Texas Instruments

DSP/BIOS I2C Device Driver

Architecture/Design Document

Platform Software Group

Revision History

Document Version	Author(s)	Date	Comments
0.1	Nagarjuna Kristam	November 6, 2006	Initial version – created for DRA44X
0.2	Nagarjuna Kristam	December 11, 2006	- Added dynamic IO mechanism support in the design. - Details added/enhanced as per MT-TI review on Nov08. - Updated Error codes, IOCTLs (nov19) - Divided HW setup structure into three types (nov21)
0.3	Nagarjuna Kristam	December 16, 2006	Updated the document as per the review comments
0.4	Nagarjuna Kristam	January 29, 2007	Renamed DM64g to DM648
0.5	Nagarjuna Kristam	June 15, 2007	Corrected version numbers for tools used
0.6	Nagarjuna Kristam	July 6, 2007	Adding XDC toll information
0.7	Nagarjuna Kristam	July 6, 2007	Removed hw even call back support
0.8	Nagarjuna Kristam	October 22, 2007	Updated XDC and BIOS versions
0.9	Nagarjuna Kristam	November 14, 2007	Updated for DM6437/C6424 and DM648/C6452

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments

Post Office Box 655303

Dallas, Texas 75265

Table of Contents

List Of Figures

1 System Context

The purpose of this document is to explain the device driver design for I2C peripheral, using DSP/BIOS operating system running on DSP 64+ joule.

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
IOM	Input/Output Mini-driver
DDC	Device Driver Core - TI terminology for portion of device driver that is abstracted of any given OS
I2C	Inter Integrated Circuit
FSM	Finite State Machine
GIO	Generic Input/Output
ICDRR	I2C Data receive register
ICDXR	I2C Data transmit register
ICMDR	I2C mode register
ICRSR	I2C receiver shift register
ICXSR	I2C transmit shift register
IP	Intellectual Property
ISR	Interrupt Service Routine
LLC	Low Level Controller
OS	Operating System
PSP	Platform Support Package
SCL	Serial Clock line
SDA	Serial Data Line
SOC	System On Chip

1.2 Related Documents

1.	SPRU4.4g.pdf	DSP/BIOS Driver Developer's Guide
2.	spruek8_I2C.pdf	I2C specification

1.3 Hardware

The I2C device driver design is in the context of DSP/BIOS running on DSP 64x+ joule core.

The I2C module core used here has the following blocks:

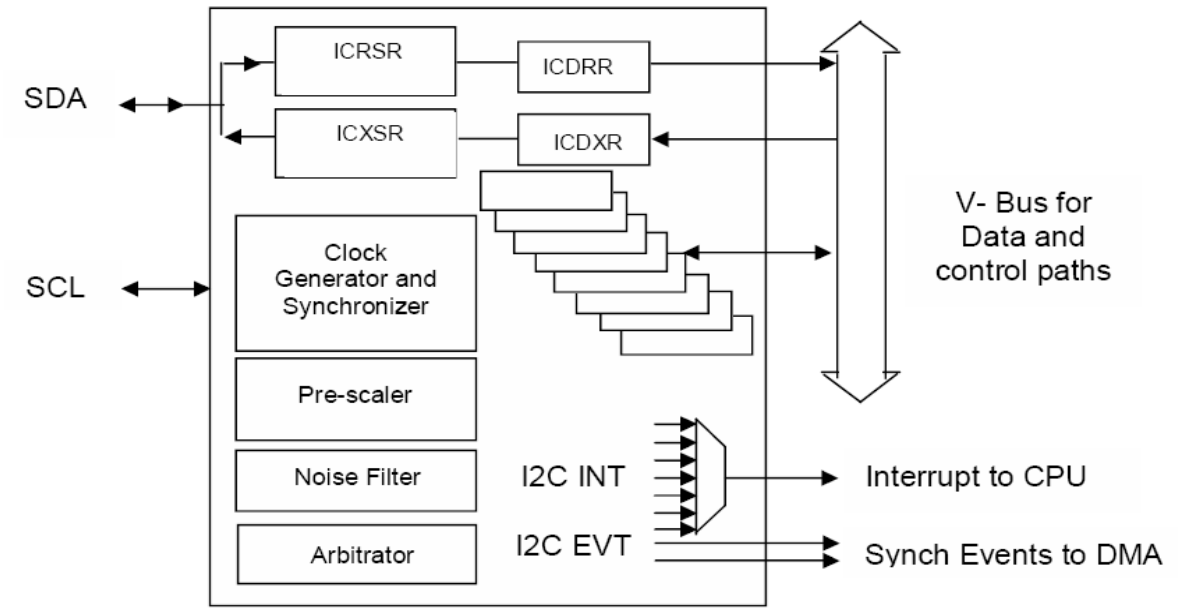

Figure 1: I2C device

1.4 Software

The I2C mini-driver discussed here is running DSP/BIOS on the 64x+ DSP. However the I2C 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 Architechture

The Application would invoke the driver routines through the GIO Calls APIs, whiich is OS based adaptation layer and device drivers are accessed by the applications for performing I/O using BIOS through the above mentioned GIO calls.

IOM is the component that exposes the driver Core to the OS. But the implementation of IOM is in such a way that it forms a wrapper or gel layer in between GIO and actual DDC layer. Thus we can say that IOM acts as a DDA adaptation layer.

DDC is the driver core which actually performs the device specific operations. This works as an independent driver also, by exposing all its PSP API's to application. CSLR is the layer that is embedded inside DDC and has a direct access to the hardware.

Figure 2 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). The rest of the document elaborates on the architecture of the Device driver by TI.

The block diagram below shows the overall system architecture.

