TEXAS INSTRUMENTS

SPI BIOS Device Driver

SPI Architecture/Design Document

IMPORTANT NOTICE

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		Applications	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address:
Texas Instruments
Post Office Box 655303, Dallas, Texas 75265

Document Version	Author(s)	Date	Comments
0.1	Shailesh Kumar Sharma	Dec 13,2006	Modified for DM648 from SPI.
0.2	JP	Dec 16, 2006	Cleanup
0.3	Shailesh Kumar Sharma.	Dec 26, 2006	Added flow of API.
0.4	Nagarjuna Kristam	Jan 29, 2007	Renamed DM64g to DM648
0.5	Shailesh Kumar Sharma	Feb 3, 2007	Updated for DM6446
0.6	Nagarjuna Kristam	June 1, 2007	Updated data structures with new design
0.7	Nagarjuna Kristam	June 15, 2007	Correcting version numbers for tools used and minor typo errors
0.8	Nagarjuna Kristam	July 6, 2007	Adding XDC tool information
0.9	Nagarjuna Kristam	January 17, 2008	Updated ddc_i2cObj structure members

Table of Contents

List Of Figures

1 System Context

The purpose of this document is to explain the device driver design for SPI peripheral used in DM648/C6452 SoC using DSP/BIOS operating system running on DSP 64+ joule. This driver is aimed at providing support for multiple SPI instances i.e. it can be used with other SPI supported SoC platforms.

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

API	Application Programmer's Interface
CSL	TI Chip Support Library – primitive h/w abstraction
IOM	Input/Output mini driver - TI terminology for portion of device driver that is specific to target OS. This constitutes "adaptation" of the generic DDC to identified target OS.
DDC	Device Driver Core - TI terminology for portion of device driver that is abstracted of any given OS
IP	Intellectual Property
ISR	Interrupt Service Routine
OS	Operating System
PAL OS	Platform Abstraction Layer.
SOC	System on chip
SPI	Serial Peripheral Interface

1.2 References

1.	sprue32_SPI.pdf	SPI Driver Documentation
2.	SPRU-404g.pdf	DSP/BIOS Driver Guide
3.	Dm648_spi_rdd.pdf	SPI RDD

1.3 Hardware

The SPI device driver architecture presented in this document is situated in the context of DM648/C6452 SoC targeted at Video Surveillance/ Packet Voice/ Catalog applications. The driver design is in the context of DSP/BIOS running on DSP 64x+ joule core. The following figure (Figure 1) shows DM648/C6452 Architecture shall be used for Video application.

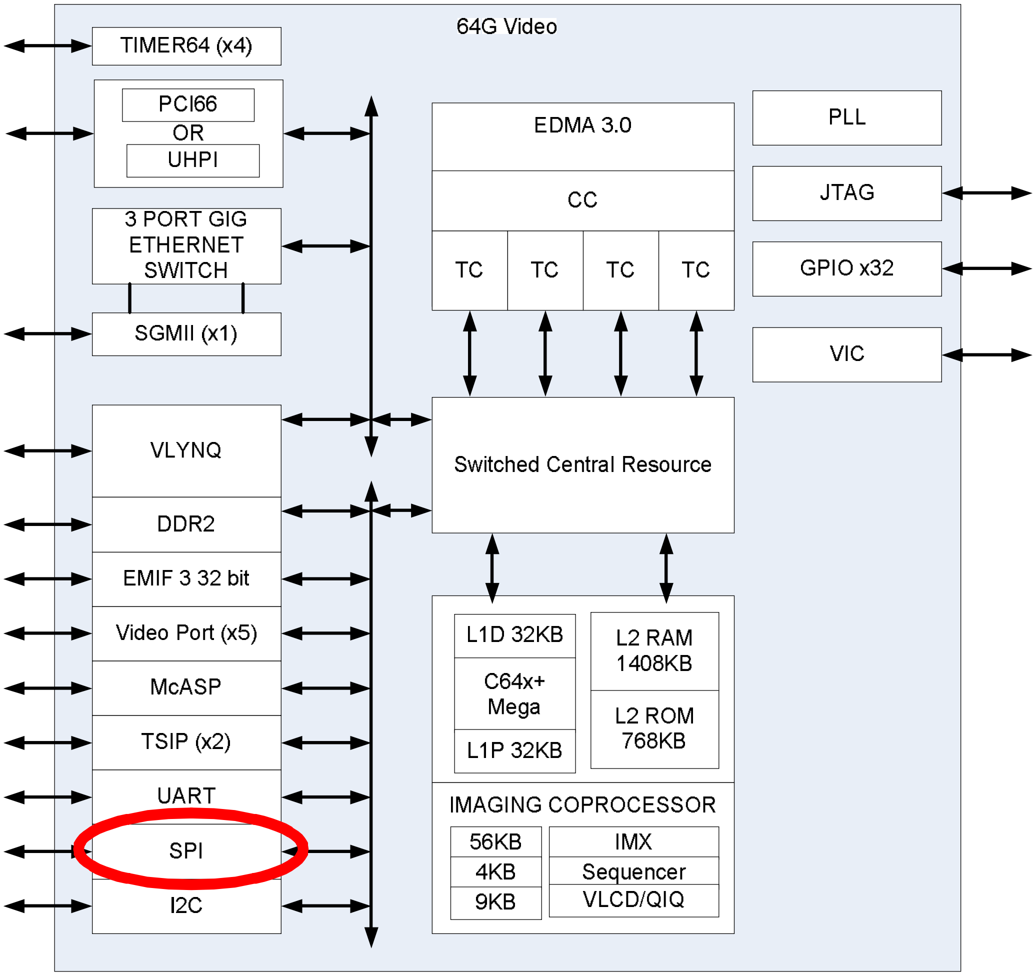


Figure 1 DM648/C6452 Block Diagram

The SPI module used in DM648/C6452 SOC core has the following blocks:

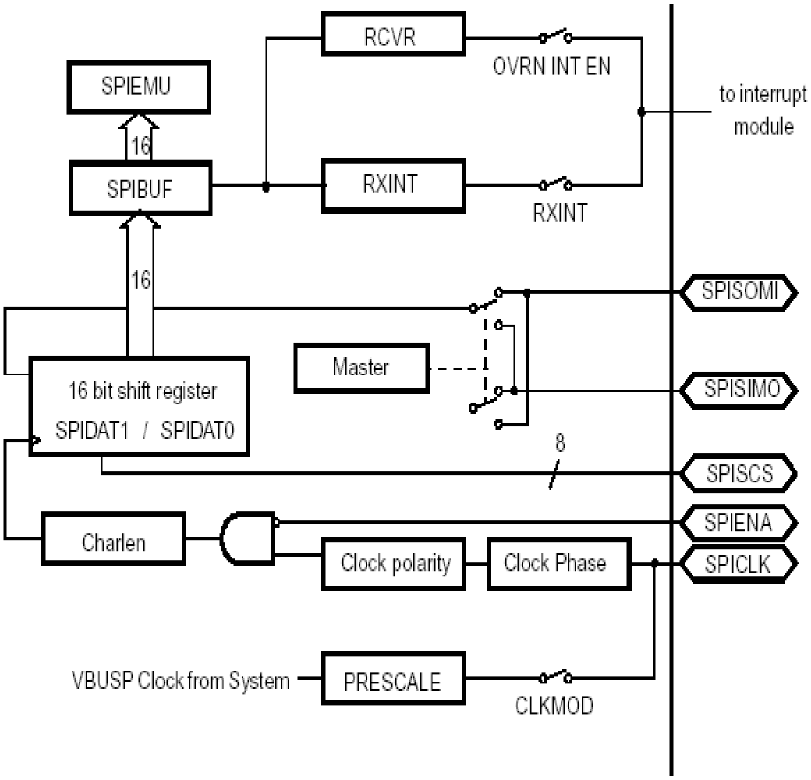


Figure 2 SPI HW diagram

1.4 Software

The SPI mini-driver discussed here is targeted at the DM648/C6452 device, running DSP/BIOS on the 64x+ DSP. However the SPI driver can also be ported to any other OS, with minimal modifications in the OS specific section of the driver. More details can be found in the later part of this section.

1.4.1 Operating Environment and dependencies

Details about the tools and the BIOS version that the driver is compatible with can be found in the system Release Notes.

1.4.2 System Architecture

The device driver described here is part of an IOM mini-driver. That is, it is implemented as the lower layer of a two layer device driver model and is a super set of all other driver layers. The upper layer is called the class driver and is the generic DSP/BIOS GIO module. The class driver provides an independent and generic set of APIs and services for a wide variety of mini-drivers and allows the application to use a common interface for I/O requests. Figure 3 shows the overall DSP/BIOS device driver architecture. For more information about the IOM device driver model, see the *DSP/BIOS Device Driver Developer's Guide (SPRU616).*

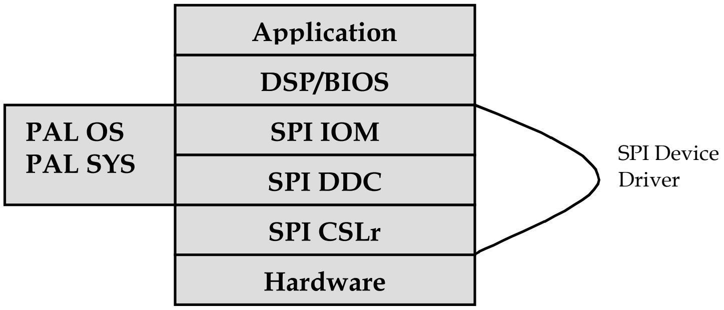


Figure 3 Device driver layer

This device driver can be used as a general-purpose stand-alone mini-driver to interface with the SPI peripheral on DM648/C6452.

Please refer PSP framework manual to get to know more details about the various device driver layers.

1.5 Component Interfaces

In the following subsections, the interfaces implemented by each of the sub-component are specified. Refer to SPI device driver API reference documentation for the complete details of the APIs.

1.5.1 IOM Interface

The IOM constitutes the Device Driver Manifest to Application. The user may not look into IOM interface, especially the upper-edge services exposed to the Application/OS. All other interfaces discussed later in this document are more of interest to people developing/maintaining the device driver.

The IOM can be modified to re-target Driver and/or customize to specific Apps framework by doctoring the upper-edge services.

The *spi_mdBindDev ()* populates static settings in driver object creates the necessary interrupt handler, attaches the Driver Core interfaces. All these operations in effect, constitute the "loading" of SPI Driver implementation. The spi_mdUnbindDev () constitutes the "Un-loading" of the SPI driver. The IOM mini-driver implements the following API interfaces to the class driver.

