Yes
No
Is Opmode of storage driver is polled mode?
Yes
No
Allocate a buffer whose address is Catch line size aligned.
Allocate request on request queue.
Pass data buffer and size to storage driver
No
Is IOSem is available?
Yes
Remove token from request queue.
Return error value to file system.
Storage Media Driver
Pass call to lower layer (physical layer) of driver to perform actual read/write.
Set error value in return structure (BLK_DEV_Res_t)
Give call back to block media to notify event completion.
Block media callback ()
Free IOSema semaphore.

To port any storage driver on block media, the driver needs to implement wrapper of following three functions.

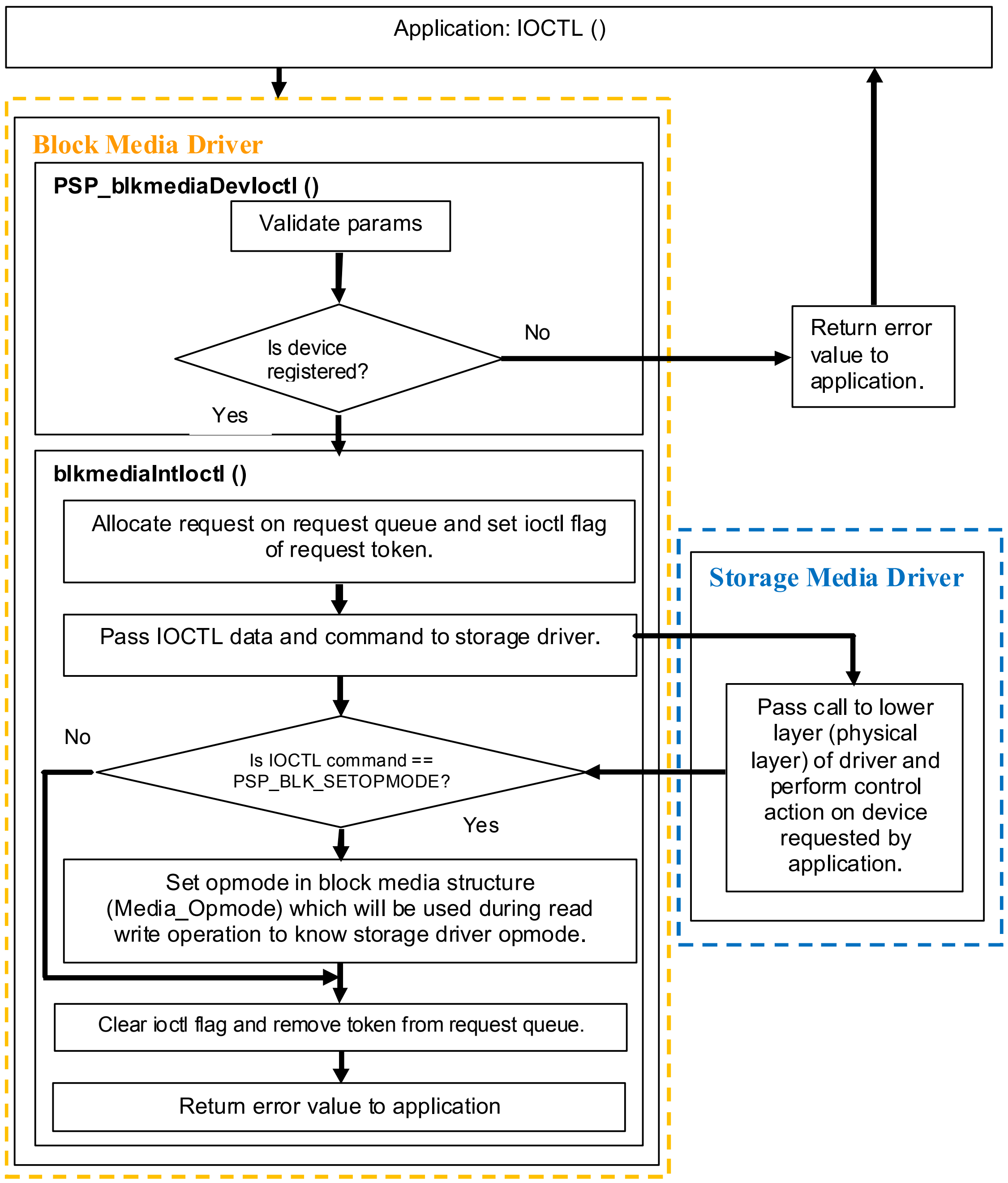
1. Media_Read
2. Media_Write
3. Media_ioctl

Driver must support Media_Read, Media_Write and Media_ioctl functions to be interfaced with block media driver. Media_Read and Media_Write function must follow following prototype:

Int32 Media_Write (Ptr handle, Ptr mediaHandle, Ptr buffer, Uint32 sectNum, Uint32 numSect)
Int32 Media_Read (Ptr handle, Ptr mediaHandle, Ptr buffer, Uint32 sectNum, Uint32 numSect)
Int32 Media_Ioctl(Ptr handle, Ptr mediaHandle, PSP_BlkDevIoctlInfo_t const *pIoctlInfo, Bool *pIsComplete)

4.3 IOCTL sequence

Unlike read/write, IOCTL function execution flow does not differ for RAW and File System interface. File system does not provide IOCTL function so user needs to call IOCTL function PSP_blkmediaDevIoctl() using block media interface. User can call IOCTL function of driver using the storage device id. Control operation will not be performed on whole device. Device information retrieved using this IOCTL will be device specific. Following is a flow diagram of IOCTL function:


Application: IOCTL ()
Block Media Driver
PSP_blkmediaDevIoctl ()
Validate params
Is device registered?
No
Yes
Return error value to application.
blkmediaIntIoctl ()
Allocate request on request queue and set ioctl flag of request token.
Pass IOCTL data and command to storage driver.
Storage Media Driver
Pass call to lower layer (physical layer) of driver and perform control action on device requested by application.
Is IOCTL command == PSP_BLK_SETOPMODE?
No
Yes
Set opmode in block media structure (Media_Opmode) which will be used during read write operation to know storage driver opmode.
Clear ioctl flag and remove token from request queue.
Return error value to application

Driver must implement ioctl commands to be used by block media during initialization. These ioctl commands will be used

1. To set operational mode of driver: PSP_BLK_SETOPMODE

   Block media sets best operation mode of storage driver by passing BLK_DEV_AUTO as parameter. Wrapper function of IOCTL must check for this auto mode and set default mode of storage driver by diverting a call to storage driver.

2. To get Disk Capacity: PSP_BLK_GETSECTMAX
3. To get Sector Size: PSP_BLK_GETBLKSIZE
4. To check device is write protected or not: PSP_BLK_GETWPSTAT
5. To get operational mode of driver: PSP_BLK_GETOPMODE

For a detailed list of IOCTL please refer to the `psp_blkdev.h`

5 Revision History

Version #	Date	Author Name	Revision History
0.1	07/01/2009	Vipin Bhandari	Document Created