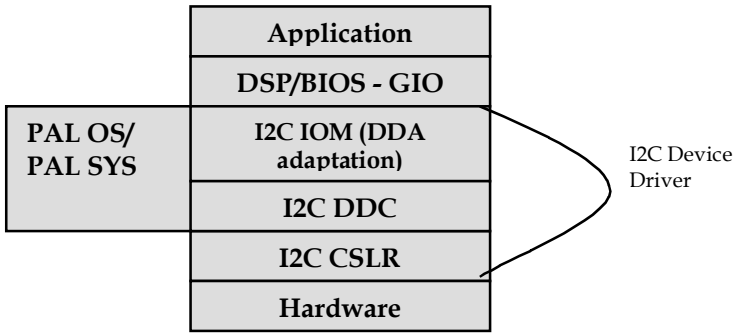

Figure 2: System Architecture

This device driver can be used as a general-purpose stand-alone mini-driver to interface with the I2C peripheral.

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 I2C device driver API reference documentation for complete details on 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 *i2c_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 I2C Driver implementation. The i2c_mdUnbindDev () constitutes the "Un-loading" of the I2C driver. The IOM mini-driver implements the following API interfaces to the class driver.

S.No	IOM Interfaces	Description
1	i2c_mdBindDev ()	Allocates and configures the I2C port specified by devid.
2	i2c_mdUnbindDev ()	Removes the I2C device from use.
3	i2c_mdCreateChan ()	Creates a communication channel in specified mode to communicate data between the application and the I2C device instance.
4	i2c_mdDeleteChan ()	Frees a channel and all its associated resources.
5	i2c_mdControlChan ()	Implements the IOCTLS for I2C IOM mini driver.
6	i2c_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 I2C 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 I2C driver.

S.No	DDC Interfaces	Description
1	PSP_i2cCreate ()	Initialize/Setup the I2C with the given configuration parameters.
2	PSP_i2cDelete ()	Does the reverse of *PSP_i2cCreate*.
3	PSP_i2cOpen ()	Configure I2C's H/W configuration as mentioned in create and registers interrupt handler if opened in interrupt mode channels.
4	PSP_i2cClose ()	Does the reverse of *PSP_i2cOpen*.
5	PSP_i2cIoctl ()	Perform input/output control on I2C Hardware.
6	PSP_i2cTransfer ()	Read/Write a buffer of data to the specified slave address.

1.5.3 CSL Interface

The CSL register interface (CSLr) provides register level implementations. CSLr is used by the DDC to configure I2C 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 I2C 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 I2C driver provides one bi-directional (IOM read/write) channel in order to perform IO operations. The *Port Object* maintains the state of the I2C device or an instance. The *port* can also be called as *instance* or *device* and the names can be used interchangeably. The port object contains placeholders for all channel objects for TX and RX, in this implementation it is 5(and this can be changed using a variable). The following figure shows the generic port-channel-hardware mapping for I2C driver.

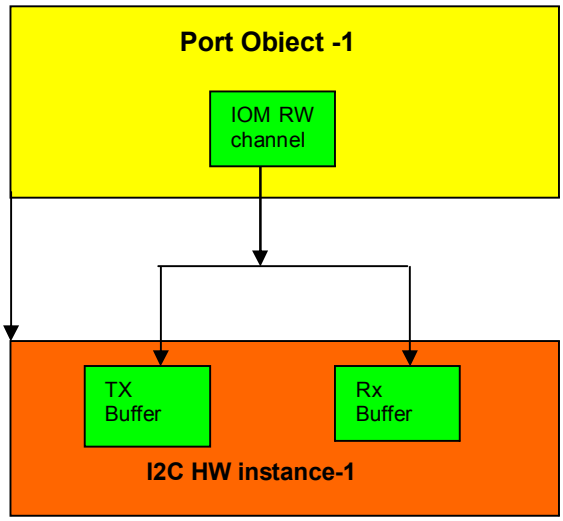

Figure 3: Port and Channel Object

1.6.2 Design Constrains

I2C mini-driver imposes the following constraint(s).

- I2C driver supports only synchronous mode of operation to the application.
- I2C driver shall only work with polled and Interrupt mode of operation. No DMA support is available in this implementation
- I2C driver shall not support dynamically changing modes between Interrupt and Polled modes of operation.

2 I2C Driver Software Architecture

This section details the data structures used in the I2C 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.

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

2.1 Static View

2.1.1 Functional Decomposition

The driver is designed keeping a device, also called port and channel concept in mind. The instance of I2C is treated as a device, which each can have a single read/write channel.

This driver uses two internal data structures, a port object and a channel object, to maintain its state during execution. The I2C 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 I2C driver.

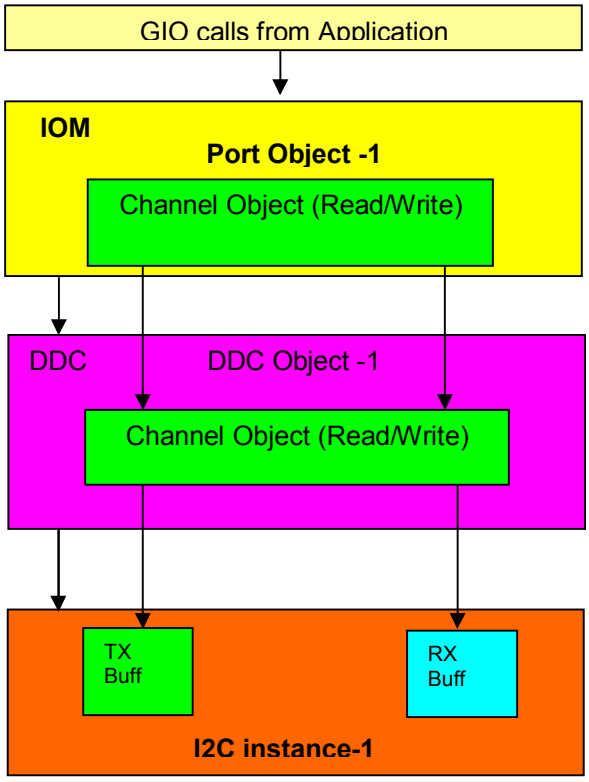

Figure 4: I2C driver static view

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. For current implementation channel parameters are NULL.