S.No	**IOM Interfaces**	**Description**
1	spi_mdBindDev ()	Allocates and configures the SPI port specified by devId
2	spi_mdUnbindDev ()	Closes the SPI device from use.
3	spi_mdCreateChan ()	Creates a communication channel in specified mode to communicate data between the application and the SPI device instance.
4	spi_mdDeleteChan ()	Frees a channel and all its associated resources.
5	spi_mdControlChan ()	Implements the IOCTLS for SPI IOM mini driver.
6	spi_mdSubmitChan ()	Submit an I/O packet to a channel for processing.

1.5.2 DDC Interface

DDC implements the core device driver layer and it provides standard abstract interfaces to the upper layers as per the PSP framework standards architecture.

The DDC layer APIs of SPI driver can be called directly from the application or from the OS adaptation layer. So this can be ported to any OS without any modification.

The following basic interfaces are implemented and exposed to the IOM layer by the DDC layer of SPI driver.

S.No	**DDC Interfaces**	**Description**
1	PSP_spiCreate ()	Initialize/Setup the SPI

		hardware with the given configuration parameters.
2	PSP_spiDelete ()	Does the reverse of *PSP_spiCreate.*
3	PSP_spiOpen ()	Configure SPI's TX/RX DMATCU channels.
4	PSP_spiClose ()	Does the reverse of *PSP_spiOpen.*
5	PSP_spiIoctl ()	Perform input/output control on SPI Hardware.
6	PSP_spiTransceive()	Submits IOP requests to perform transceive transfer.

1.5.3 CSLR Interface

The CSL register interface (CSLr) provides register level implementations. CSLr is used by the DDC to configure SPI registers. CSLR is implemented as a header file that has CSLR macros and register overlay structure.

1.6 Design Philosophy

This device driver is written in conformance to the DSP/BIOS IOM device driver model and handles communication to and from the SPI hardware.

1.6.1 The Port and Channel Concept

The IOM model provides the concept of the *Port* and *Channel* for the realization of the device and its communication path as a part of the driver implementation. The *Port Object* maintains the state of the SPI device or an instance. The *port* can also be called as *instance* or *device* and the names can be used interchangeably. DM648/C6452 contains one instance of SPI, and the driver for this needs to maintain only one port object. The port object contains placeholders for all channel objects for TX and RX, in this implementation it is only one. The following figure shows the generic port-channel-hardware mapping for SPI driver.

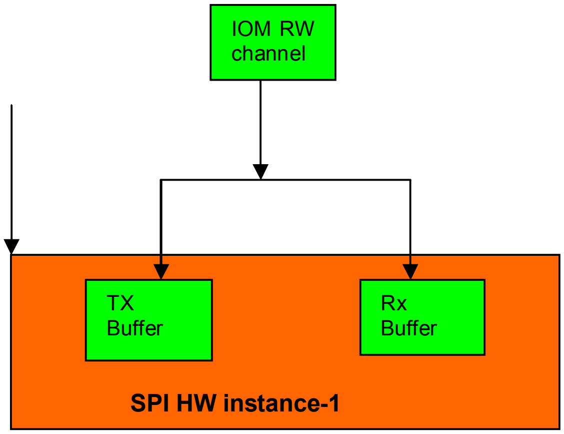


Figure 4 Port and Channel Object

1.7 Design Constraints

SPI mini-driver imposes the following constraint(s).

- Synchronous Read/Write interface for data transfer.
- Driver shall not support dynamically changing modes between Interrupt and Polled modes of operation.
- Driver shall not support the Slave mode of operation.
- Driver shall not support the DMA mode of operation.

2 SPI Driver Software Architecture

This section details the data structures used in the SPI mini-driver and the interface it presents to the GIO layer. A diagrammatic representation of the mini driver functions is presented and then the usage scenario is discussed in some more details.

2.1 Static View

2.1.1 Functional Decomposition

The driver is designed keeping a concept of port and channel in mind. The instance of SPI is treated as a device, which each can have a single read/write channel for DM648/C6452 SoC.

This driver uses two internal data structures, a port object and a channel object, to maintain its state during execution. The SPI peripheral needs the port instance to maintain its state. The channel object holds the IOM channel state during execution. These are explained in greater detail in the following *Data Structures* sub-section. The following figure shows the static view of DM648/C6452 SPI driver.

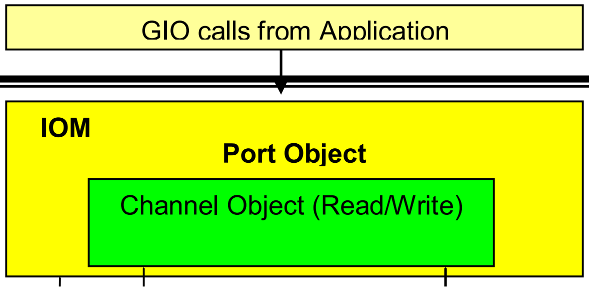


Figure 5 SPI driver static view

2.1.2 Data Structures

The mini-driver employs the PortObj and ChannelObj structures to maintain state of the port and channel respectively.

In addition, the driver has two other structures defined – the device params and channel params. The device params structure is used to pass on data to initialize the driver during DSP-BIOS initialization. The channel params structure is used to specify required characteristics while creating a channel. The IO params structure is used to specify memory buffers to do IO transfers.

The following sections provide major data structures maintained by IOM, DDC and PSP interface. For more details about IOM and DDC data structures and their usage can be found in the API reference guide.

2.1.2.1 The Port Object (IOM)

S.No	Structure Elements ((spi*_portObj))*	Description
1	*portNumber*	Preserve port or instance number of

		SPI
2	*State*	*Current state of the port object*.
3	*Chan[]*	Holds all channel objects for this port.
4	*Port object*	Pointer to store ddc object.

2.1.2.2 The Channel Params – PSP interface

The channel parameter structure is passed when creating a channel. This channel parameter contains the following mention params.

S.No	**Structure Elements (*PSP_spiChanParams)***	**Description**
1	mode	Channel mode of operation: Input or Output.
2	Port	Pointer to device port spi*_portObj* structure
3	*cbFxn and cbArg*	IOM callback function and its argument.
4	*ddc handle*	To store the channel handle passed from DDC layer.

2.1.2.3 The Driver Object (DDC)

S.No	**Structure Elements**	**Description**

	(DDC_spiObject*)*	
1	*mode*	Mode of operation.
2	*intNum*	Interrupt number
3	*spiRegs*	CSL registers handle
4	*spiHWconfig*	Hardware configuration params variable of SPI.
5	*moduleInputClkFreq*	Input clock frequency
7	*spiBusFreq*	SPI bus frequency
8	*spidat0*	Flag to select the pin mode configuration.
9	*numOpens*	Numbers of time channels can be opened.
10	*instanceId*	Instance id
11	*state*	State of driver
12	*transBuffer*	Data buffer handler for transceive operation.
13	*transcieveFlags*	Flags to select the transceive operation.
14	*devBusySem*	Semaphore to block other tasks in accessing SPI
15	*completionSem*	Semaphore to block driver during interrupt mode
16	*currError*	Current transmission error
17	*currFlags*	Current flags
18	*currBuffer*	Current transaction buffer
19	*currBufferLen*	Current transaction buffer length
20	*edmaSpiHandle*	Edma handle .
21	*charLength16Bits*	To check character length
22	*dmaChaAllocated*	Flag to indicate DMA channel allocation status
23	*csHighPolarity*	Chip Select Polarity. Either active high/Low. By default it is set to Active Low. TRUE = Active High

2.1.2.4 The Channel Object (DDC)

S.No	**Structure Elements (DDC_spiDriverObject)**	**Description**
1	*callBack*	Callback function
2	*appHandle*	Callback function argument
3	*spiObj*	SPI object handle.

2.1.2.5 The Device Params

The file **psp_spi.h** has the **PSP_spiConfig** data structure that is passed as SPIdevParams to initialization function of the driver. The params are explained below:

S.No	**Structure Elements (spi_*chanObj*)**	**Description**
1	opMode	Operational mode of the driver
2	moduleInputClkFreq	Input Frequency to SPI Module.
3	spiBusFreq	Clock used to calculate the output data rate of SPI.
4	spiHWCfgData	Hardware config data.

2.1.2.6 The Device Hardware Configuration Params

S.No	**Structure Elements (spi_*chanObj*)**	**Description**
1	intrLevel	Arm interrupt level either 0 or 1.
2	pinOpModes	SPI pin opMode params.
3	delay	Delay between two transfers.
4	masterOrSlave	Master /slave selection..
5	clkInternal	Clock direction, either External or Internal.
6	enableHighZ	Enable pin status either tristated or grounded

7	csDefault	Default chip-select
8	ConfigDatafmt[]	Data config format.
9	charLength	Character length.
10	lsbFirst	Data transfer direction.
11	parityEnable	Parity enable.
12	Polarity	To enable or disable polarity

2.2 Dynamic View

2.2.1 The Execution Threads

The SPI device driver operation involves following execution threads:

BIOS thread: Function to load and un-load SPI driver will be under BIOS OS initialization.

Application thread: Creation of channel, Control of channel, deletion of channel and processing of SPI frame data will be under application thread.

Interrupt context: Processing TX/RX interrupts, and Error interrupts and notifies to application through Call back function.

2.2.2 Sync IO mechanism

POLLED Mode:

Check is done to see if job is complete, if not a suitable interval of time is spent in "delay" looping – once the data transfer is completed successfully, driver is returned to application with appropriate status information.

INTERRUPT Mode:

This is very similar to above case; except for waits occurring in form of pending for Semaphore being available and SPI DDC being energized through' Interrupt thread of control. Since we pend on Semaphore here, it is possible for other application threads to run when we wait here for IO transaction to complete.

2.2.3 Functionali Decomposition

2.2.3.1 Driver Creation

The sequence diagram below depicts the creation phase of the BIOS SPI driver. The DEV_createDevice which calls spi_mdBindDev to create a driver instance. That means device creation is done dynamically.

While at the DDC level, create and init phases of driver instance are clearly demarcated. Regardless, once this phase is complete, the basic driver data structures and setups are complete and ready for formally opening device to perform IO.

The DEV_createDevice is expected to invoke spi_mdBindDev (), way up in the application startup phase, in a central driver initialization function.

The spi_mdBindDev () performs book-keep functions on the driver and allocates memory for instance data structures.

