MMCSD

Driver Design Document

Document #	Author(s)	Approval(s)

Template Version 0.1

TABLE OF CONTENTS

LIST OF FIGURES

1 Introduction

The purpose of this document is to explain the device driver design for MMCSD peripheral using DSP/BIOS operating system. This manual provides details regarding how the MMCSD 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
MMC	Multi Media Card
SD	Secure Digital
SDHC	High capacity SD card

1.2 Related Documents

1.	spru403o.pdf	DSP/BIOS Driver Developer's Guide
2.	xxxx_BIOSPSP_Userguide.doc	MMCSD user guide
3.	Part 1 Physical Layer Specification Ver2[1].00 Final 060509.pdf	SD protocol Spec
4.	SPRUFM2_MMCSD.pdf	MMCSD H/W Controller

2 System Purpose

The multimedia card (MMC)/secure digital (SD) card is used in a number of applications to provide removable data storage. The MMC/SD card controller provides an interface to external MMC and SD cards. The communication between the MMC/SD card controller and MMC/SD card(s) is performed according to the MMC/SD protocol.

MMCSD driver lies below the Block Media module. The Block Media Driver transfers calls from application/file system to the MMCSD driver which is registered to block media. The driver would use the DSP BIOS™ APIs for OS services. The Driver is supposed to be asynchronous driver. The MMCSD driver actually read/write the data to the card.

2.1 Context

The block diagram below shows the overall system architecture:

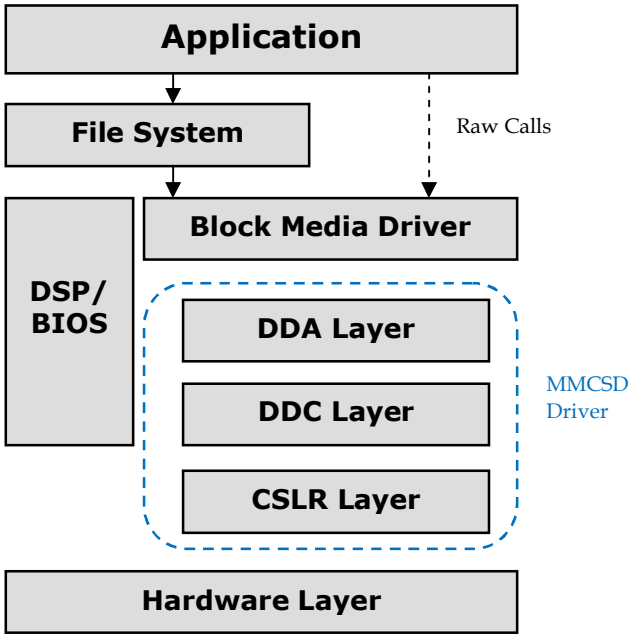


Figure 1: System Architecture

The central portion shown constitutes the mainline MMCSD driver component; the surrounding modules constitute the supporting system components. The supporting system modules, do not specifically deal with MMCSD, but assist the driver by providing utility services.

2.2 System Interface

The MMCSD driver provides interface to read write the data to the actual MMC or SD card using the MMC or SD protocol. The driver also provides various other services like providing the vendor id, set/get write protect status of the card, checking whether card is high capacity or not, get the size of card etc.

2.3 Non-functional Requirements

The MMCSD driver does not know anything about the file system being used and in no way dependent on it for its functioning. So formatting, recognizing the partition and understanding the FAT does not come under the purview of the driver.

The MMCSD driver is expected to:

- Asynchronous Read/Write interface for single as well as multiple blocks.
- Provide mechanism to transfer in Polled and DMA mode.

3 Structure

This section deals with the overall architecture of MMCSD device driver, including the device driver partitioning as well as deployment considerations. We'll first examine the system decomposition into functional units and the interfaces presented by these

units. Following this, we'll discuss the deployed driver or the dynamic view of the driver where the driver operational scenarios are presented.

3.1 Overview

The device driver is partitioned into distinct sub-components, consistent with the roles and responsibilities. The MMCSD device driver functionality can be enacted by three key roles defined here under:

- Device Driver Adaptation (DDA), takes care of interfacing with the generic block media layer
- Device Driver Core (DDC), implements the protocol part of the driver
- Low level Controller (LLC), provides services to perform primitive access necessary to control/configure/examine status, of the underlying h/w device.

3.2 Components

It should be clear by now that a device driver is composed of three main sub-components – the HW specific bottom-edge, the block media specific upper-edge and the central device driver core.

Since there exists a clear separation of roles and responsibilities of the three sub-components of the driver, the prescribed architecture helps in creation of robust device drivers through tested/reusable pieces. In the following sub-sections, each of these functional sub-components of the mmcsd device driver is further elaborated.

3.2.1 H/W Device Specific Layer (LLC)

The LLC forms the lower most, h/w specific under-pinning of the MMCSD device driver. It consists of two parts:

3.2.1.1 ***CSL Register Layer:*** This is comprised of symbolic constants (#defines) that expose the register bit-field details of the h/w along with assorted other constants such as bit-field masks, shift values and default settings. These constants can be supplied to a set of eight generic, bit field manipulation macros in order to read/write values into device h/w registers.

A C-structure data type definition is provided as an as-is map of the peripheral device registers in the processor's memory map. This structure is termed the Register Overlay structure. The intended mode of usage is to initialize a pointer variable to base address of the peripheral h/w device and typecast it using structure overlay definition. This way, an otherwise un-adorned pointer assumes a strong C-type thereby making it possible to program the h/w registers by read/write to structure member elements.