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 (i2c_*portObj)*	Description
1	*portNumber*	Preserve port or instance number of I2C
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 Object (IOM)

S.No	Structure Elements (i2c_*chanObj)*	Description
1	*dataPacket*	Pointer to store current IO packet
2	*inUse*	To check whether channel is in use or not.
3	*mode*	Channel mode of operation: Input or Output.
4	*port*	Pointer to device port *i2c_portObj* structure.
5	*cbFxn and cbArg*	IOM callback function and its argument
7	*Ddc handle*	To store the channel handle passed from DDC layer

2.1.2.3 The Driver Object (DDC)

S.No	Structure Elements (DDC_I2cObject)	Description
1	*i2cRegs*	CSL registers
2	*intNum*	Interrupt number
3	*mode*	Mode of operation.
4	*i2cOwnAddr*	Own address of I2C
5	*moduleInputClkFreq*	Input clock frequency
7	*i2cBusFreq*	I2C bus frequency
8	*numBits*	Bit count
9	*addressing*	7/10 bit addressing
10	*instanceId*	Instance id
11	*state*	State of driver
12	*numOpens*	Number of opened channels
13	*pendingState*	IO is in progress or not
14	*Cancel_Pending_IO*	To cancel current IO
15	*devBusySem*	Semaphore to block other tasks in accessing I2C
16	*completionSem*	Semaphore to block driver during interrupt mode
17	*currError*	Current transmission error
18	*currFlags*	Current flags
19	*currBuffer*	Current transaction buffer
20	*currBufferLen*	Current transaction buffer length
21	*hwEventCallback*	Call back when accessed as slave
22	*appSlvHandle*	Argument to above callback

2.1.2.4 The Channel Object (DDC)

S.No	Structure Elements (DDC_I2cDriverObject)	Description
1	*callBack*	Callback function
2	*appHandle*	Callback function argument
3	*pi2cInstHandle*	I2C driver object handle

2.1.2.5 The Device Params

The file **psp_i2c.h** has the **PSP_I2cConfig** data structure that is passed as I2CdevParams to initialization function of the driver. The params are explained below:

S.No	Structure Elements (i2c_*chanObj)*	Description
1	opMode	Operational mode of the driver
2	moduleInputClkFreq	Input Frequency to I2C Module.
3	i2cBusFreq	Output Data rate, in Kbps, of the I2C
4	i2cOwnAddr	Own address of the device
5	NumBits	BitCount
6	addressing	Addressing mode[7bit or 10 bit addressing]

2.2 Dynamic View

2.2.1 The Execution Threads

The device drivers typically implement Synchronous interface to the user. The I2C device driver operation involves following execution threads:

BIOS thread: Function to load and un-load I2C driver will be under BIOS OS initialization.

Application thread: Creation of channel, Control of channel, deletion of channel and processing of I2C data will be under application thread. All Synchronous IO occur in the application thread of control, the calling thread may suspend for the requested transaction to complete.

Interrupt context: Processing TX/RX data transfer and Error interrupts if the driver mode is interrupt.

2.2.2 Input / Output using I2C driver

In I2C, the application can perform IO operation using GIO_submit () calls (corresponding IOM function is i2c_mdSubmitChan ()) to receive transmission parameters like buffers and flags. The configuration for memory buffer address and size of number of bytes to transfer should be passed as an argument to the GIO_submit call.

The I2C channel transfer is enabled upon submission of the IO request. Once the IOP is submitted, the driver configures the appropriate registers (Mainly mode register) from the IOP. Once the requested numbers of buffers have been received or transmitted, the driver will notify the IOP completion to the application by returning IOM_COMPLETED value or any appropriate error code.

2.2.3 Synch-IO Mechanism

I2C provides synchronous mode of operation in between application and driver. Once application submits an IO driver returns to application only after it completes the requested operation. Sync IO mechanism for different modes are explained below.

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 I2C 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.

2.2.4 IOM (DDA Adaptation)

2.2.4.1 Driver Creation

The Flow diagram below depicts the creation phase of the BIOS I2C driver. The OS calls DEV_init which calls i2c_mdBindDev to create a driver instance.

User is expected to invoke i2c_mdBindDev way up in the application startup phase, perhaps in a central driver initialization function.

The i2c_mdBindDev performs book-keep functions on the driver. It attaches the DDC create functions for use later during actual initialization of each device instance.

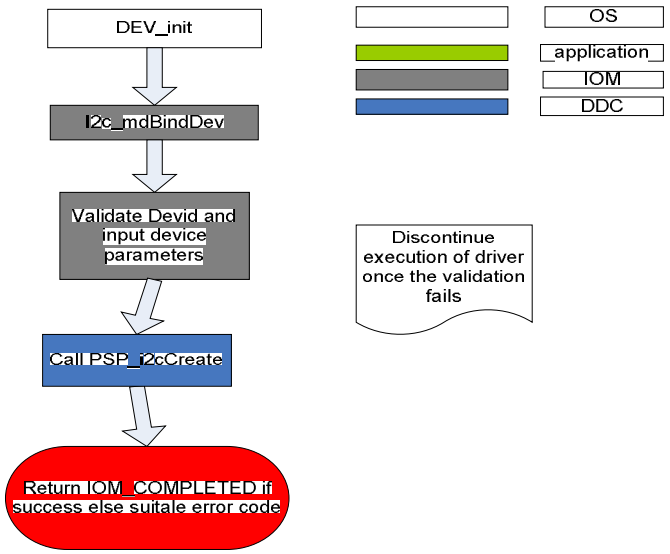

Figure 5: Driver Create detailed Flow Diagram -1

2.2.4.2 Driver Open

When the application calls the GIO_Create () which inturn calls i2c_mdCreateChan (), driver entry point. The callback is registered with DSP/BIOS, which is currently not used as I2C is a Sync driver. This creates the handle which will be returned to the application and application uses this for performing IO operations or IOCTL operations.