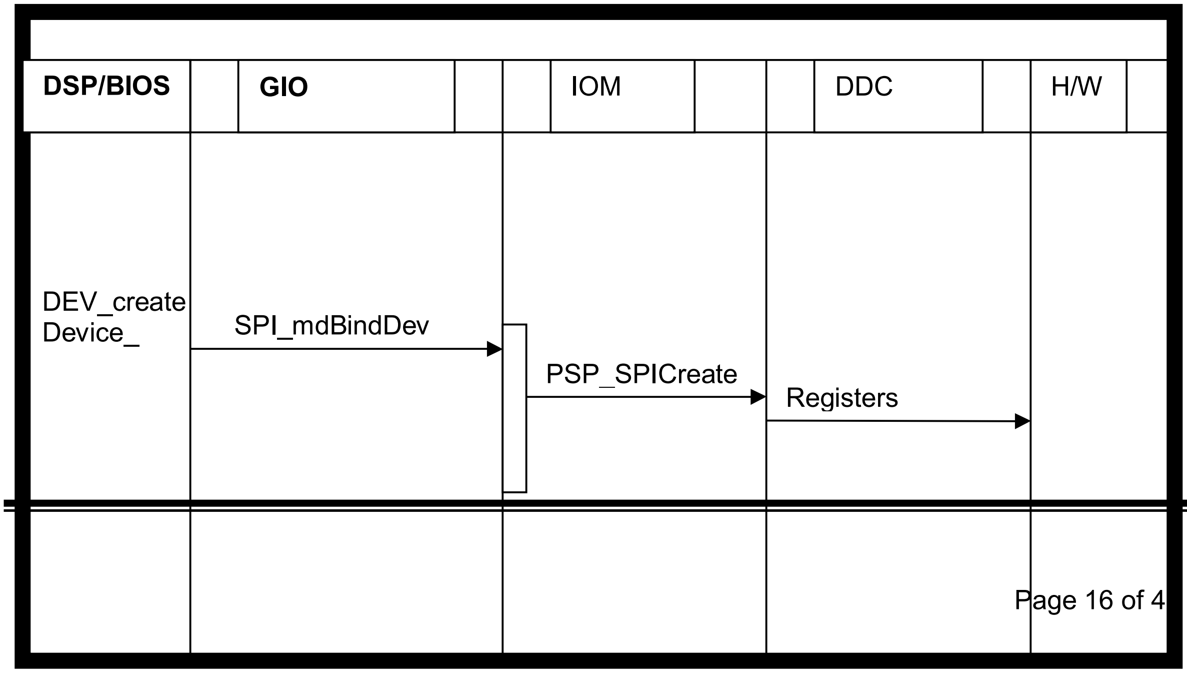


Figure 6 Driver Create Flow Diagram

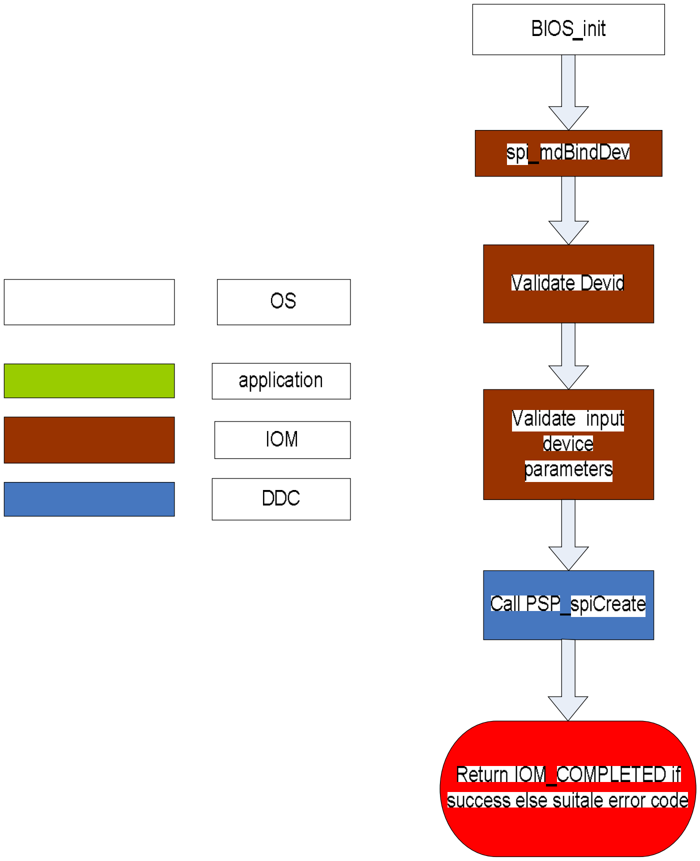

OS
application
IOM
DDC
BIOS_init
spi_mdBindDev
Validate Devid
Validate input device parameters
Call PSP_spiCreate
Return IOM_COMPLETED if success else suitale error code

Figure 7 Driver Create detailed Flow Diagram -1.

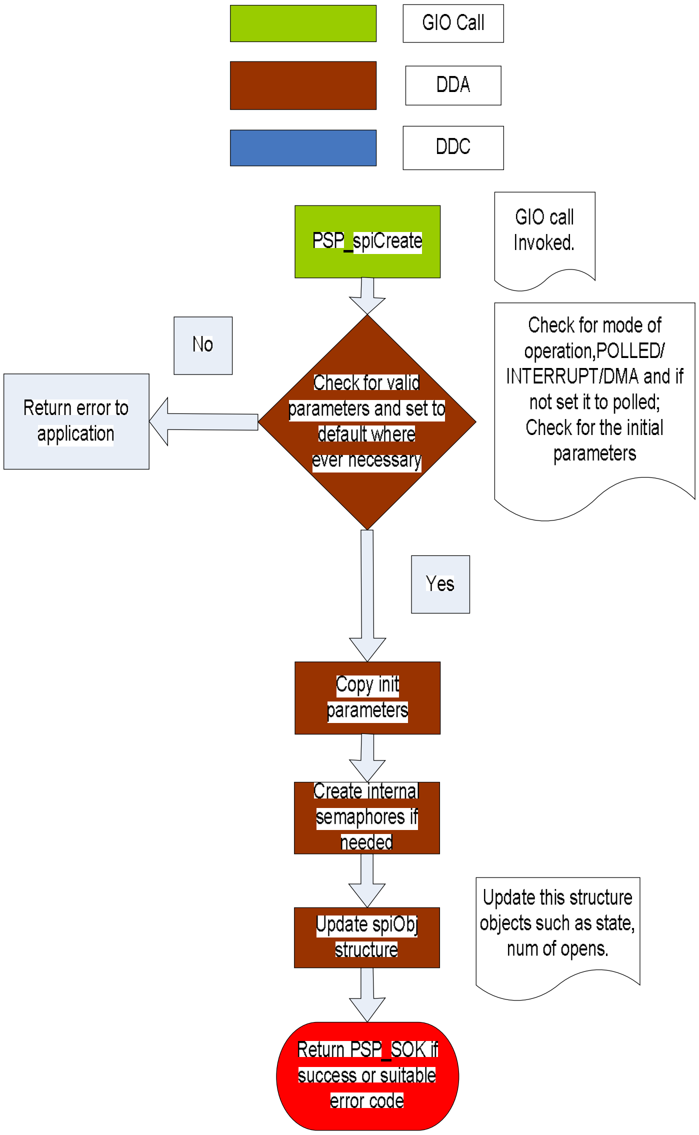

GIO Call
DDA
DDC
PSP_spiCreate
GIO call Invoked.
No
Return error to application
Check for valid parameters and set to default where ever necessary
Check for mode of operation,POLLED/ INTERRUPT/DMA and if not set it to polled; Check for the initial parameters
Yes
Copy init parameters
Create internal semaphores if needed
Update spiObj structure
Update this structure objects such as state, num of opens.
Return PSP_SOK if success or suitable error code

Figure 8 Driver Create Detailed Flow Diagram - 2

DEV_createDevice invoke spi_mdBindDev ( ), way up in the application startup phase in a central driver initialization function.

The spi_mdBindDev ( ) performs bookkeeping functions on the driver and allocates memory for instance data structures. It attaches the DDC create functions for use later during actual initialization of each device instance.

2.2.3.2 *Driver Open*

When the application calls the GIO_Create () which calls spi_mdCreateChan (), driver entry point, the PSP_spiOpen function is invoked to provide a handle for the further operations on the SPI. The callback is registered with DSP/BIOS as well as the application call back. The driver is ready to accept Read/Write jobs.

Following is the flow diagram for the GIO_Create functionality.