3.2.1.2 ***LLC Layer:*** This layer comprises of C functions and makes use of the CSL Register Layer for implementation. The functions supported include:

- Operations to formally begin/end access to device h/w
- Operations to perform one time setup of the H/W device such as during device driver initialization
- Operations to program the H/W device registers to change one or more of its configuration parameters
- Operations to query and infer the current state of the H/W device

It is important to note here that LLC scope is limited to directed access of the underlying h/w device. It does not depend on any specific OS and does not exploit optimizations specific to a given compiler. In contrast to a device driver, it does not perform operations that are viewed as management of data movement over the peripheral device. It does not model state machines or protocols.

3.2.2 Device Driver Core functionality (DDC)

The DDC module portion of the driver provides basic behavior of the driver, modeling the main FSMs and protocols. The DDC does not directly touch the underlying h/w; it does so via the LLC/CSL.

The objective of a good driver partitioning is to sediment as much of the driver behavior into the DDC as is practical. This calls for a DDA that is thin and efficient. Reuse and performance improvement efforts can then focus on DDC where bulk of the functionality is realized.

DDC implements a set of functions that constitute the driver functional interface. The functions defined under DDC, purely act upon the driver data structures and hardware. Following are the functions that will be implemented in the DDC layer which will be used from the above layer.

- Int32 PSP_mmcsdInitialize(Uint32 instNum, PSP_MmcsdHandle* const hDDC, PSP_MmcsdConfig* const pConfig, Uint32 moduleInputClk);
- Int32 PSP_mmcsdOpenChannel(PSP_MmcsdHandle hDDC, PSP_MmcsdChannelHandle* const hChannel, PSP_MmcsdChannelConfig* const chnlConfig);
- Int32 PSP_mmcsdCloseChannel(PSP_MmcsdChannelHandle hChannel);
- Int32 PSP_mmcsdInput(PSP_MmcsdChannelHandle hChannel, Ptr mediaHandle, Uint32 address, Uint16* bufPtr, Uint32 bufSz, Uint32 timeout);
- Int32 PSP_mmcsdOutput(PSP_MmcsdChannelHandle hChannel, Ptr mediaHandle, Uint32 address, Uint16* bufPtr, Uint32 bufSz, Uint32 timeout);
- Int32 PSP_mmcsdStatus(PSP_MmcsdChannelHandle hChannel, PSP_MmcsdIoctl cmd, Ptr const arg, Ptr param);
- Int32 PSP_mmcsdTerminate(PSP_MmcsdHandle hDDC);

3.2.3 Device Driver Adaptation (DDA)

As discussed above, the DDC is not complete unless it is supplemented by the DDA. The DDA adapts the mmcsd driver core to the block media layer. DDA implements aspects such as – registering itself with block media, giving the read write and ioctl function pointers and updating the callback function to be called in case of read and write finishes.

Following are the functions that shall be implemented in this layer.

- Int32 PSP_mmcsdDrvInit(Uint32 moduleFreq, Uint32 instanceId, PSP_MmcsdConfig* const config);
- Int32 PSP_mmcsdDrvDeInit(Uint32 instanceId);
- Int32 PSP_mmcsdCheckCard(PSP_MmcsdCardType* cardType, Uint32 instanceId);

3.2.4 DMA Configuration

MMCSD 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 this implementation, we use two separate EDMA channels to move data (transmit and receive data). The EDMA is configured to move data only when there is VALID read request from the application. In the absence of IO requests the driver/application will not receive any data.

3.3 Interfaces

In the following subsections, the interfaces implemented by each of the sub-component are specified. Please refer to BIOSPSP_mmcsd.chm for complete details on APIs

3.3.1 DDA Interface

The user of device driver will only need bother about the DDA interface, as these are the services exposed to the Application. All other interfaces discussed later in this document are more of interest to people developing/maintaining the mmcsd device driver.

The PSP_mmcsdDrvInit() function is the main init function of the driver. The function first registers the module's mount complete callback function with block media. This registered function will be called from the block media once the device is mounted. The function then calls PSP_mmcsdInitialize() and PSP_mmcsdOpenChannel() which populates static settings in driver object, creates the necessary interrupt handler, registers the interrupt, and attaches the Driver Core interfaces. The function then register the MMCSD Driver with the Block Device Driver by calling PSP_blkmediaDrvRegister() and giving the address of a function which returns the read write and ioctl function. This registered function will be called by block media driver when MMCSD will be detected at hardware layer. The function then checks if the card is present by calling the PSP_mmcsdCheckCard(). This function upon checking that the card is present calls PSP_blkmediaCallback() in block media for propagating the device insertion event from the MMCSD driver to the block driver. All these operations in effect, constitute the "loading" of MMCSD driver.

The PSP_mmcsdCheckCard() function check for the availability of any card present and updates the parameter with the card type, if one is present.

The PSP_mmcsdDrvDeInit() de-initializes the mmcsd driver. This function calls the PSP_mmcsdCloseChannel() and PSP_mmcsdTerminate() to close and terminate the mmcsd driver. It then calls PSP_blkmediaCallback() to propagate the device removal event to block media driver.

3.3.2 DDC Interface

DDC forms the heart of mmcsd device driver. It models driver state machine and implements the data movement (read/write). The Device DDC is inherently asynchronous and contains the crux of the ISR functionality. The mmcsdIsrHandler() and mmcsdDmaIsrHandler() are the function registered for mmcsd interrupts and dma interrupts.

The h/w access and IO completion callback are not completed in the ISR context itself. The MMCSD isr posts the semaphore on which the mmcsd task waits and from there IO completion callback completes.

All IO Transactions are performed via jobs called IO Packets or IOPs. The DDC takes the role of preparing Job packets. Device IOCTL can be performed on an opened mmcsd controller. The DDC only implements a fully asynchronous data-mover.