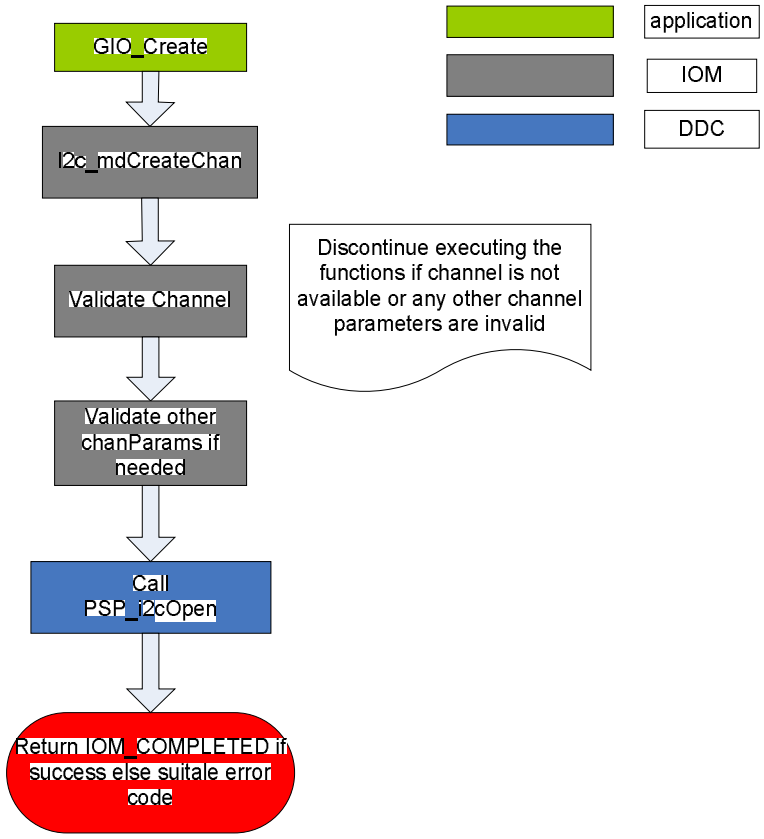

Figure 6 Driver Open Detailed Flow Diagram - 1

2.2.4.3 IO Control

The I2C Driver provides i2c_mdControlChan () to set/get common configuration parameters on the driver at run time.

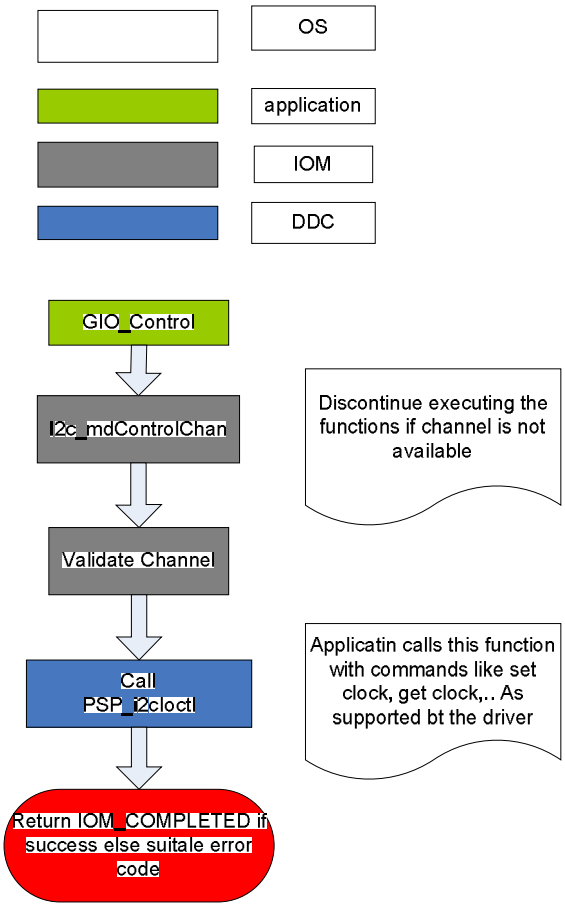

Figure 7: Driver IOCTL Detailed Flow Diagram -1

2.2.4.4 IO Access

The application will access I2C driver IOM API I2C_mdSubmitChan through interface functions from DSP/BIOS. These functions are registered on the DSP/BIOS during the driver initialization