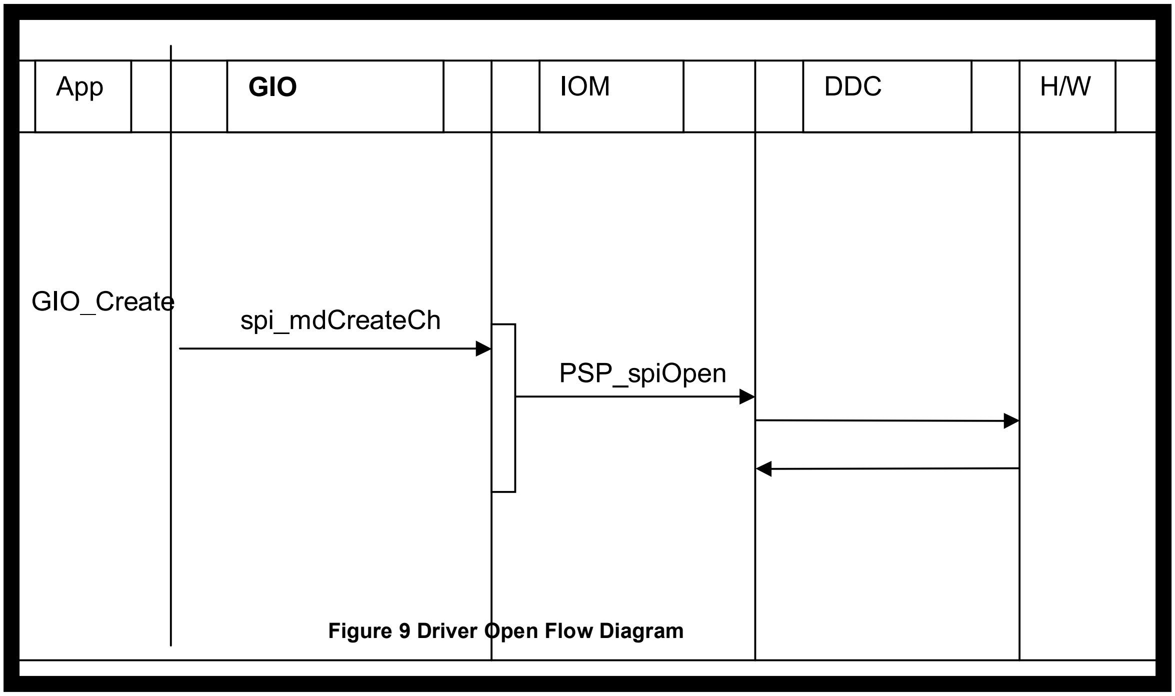


Figure 9 Driver Open Flow Diagram

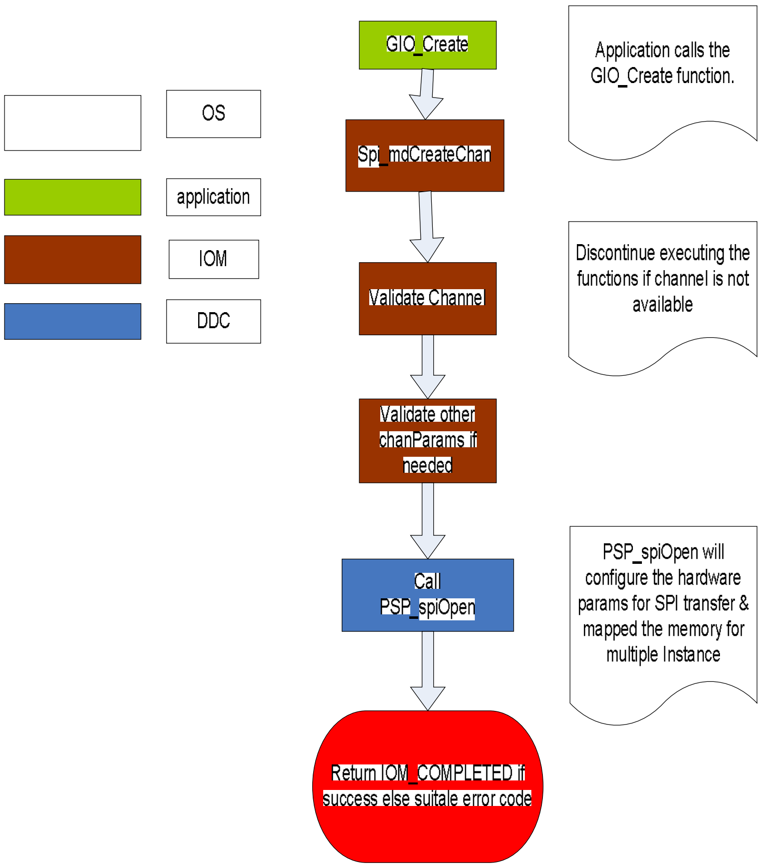

OS
application
IOM
DDC
GIO_Create
Spi_mdCreateChan
Validate Channel
Validate other chanParams if needed
Call PSP_spiOpen
Return IOM_COMPLETED if success else suitale error code
Application calls the GIO_Create function.
Discontinue executing the functions if channel is not available
PSP_spiOpen will configure the hardware params for SPI transfer & mapped the memory for multiple Instance

Figure 10 Driver Open Detailed Flow Diagram - 1

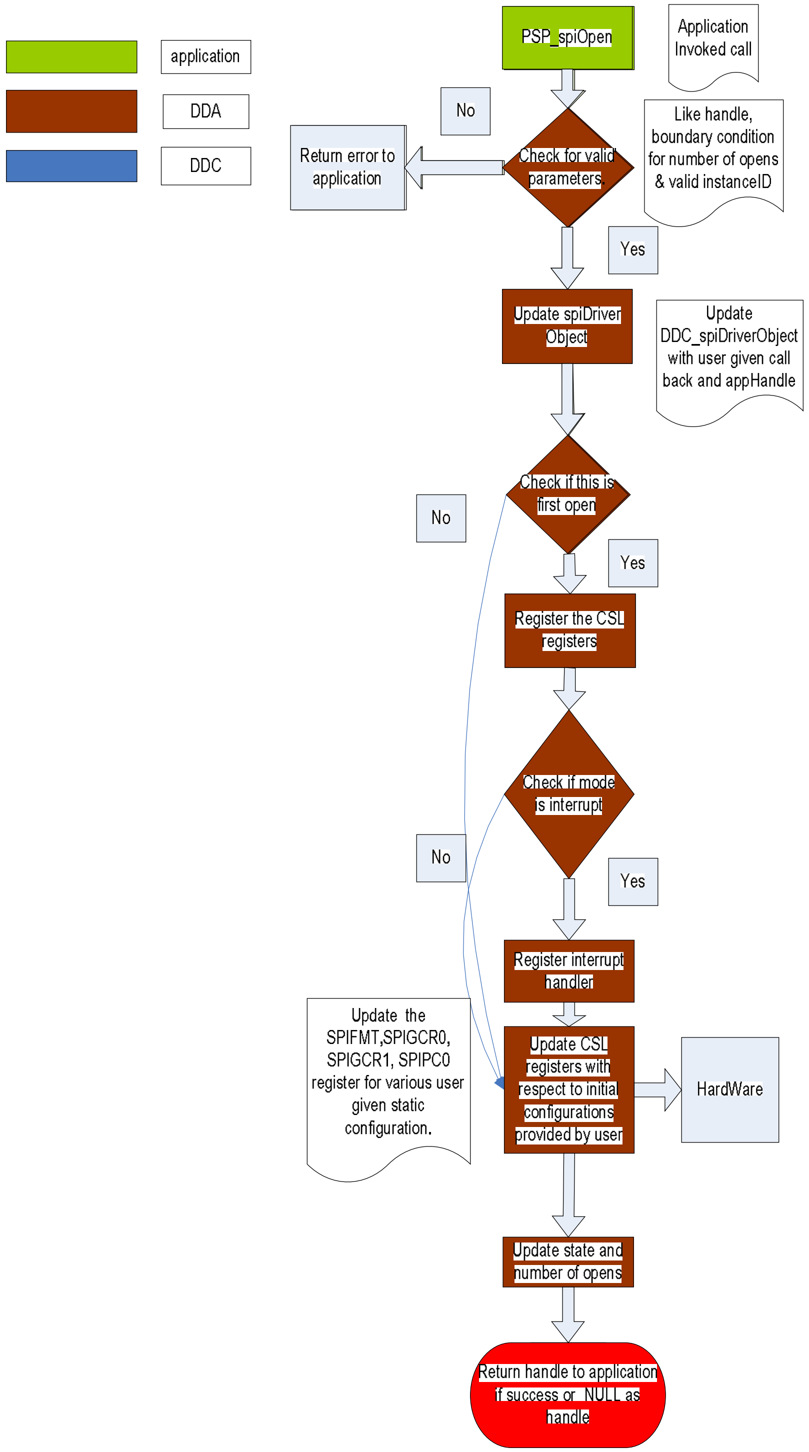

application
DDA
DDC
PSP_spiOpen
Application Invoked call
No
Return error to application
Check for valid parameters
Like handle, boundary condition for number of opens & valid instanceID
Yes
Update spiDriver Object
Update DDC_spiDriverObject with user given call back and appHandle
Check if this is first open
No
Yes
Register the CSL registers
Check if mode is interrupt
No
Yes
Register interrupt handler
Update the SPIFMT,SPIGCR0, SPIGCR1, SPIPC0 register for various user given static configuration.
Update CSL registers with respect to initial configurations provided by user
HardWare
Update state and number of opens
Return handle to application if success or NULL as handle

Figure 11 Driver Open Detailed Flow Diagram – 2

2.2.4 IO Control

The SPI Driver provides spi_mdControlChan ( ) to set/get common configuration parameters on the driver at run time through the corresponding DDC IOCTL function, PSP_spiIoctl. Moreover IOCTL commands that are device specific or that require action on the part of the device driver call the driver's IOCTL.

Following is the flow diagram for the above functionality.

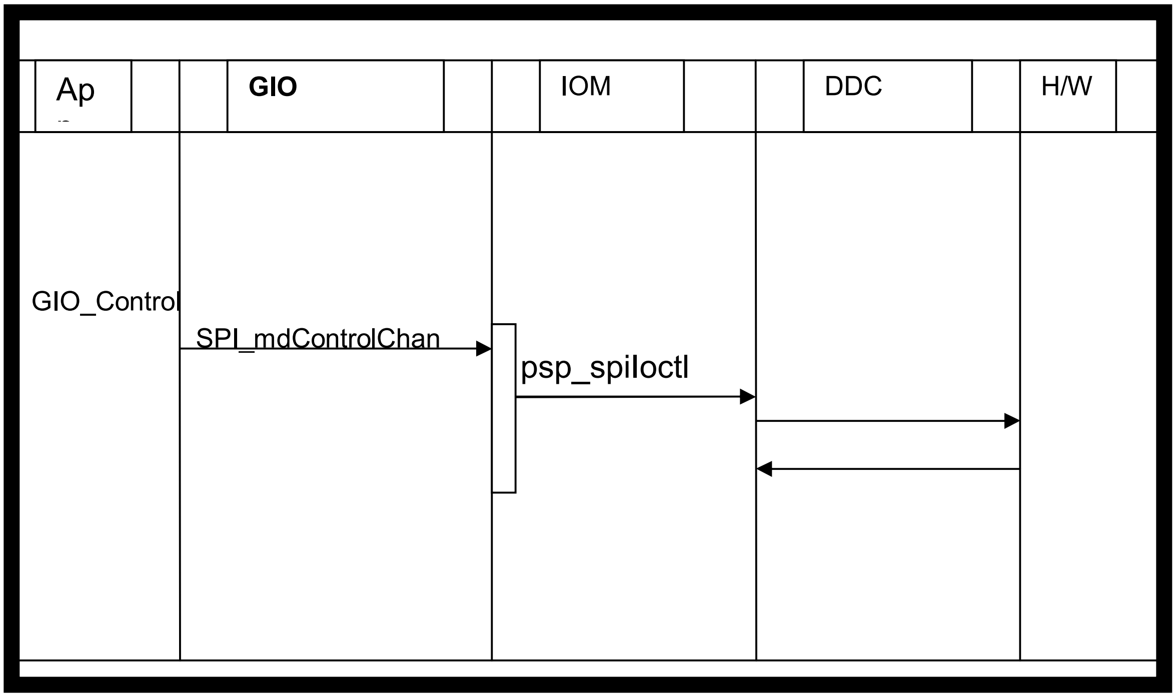
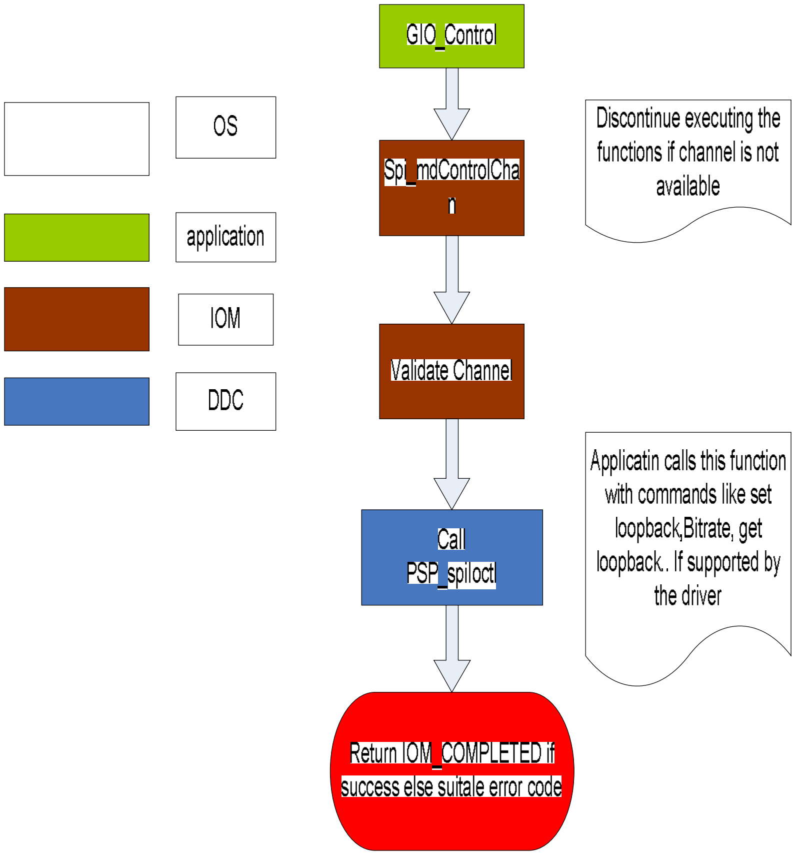