The driver instance is created and initialized through PSP_mmcsdInitialize() passing setup parameters. Following this, the device controller and IO Channel are opened via PSP_mmcsdOpenChannel(). Once the device has been opened, transactions can be performed using PSP_mmcsdInput() and PSP_mmcsdOutput() for input and output of data and PSP_mmcsdStatus() for ioctl methods.

Upon completion of use, driver can be gracefully closed and removed from system by following sequence of calls to PSP_mmcsdCloseChannel() and PSP_mmcsdTerminate().

3.3.3 LLC Interface

The services implemented by LLC are restricted only to touching the hardware using CSL and can be leveraged in DDC implementation. It should be noted that LLC never calls DDC functions; it's always the other way round.

LLC services will have usual call sequence of: LLC_mmcsdInit (), etc. LLC maintains its own instance specific data and executes in caller's context and interfaces non-blocking in nature. They do not allocate memory dynamically and do not use OS services.

4 Dynamic Behavior

The mmcsd device driver typically implement asynchronous interface to the user. The actual implementation may or may not allow for use of Interrupts and DMA.

The MMCSD device driver operation involves following execution threads:

- All Synchronous IO occur in the application thread of control, the calling thread may suspend for the requested transaction to complete. This 'wait on completion' occurs in the DDC layer.
- All Asynchronous IO complete in the Interrupt thread of control, the calling thread is freed as soon as the job is submitted to the driver. The calling thread is notified (indirectly through an ISR) asynchronously via an Interrupt.

In the case of DMA mode, driver uses task which is created at init time of driver. Request is taken from calling thread and further transaction with controller is done from task thread which has been created by driver at init time. Synchronization between mmcsd interrupt and this task thread has been done using semaphore. Priority of this task can be changes as per the need of application. Priority and stack size of the mmcsd task could be set by using macros `PSP_MMCSD_DEV_TASKPRIO` and `PSP_MMCSD_DEV_STACK_SIZE_BYTE` respectively, which is available in file `dda_mmcsdCfg.h` file. Please make sure that the driver task has higher priority than the application task.

4.1 Driver Creation/Initialization

The sequence diagram below depicts the creation/initialization phase of the MMCSD driver. Once this phase is complete, the basic driver data structures and setups are complete and ready for formally opening device to perform IO.

User is expected to invoke PSP_mmcsdDrvInit() which in turn will call PSP_mmcsdInitialize(). The PSP_mmcsdInitialize() performs book-keep functions on the driver and updates instance data structures.

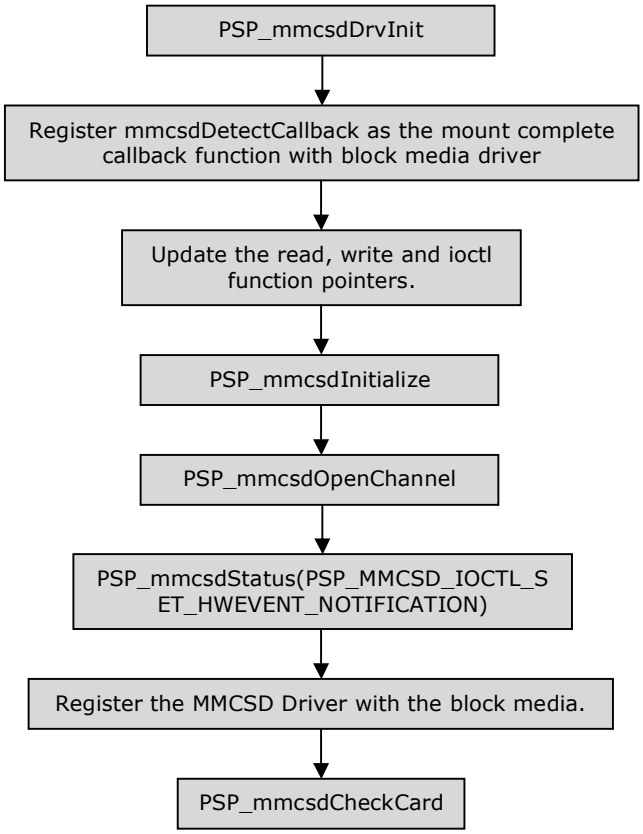
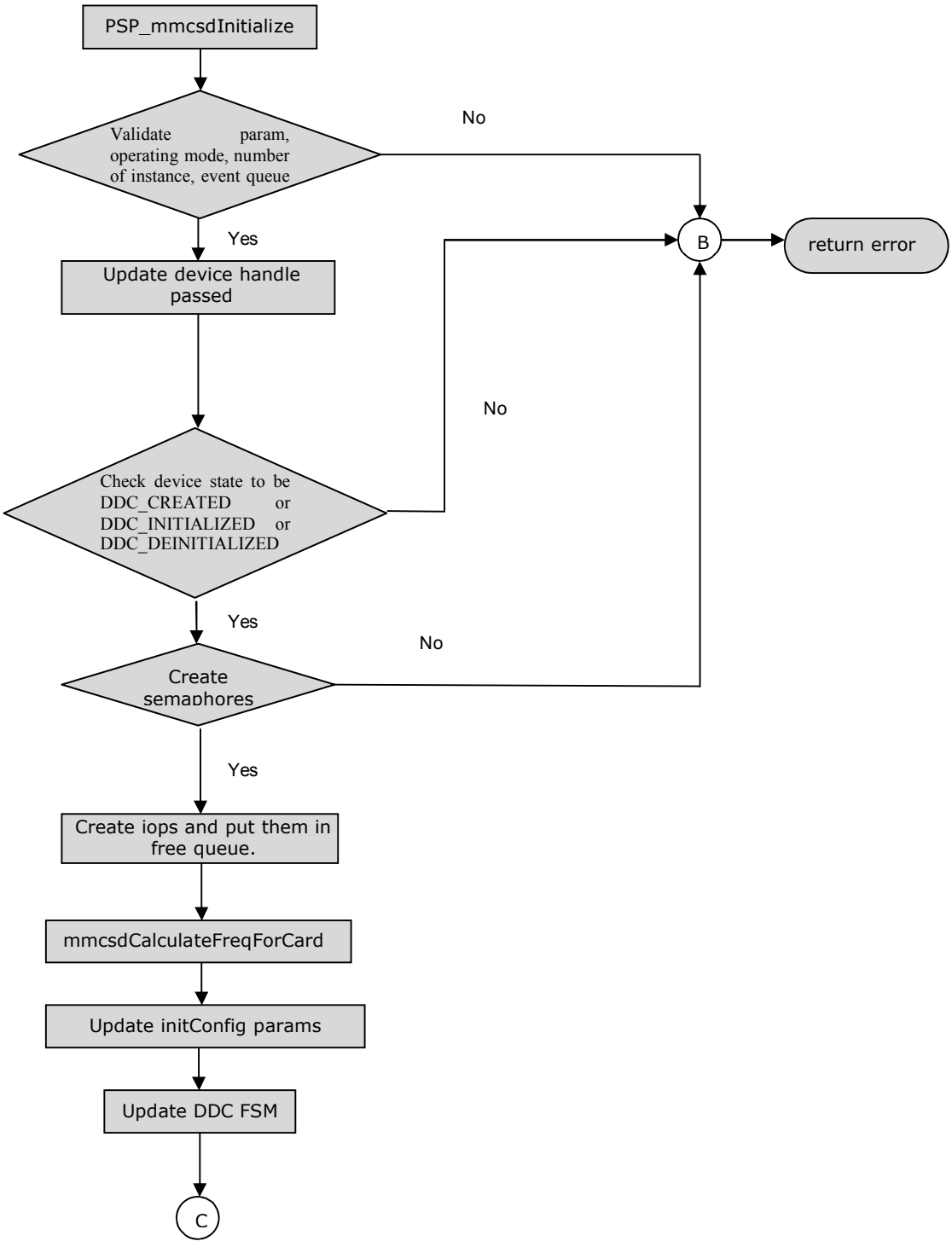