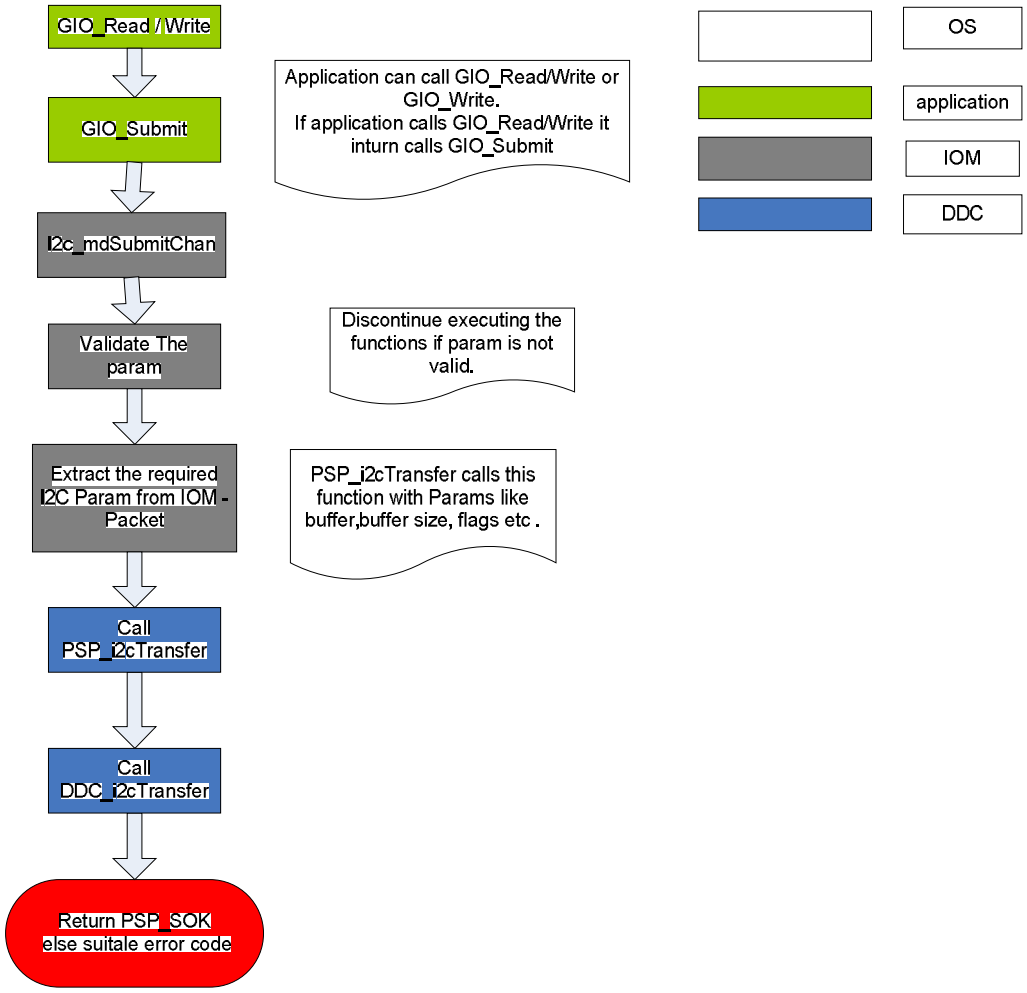

Figure 8: Driver transfer Detailed Flow Diagram - 1

2.2.4.5 Driver Close

The application invokes the GIO_delete () function to close the channel of the I2C device.

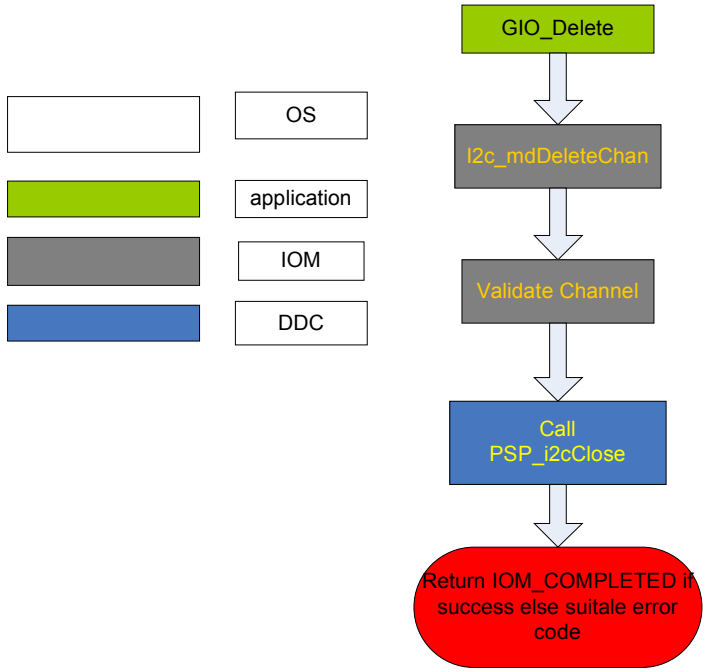

Figure 9: Driver Close Detailed Flow Diagram - 1

2.2.4.6 Driver Teardown

Following the call I2C_mdUnBindDev () one is required to restart from beginning over an mdBindDev () call to bring driver back to life. The driver de-initialize and delete functions de-initialize the I2C DDC and delete if any OS resources originally allocated through mdBindDev ().

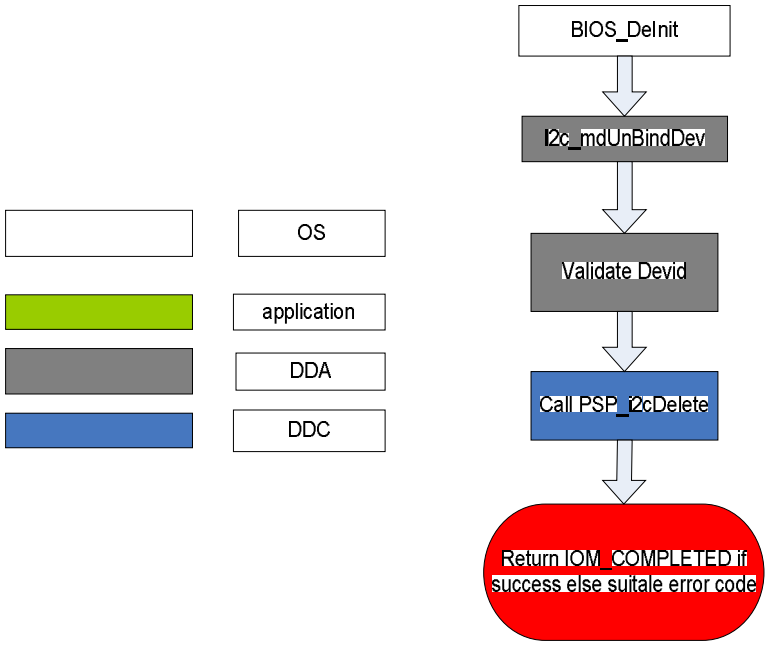

Figure 10: Driver Delete Detailed Flow Diagram – 1

2.2.5 DDC (Driver Core)

2.2.5.1 Driver Creation

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.