Figure 12 Driver IOCTL Detailed Flow Diagram.



Discontinue executing the functions if channel is not available

Applicatin calls this function with commands like set loopback,Bitrate, get loopback.. If supported by the driver

Figure 13 Driver IOCTL Detailed Flow Diagram - 1

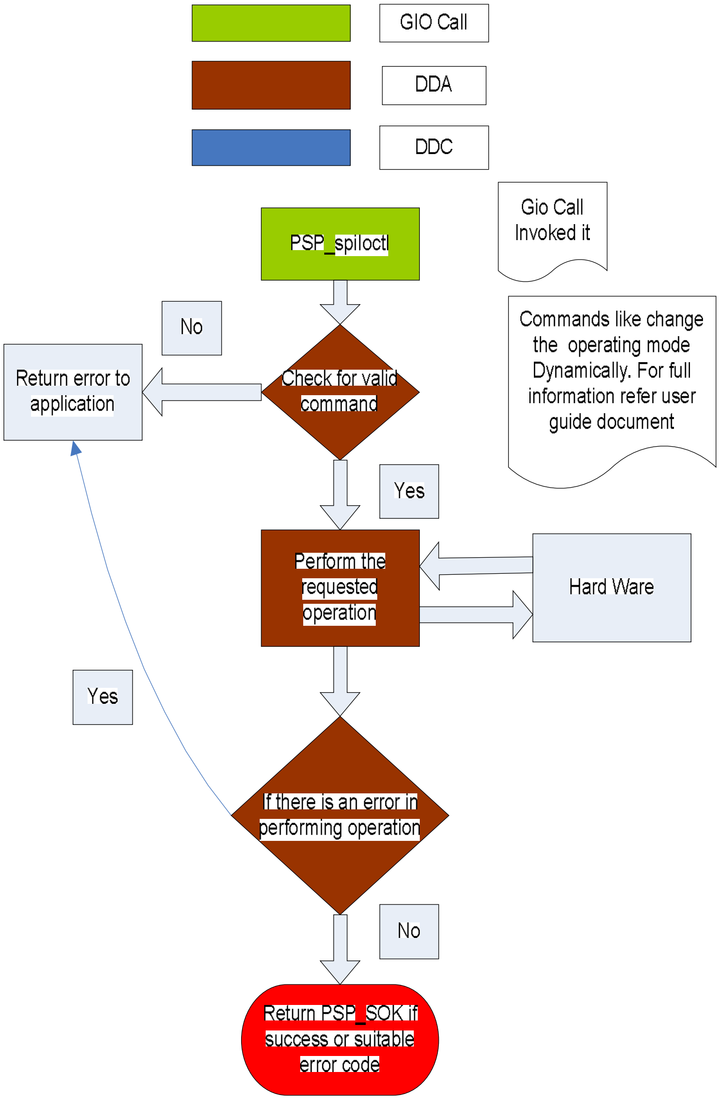

GIO Call
DDA
DDC
Gio Call Invoked it
PSP_spiIoctl
Commands like change the operating mode Dynamically. For full information refer user guide document
No
Return error to application
Check for valid command
Yes
Perform the requested operation
Hard Ware
Yes
If there is an error in performing operation
No
Return PSP_SOK if success or suitable error code

Figure 14 IOCTL Detailed Flow Diagram - 2

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

2.2.5 IO Access

The application will access SPI driver IOM API spi_mdSubmitChan through interface functions from DSP/BIOS. These functions are registered on the DSP/BIOS during the driver initialization.

Following completion of IO, the packet is recycled back to the free IOP's pool in the IOM.

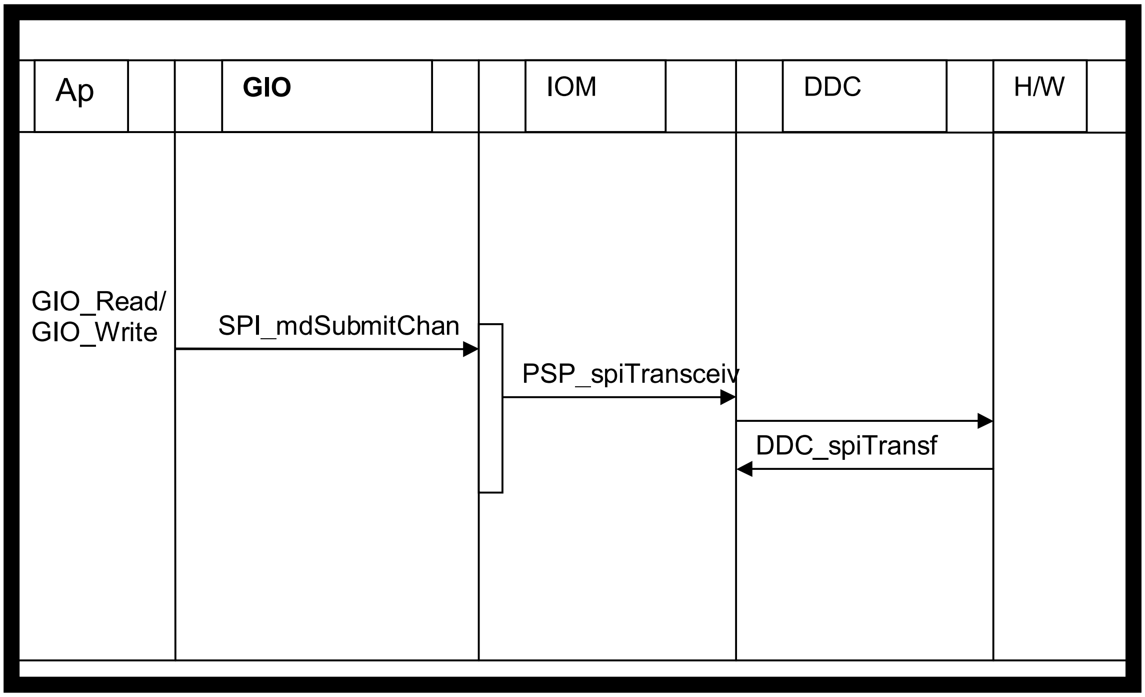


Figure 15 Driver transfer overview.

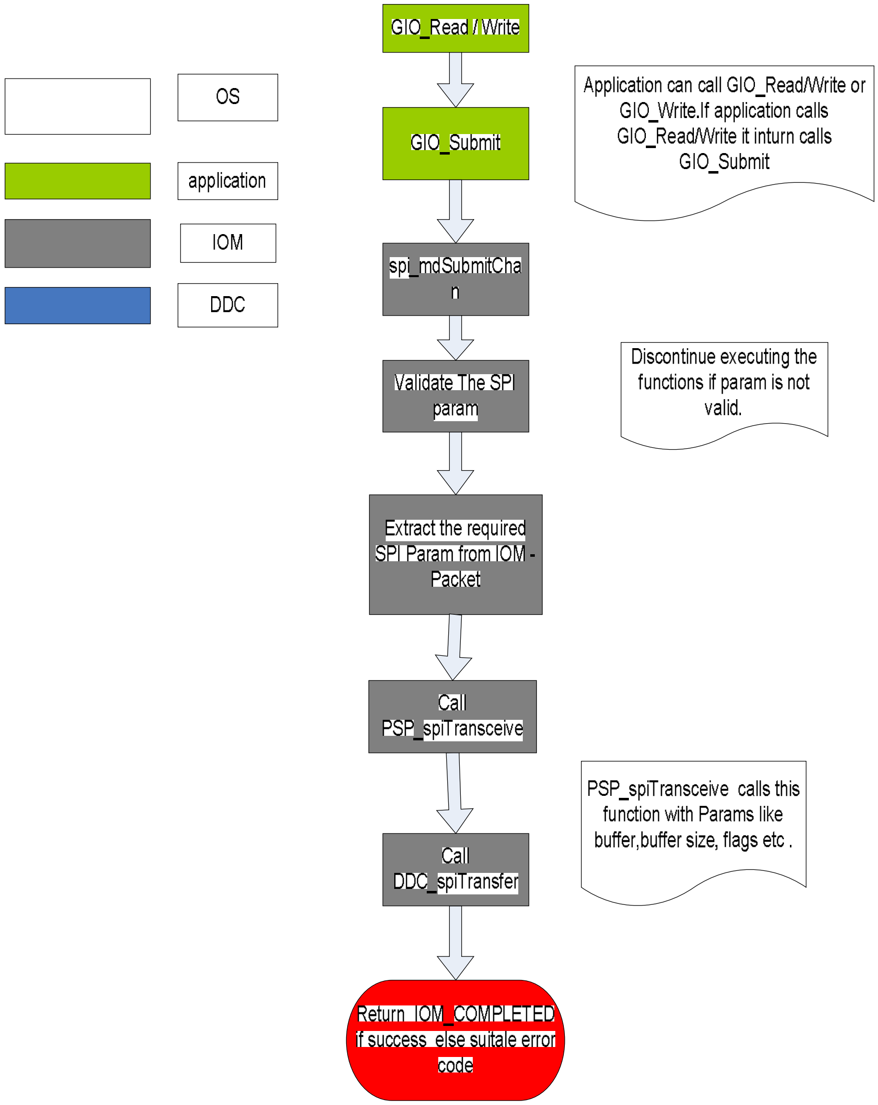

OS
application
IOM
DDC
GIO_Read / Write
GIO_Submit
spi_mdSubmitChan
Validate The SPI param
Extract the required SPI Param from IOM - Packet
Call PSP_spiTransceive
Call DDC_spiTransfer
Return IOM_COMPLETED if success else suitale error code
Application can call GIO_Read/Write or GIO_Write.If application calls GIO_Read/Write it inturn calls GIO_Submit
Discontinue executing the functions if param is not valid.
PSP_spiTransceive calls this function with Params like buffer,buffer size, flags etc .