Figure 2: MMCSD Driver Creation Flow Diagram


PSP_mmcsdInitialize
Validate param, operating mode, number of instance, event queue
No
Yes
B
return error
Update device handle passed
No
Check device state to be DDC_CREATED or DDC_INITIALIZED or DDC_DEINITIALIZED
Yes
No
Create semaphores
Yes
Create iops and put them in free queue.
mmcsdCalculateFreqForCard
Update initConfig params
Update DDC FSM
C

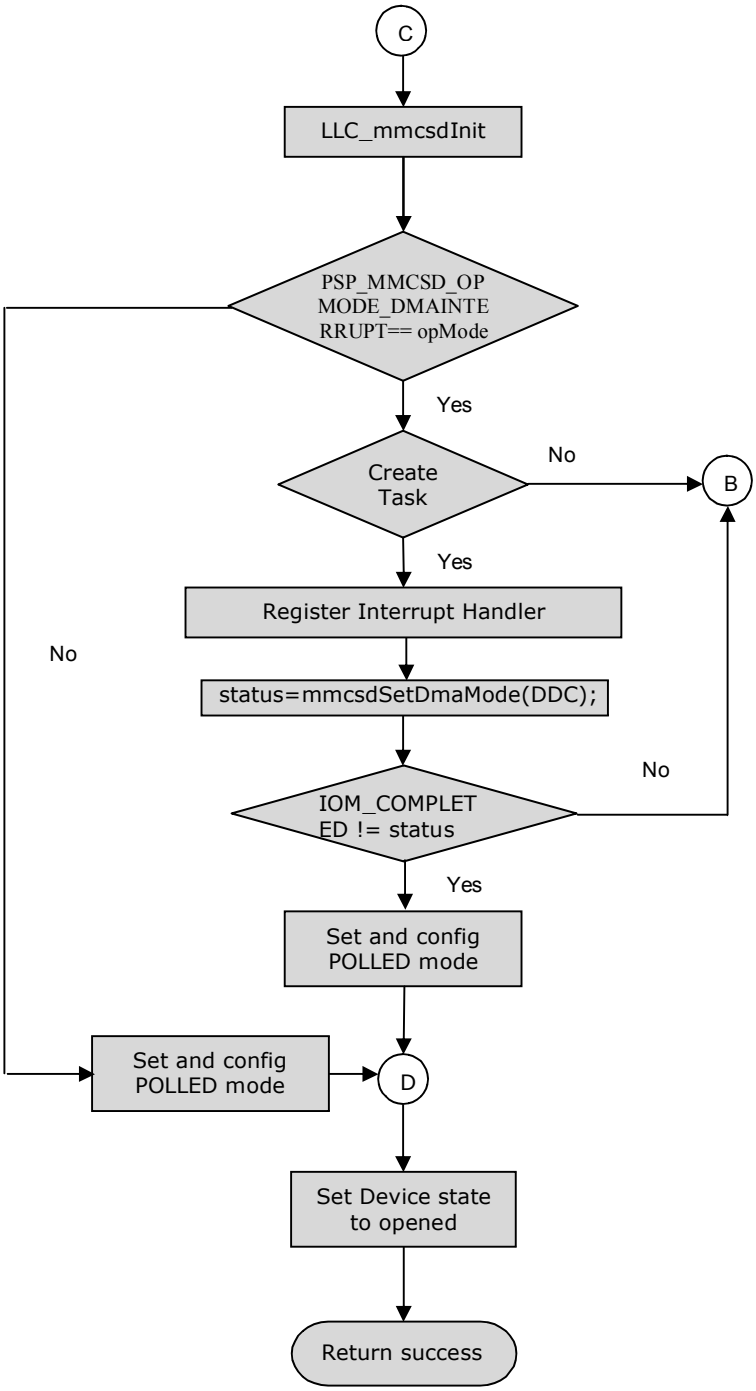


Figure 3: MMCSD Driver Creation Flow Diagram

4.2 Driver Open

When the application calls the PSP_mmcsdOpenChannel () driver entry point, the DDC registers the application callback function which will be called on I/O completion and also device state is started. The device driver is now ready to accept Read/Write jobs.