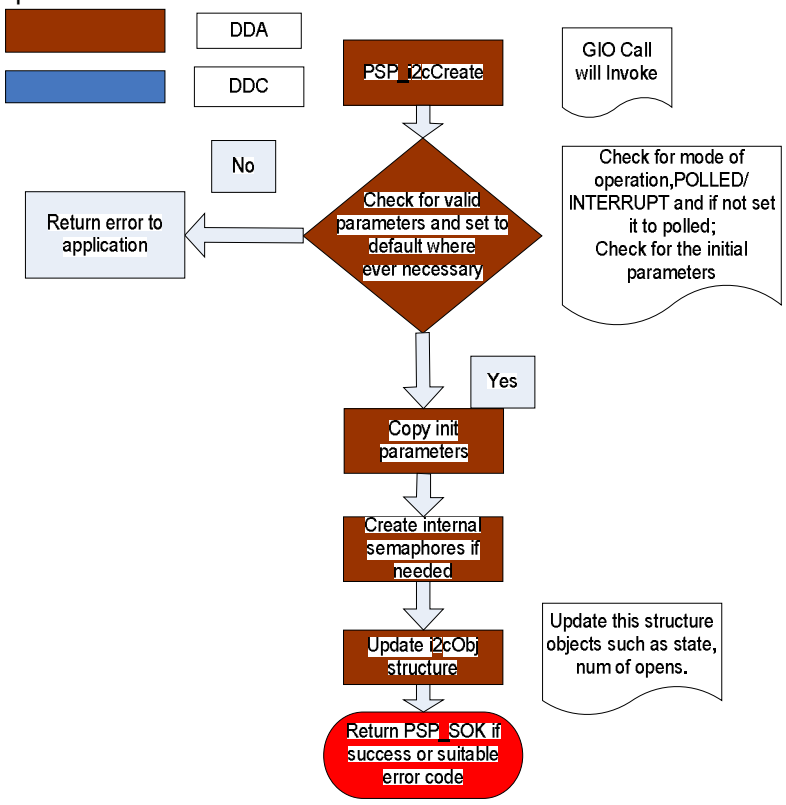

Figure 11 Driver Create Detailed Flow Diagram – 2

2.2.5.2 Driver Open

PSP_i2cOpen function is invoked to provide a handle for the further operations on the i2c. The callback is registered for the application call back, which is currently not used as I2C is a Sync driver. This creates the handle which will be returned to the application in case of no DDA adaptation layers or to the DDA adaptation layer. The driver is ready to accept Read/Write jobs or IOCTL operations

Design Constraints :

a. While porting to any board, clock calculations for the i2c bus have to be taken care properly.

b. Also while registering and un registering interrupts, care has to be taken care to perform the operation of that driver and of that instance only