Figure 16 Driver transfer Detailed Flow Diagram – 1.

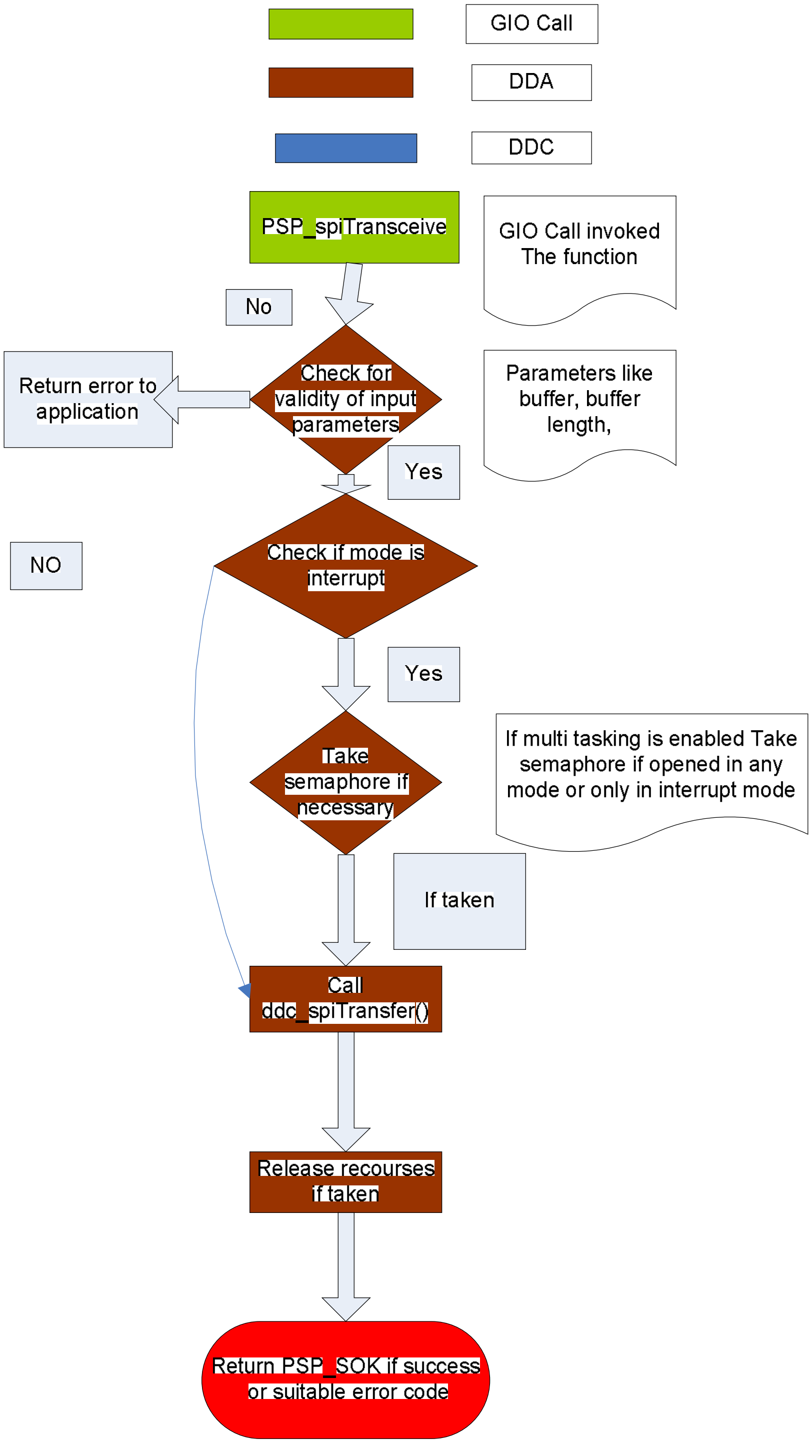

GIO Call
DDA
DDC
PSP_spiTransceive
GIO Call invoked The function
No
Return error to application
Check for validity of input parameters
Parameters like buffer, buffer length,
Yes
NO
Check if mode is interrupt
Yes
Take semaphore if necessary
If multi tasking is enabled Take semaphore if opened in any mode or only in interrupt mode
If taken
Call ddc_spiTransfer()
Release recourses if taken
Return PSP_SOK if success or suitable error code

Figure 17 Driver transfer Detailed Flow Diagram – 2.

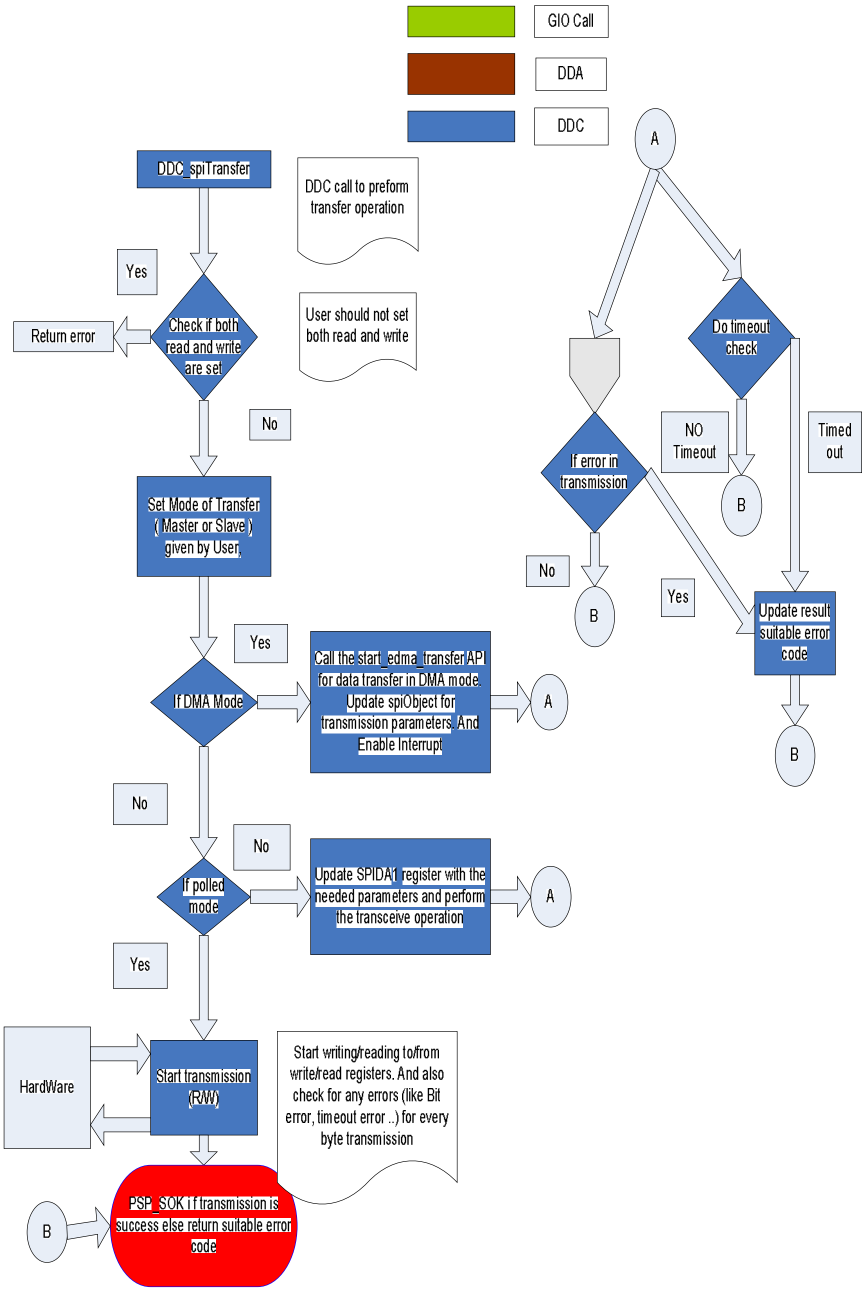

GIO Call
DDA
DDC
DDC_spiTransfer
DDC call to preform transfer operation
Yes
Return error
Check if both read and write are set
User should not set both read and write
No
Set Mode of Transfer (Master or Slave) given by User,
Yes
If DMA Mode
Call the start_edma_transfer API for data transfer in DMA mode. Update spiObject for transmission parameters. And Enable Interrupt
A
No
No
If polled mode
Update SPIDAT1 register with the needed parameters and perform the transceive operation
A
Yes
HardWare
Start transmission (R/W)
Start writing/reading to/from write/read registers. And also check for any errors (like Bit error, timeout error ..) for every byte transmission
B
PSP_SOK i f transmission is success else return suitable error code
A
If error in transmission
No
B
Yes
Do timeout check
NO Timeout
B
Timed out
Update result suitable error code
B

Figure 18 Driver transfer Detailed Flow Diagram – 3.