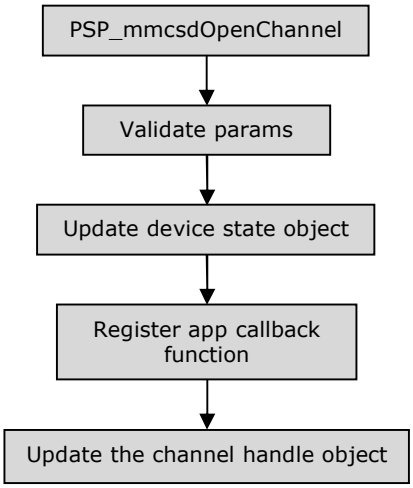


Figure 4: MMCSD Driver Open Flow Diagram

4.3 IO Control

The MMCSD Driver provides an IOCTL interface to set/get common configuration parameters on the driver at run time. The PSP_mmcsdStatus() function needs to be invoked for the same.

It should be observed that the user's IOCTL request completes in the context of calling thread i.e., application thread of control.

4.4 IO Access

The application invokes the PSP_mmcsdOutput() and PSP_mmcsdInput() IO interfaces for data transfer using the MMCSD. Here, the DDC's mmcsdSubmitIoRequestSingle() service is invoked. The DDC creates an IOP packet and put the request in pending queue. It then starts the transfer by reading the IOIP from the pending queue.

At some later point in time, when the H/W interrupt is asserted, the ISR (DMA as well as MMCSD interrupt) is posted. If requested IO is completed, packet is shifted to complete queue. Else, it simply returns waiting for further H/W action.

At the DDC level, upon IO completion notification, a check is made to see if user has registered a callback. If yes, the user callback is invoked with the correct context information, which in turn returns processed buffer to user and frees the IO Packet. Following completion of IO, the packet is recycled back to the free IOPs pool in the DDC.