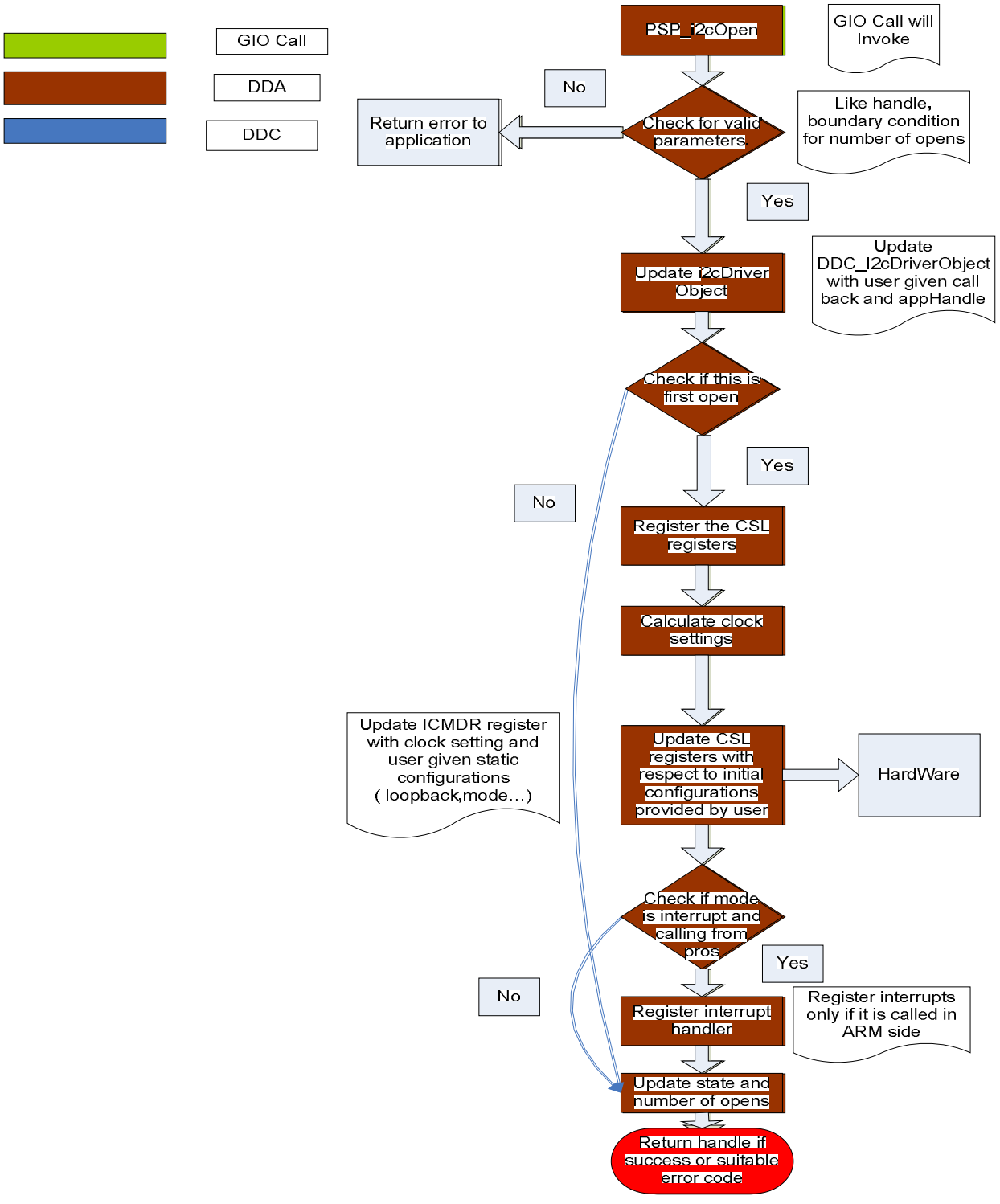

Figure 12: Driver Open Detailed Flow Diagram – 2

2.2.5.3 IO Control

DDC IOCTL function PSP_i2cloctl has IOCTL commands that are device specific or that require action on the part of the device driver call the driver's IOCTL and this function is used to set/get common configuration parameters on the driver at run time.

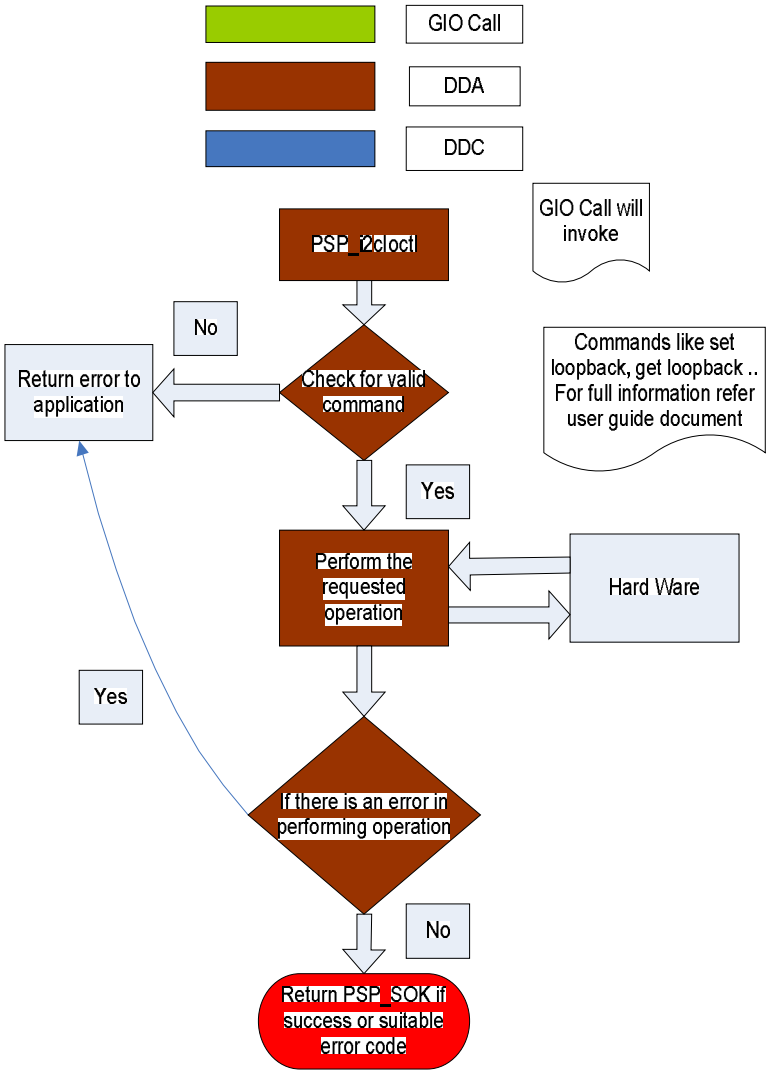

Figure 13 Driver IOCTL Detailed Flow Diagram - 2

2.2.5.4 IO Access

Application or the Adaptation layer calls PSP_i2cTransfer function to perform the read/write operation on a particular slave in the mode with which the driver is created.

Design Constraints :

c. If there comes any error while transmission, for ex: - NACK error, driver has to cancel the current IO as hardware does not perform the same.

d. While updating mode register with the flag parameters which are passed from application ensure the validity of flags and transfer parameters.

For Ex:- when repeat mode is set along with stop bit then buffer length should be zero other wise driver hangs up.