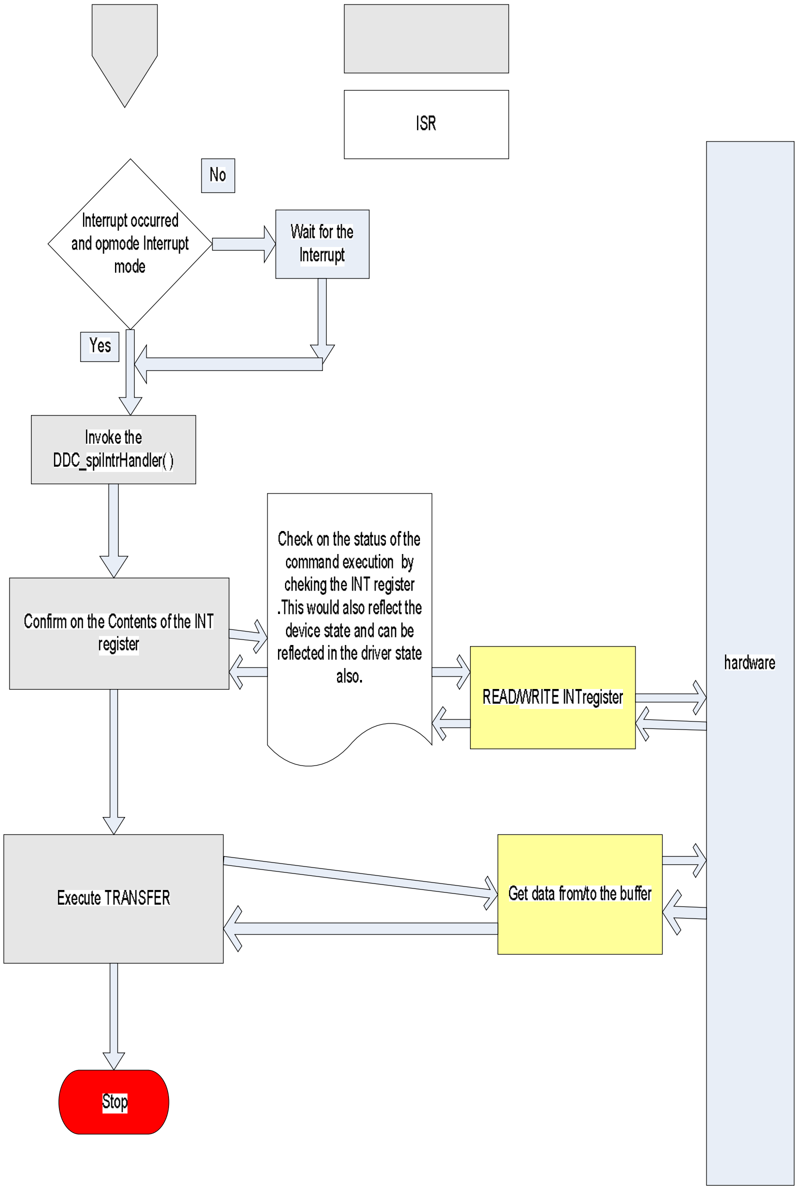

ISR
No
Interrupt occurred and opmode Interrupt mode
Wait for the Interrupt
Yes
Invoke the DDC_spiIntrHandler( )
Check on the status of the command execution by cheking the INT register .This would also reflect the device state and can be reflected in the driver state also.
Confirm on the Contents of the INT register
READ/WRITE INTregister
hardware
Execute TRANSFER
Get data from/to the buffer
Stop

Figure 19 Driver transfer in ISR Detailed Flow Diagram

2.2.6 Driver Close

The application invokes the spi_mdUnBindDev () function to close the channel of the SPI device.

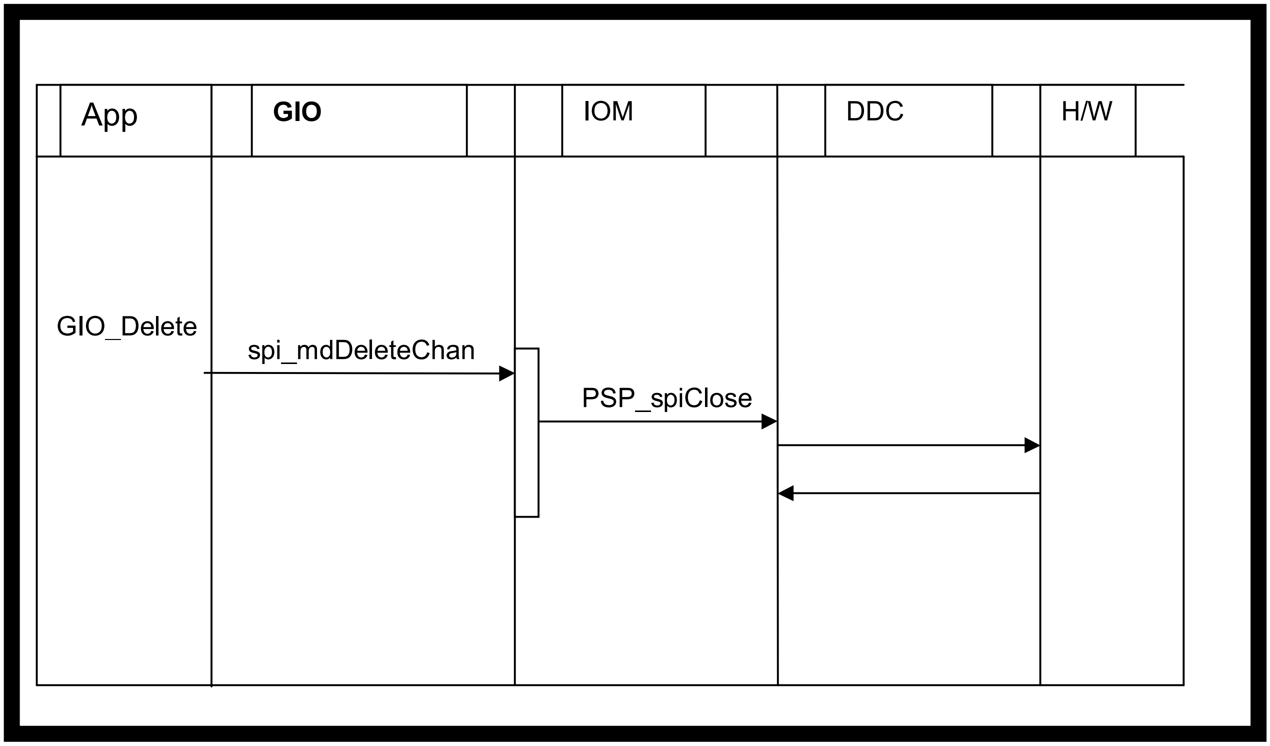


Figure 20 Driver Close Detailed Flow Diagram.

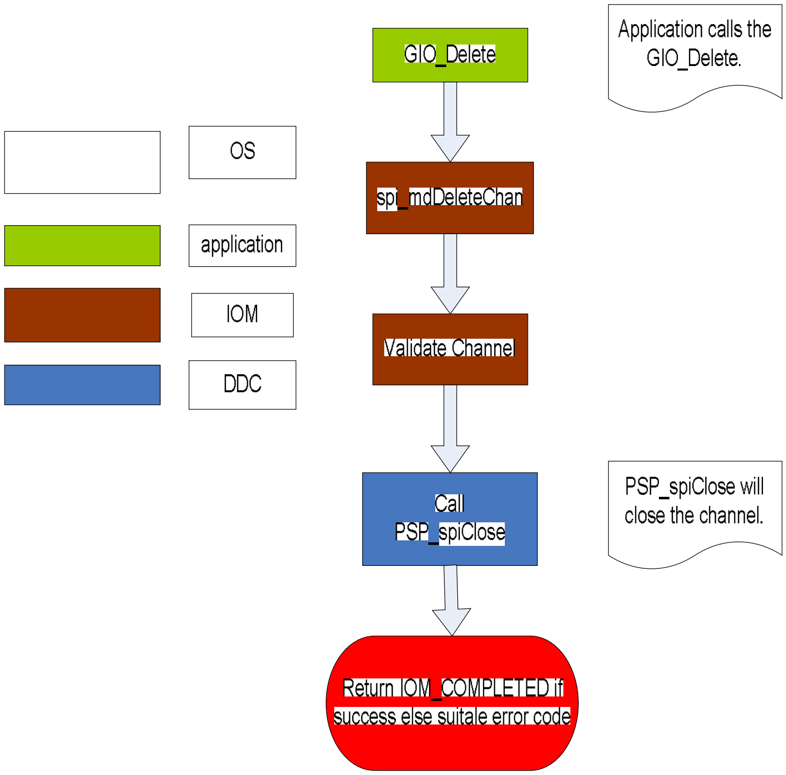

GIO_Delete
Application calls the GIO_Delete.
OS
application
IOM
DDC
spi_mdDeleteChan
Validate Channel
Call PSP_spiClose
PSP_spiClose will close the channel.
Return IOM_COMPLETED if success else suitale error code

Figure 21 Driver Close Detailed Flow Diagram – 1.

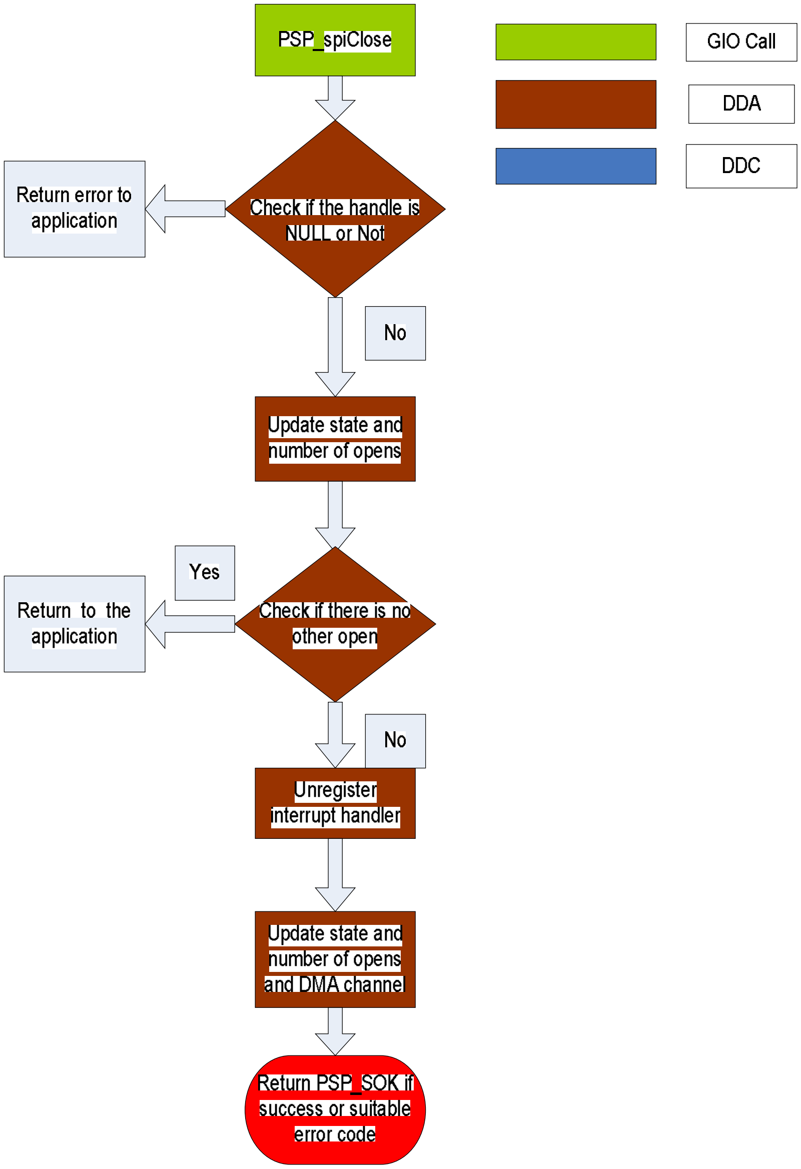

PSP_spiClose
GIO Call
DDA
DDC
Return error to application
Check if the handle is NULL or Not
No
Update state and number of opens
Yes
Return to the application
Check if there is no other open
No
Unregister interrupt handler
Update state and number of opens and DMA channel
Return PSP_SOK if success or suitable error code

Figure 22 Driver Close Detailed Flow Diagram – 2.