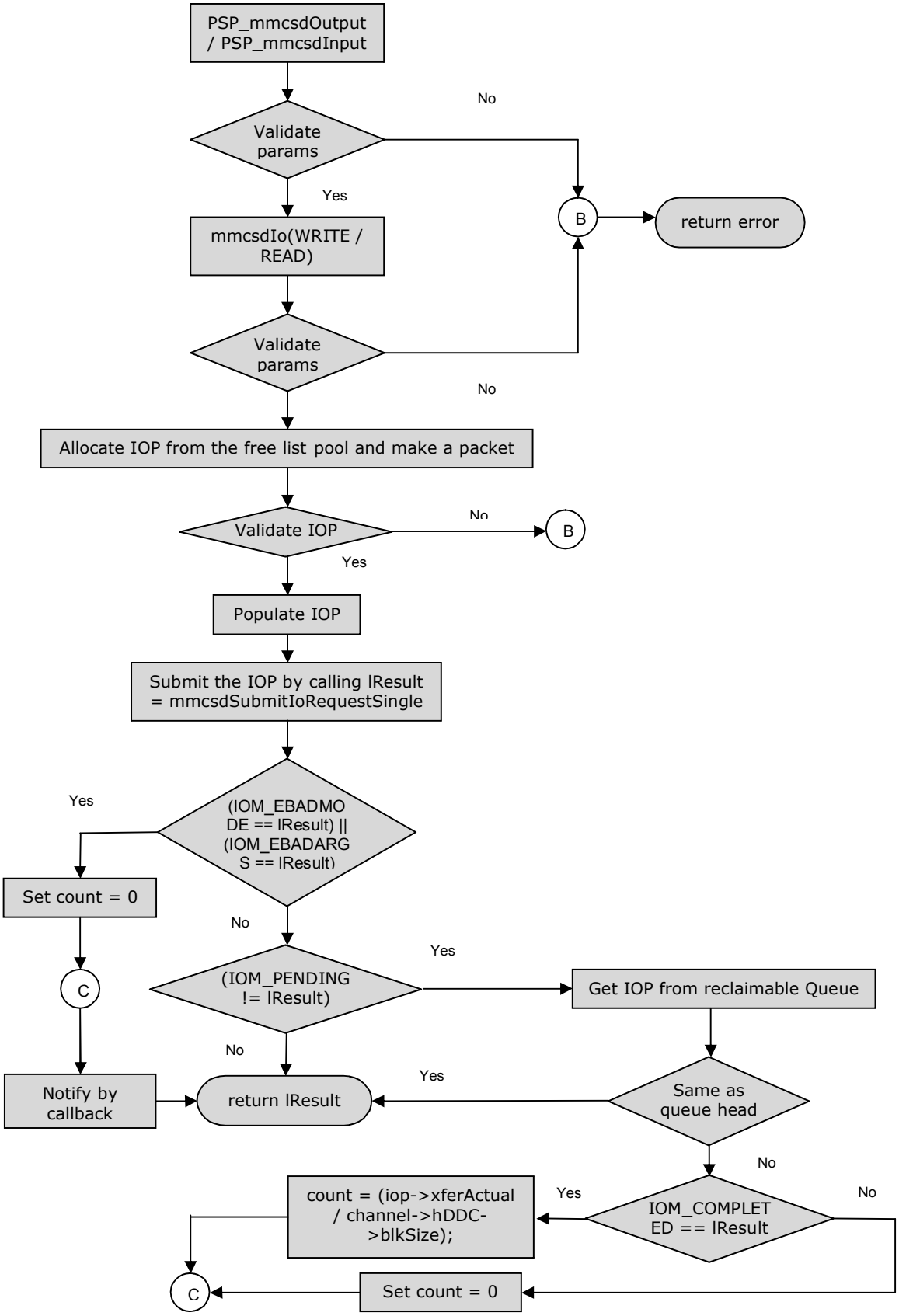

PSP_mmcsdOutput / PSP_mmcsdInput
Validate params
No
Yes
B
return error
mmcsdIo(WRITE / READ)
Validate params
No
Allocate IOP from the free list pool and make a packet
Validate IOP
No
B
Yes
Populate IOP
Submit the IOP by calling lResult = mmcsdSubmitIoRequestSingle
Yes
(IOM_EBADMODE == lResult) || (IOM_EBADARGS == lResult)
Set count = 0
No
C
(IOM_PENDING != lResult)
Yes
Get IOP from reclaimable Queue
No
Notify by callback
return lResult
Yes
Same as queue head
No
count = (iop->xferActual / channel->hDDC->blkSize);
Yes
IOM_COMPLETED == lResult
No
C
Set count = 0