2.2.7 Driver Teardown

Following the call spi_mdUnBindDev （ ） one is required to restart from beginning over spi_mdBindDev （） call to bring driver back to life. The driver de-initialize and delete functions de-initialize the SPI DDC and delete if any OS resources originally allocated through spi_mdBindDev （ ） by calling PSP_spiDelete function.

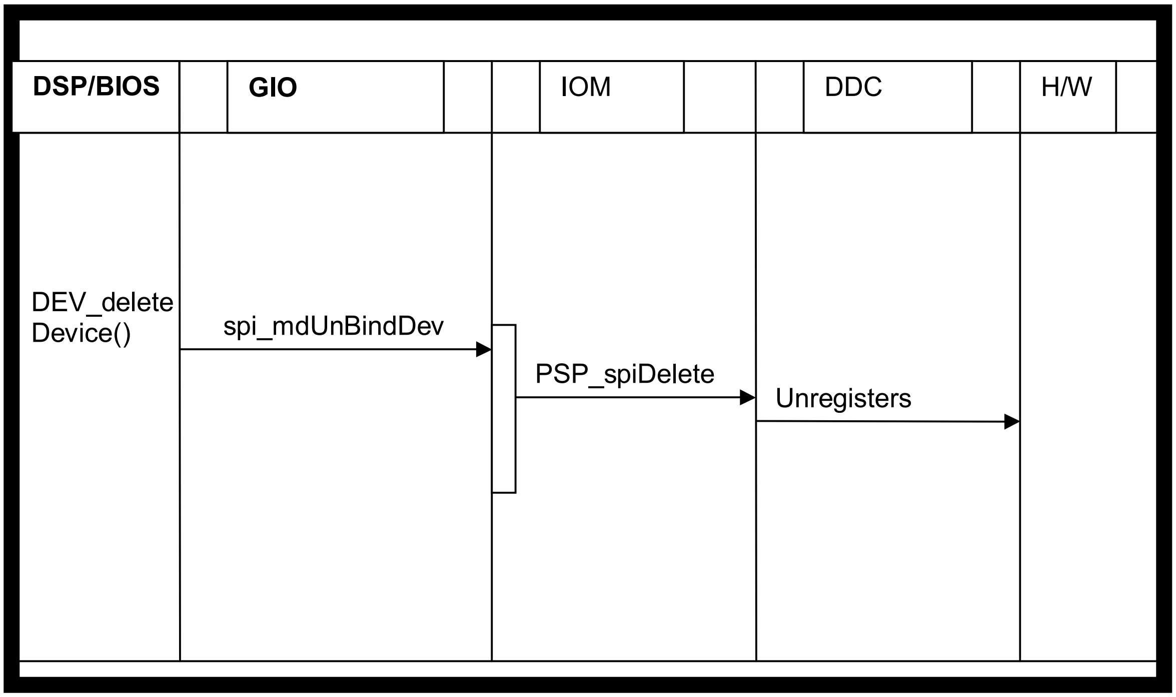


Figure 23 Driver Delete Flow Diagram.

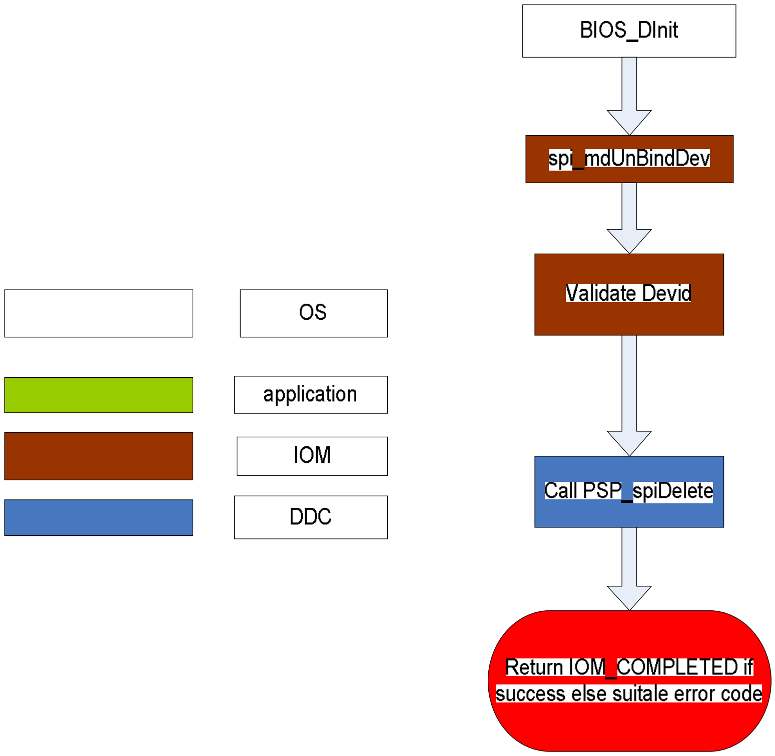


Figure 24 Delete Detailed Flow Diagram – 1.

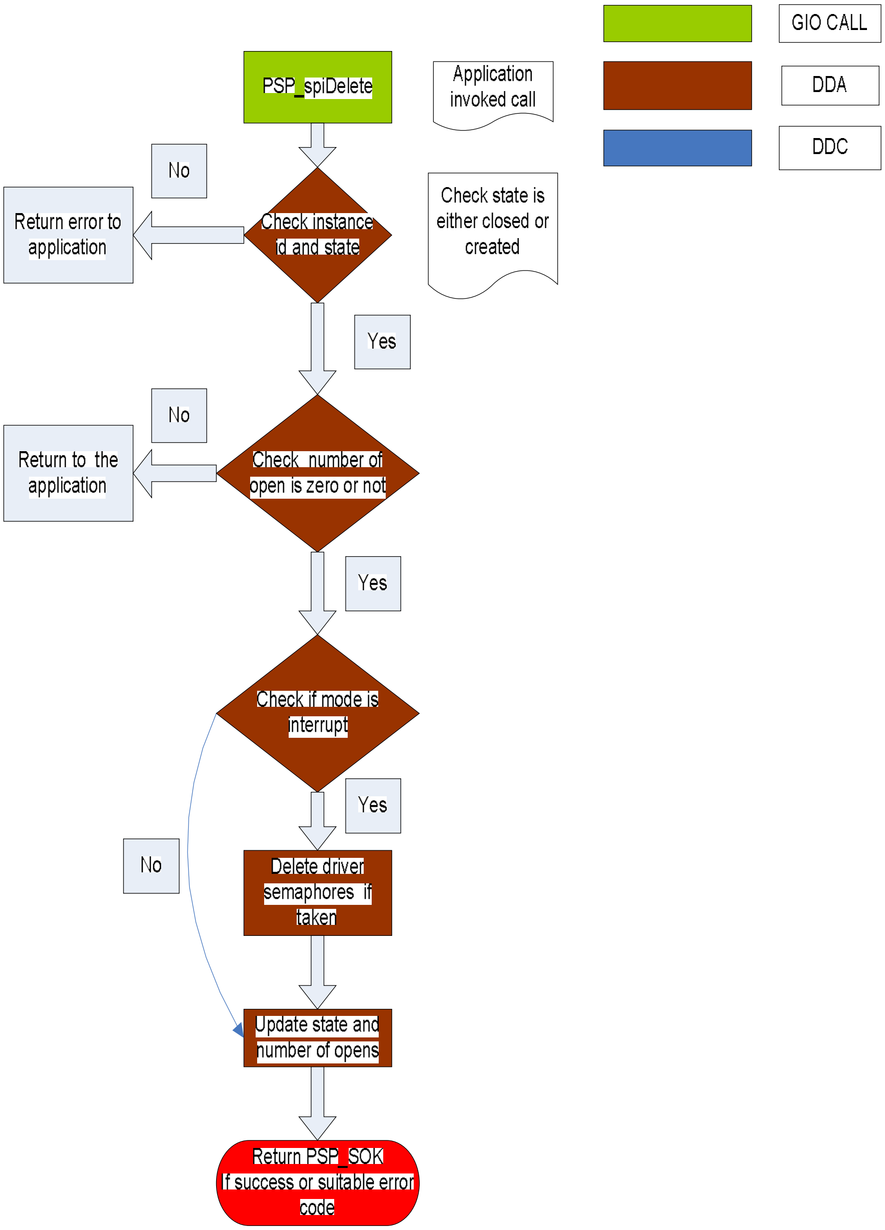

GIO CALL
DDA
DDC
PSP_spiDelete
Application invoked call
No
Return error to application
Check instance id and state
Check state is either closed or created
Yes
No
Return to the application
Check number of open is zero or not
Yes
Check if mode is interrupt
Yes
No
Delete driver semaphores if taken
Update state and number of opens
Return PSP_SOK if success or suitable error code

Figure 25 Driver Delete Detailed Flow Diagram – 2.

3 APPENDIX A – IOCTL commands

S.No	Error Code	Description
1	PSP_SPI_IOCTL_CANCEL_PENDING_IO	To cancel pending IO requests if any in the transmission

4 APPENDIX B – Error Codes

S.No	Error Code	Description
1.	PSP_SPI_TIMEOUT_ERR	Enable pin response to master cause timeout error.
2.	PSP_SPI_PARITY_ERR	Data transferred and received mismatch cause parity error.
3	PSP_SPI_DESYNC_ERR	Desynchronization of slave from master due to timing or clock glitch leads to Desync error.
4.	PSP_SPI_BIT_ERR	Data transfer at sampling point mismatch leads to bit error.
5.	PSP_SPI_RECEIVE_OVERRUN_ERR	Receive overflow error due to data received before first bytes get read.