Figure 5: MMCSD Driver Input/Output Flow Diagram - 1

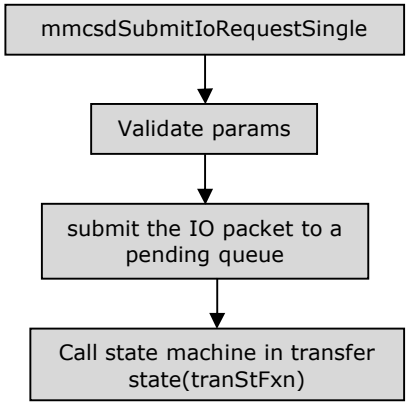


Figure 6: MMCSD Driver Input/Output Flow Diagram – 2

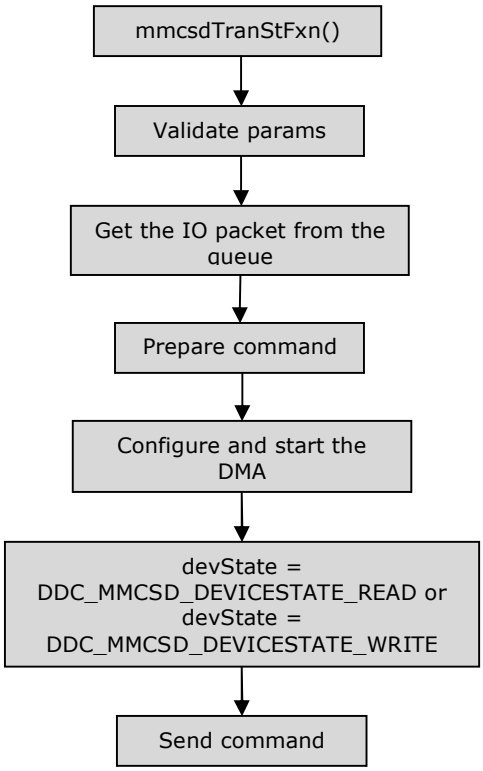


Figure 7: MMCSD Driver Input/Output Flow Diagram – 3

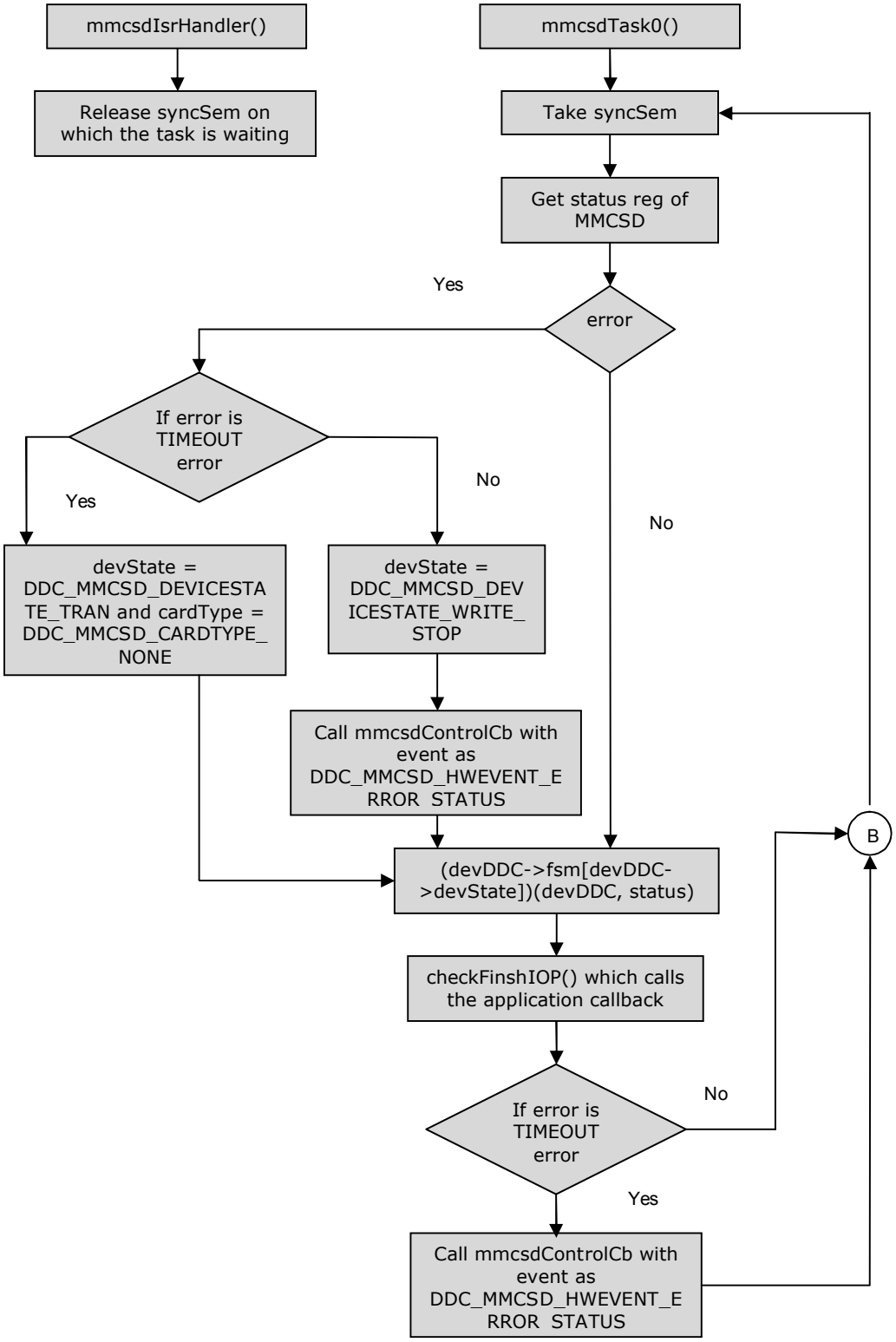


Figure 8: MMCSD Driver Input/Output Flow Diagram – 4

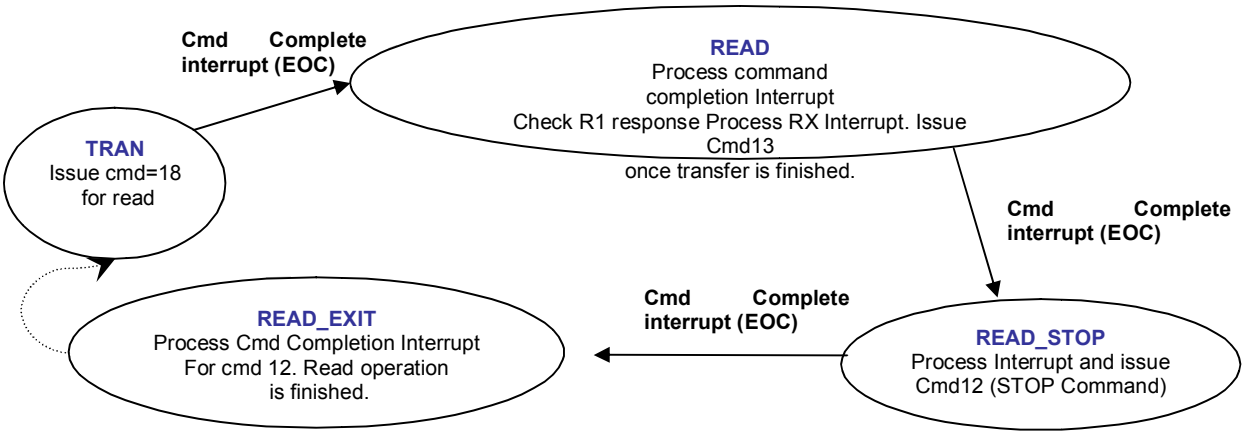


Figure 9: State Machine – Read operation

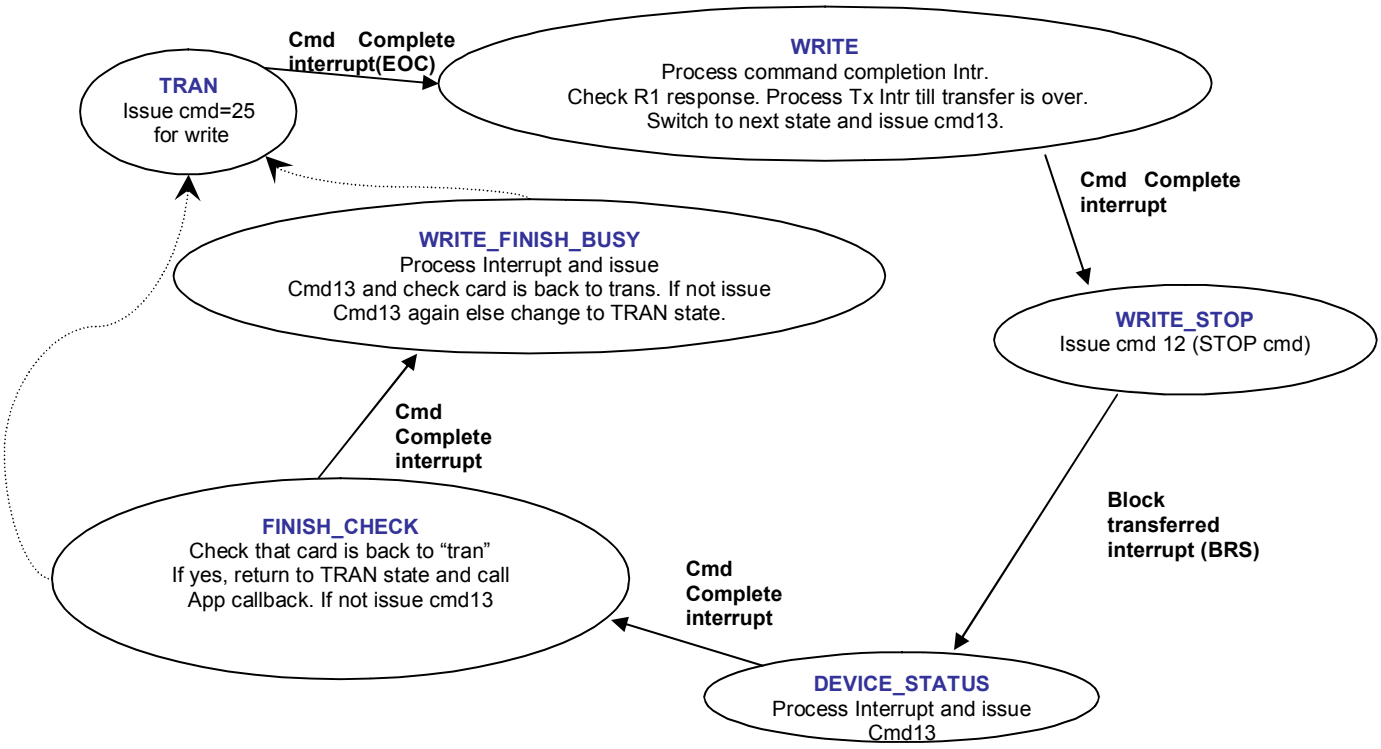


Figure 10: State Machine – Write operation

4.5 Driver Close

The application invokes the PSP_mmcsdCloseChannel() function to close the instance of the MMCSD device. Wait if any IOP processing is currently going on.

4.6 Driver Teardown

Following this operation, one is required to restart from beginning over a PSP_mmcsdDrvInit() call to bring driver back to life. On calling the PSP_mmcsdDrvDeInit() driver is de-initialized and delete any OS resources originally allocated.

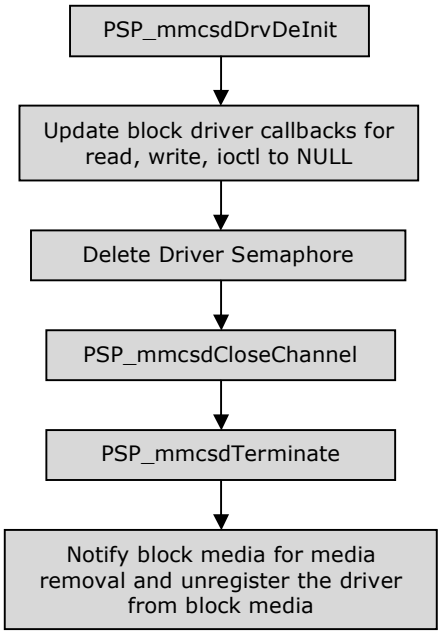


Figure 11: MMCSD Driver Teardown Flow Diagram - 1

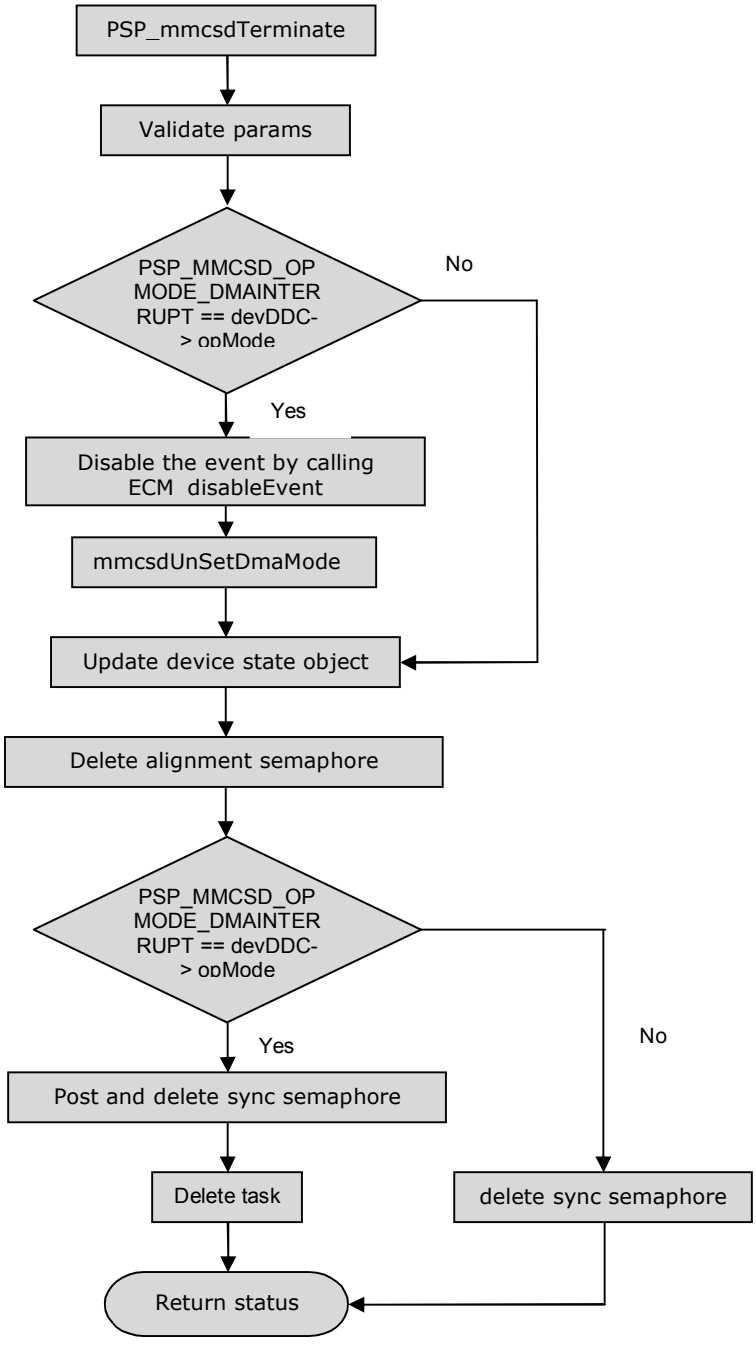


Figure 12: MMCSD Driver Teardown Flow Diagram - 2

5 Revision History

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