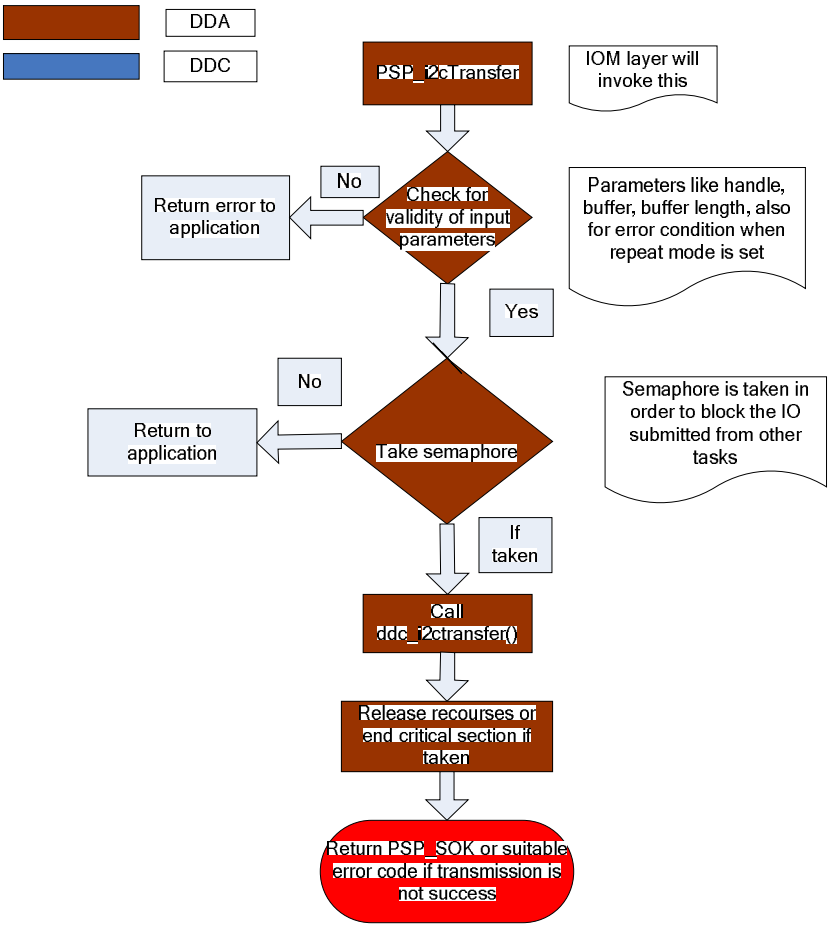

Figure 14: Driver transfer Detailed Flow Diagram - 2

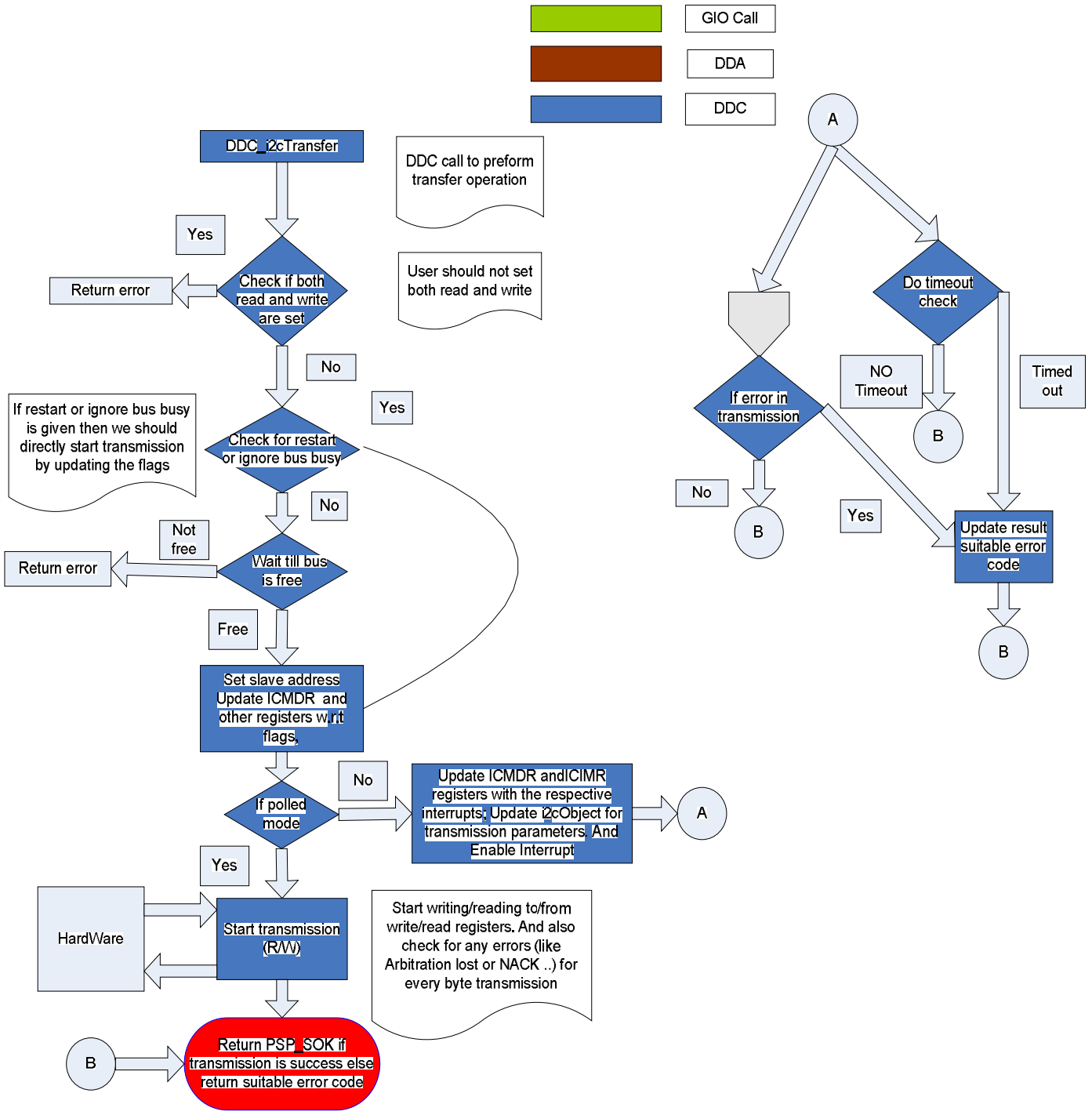

Figure 15 Driver transfer Detailed Flow Diagram - 3

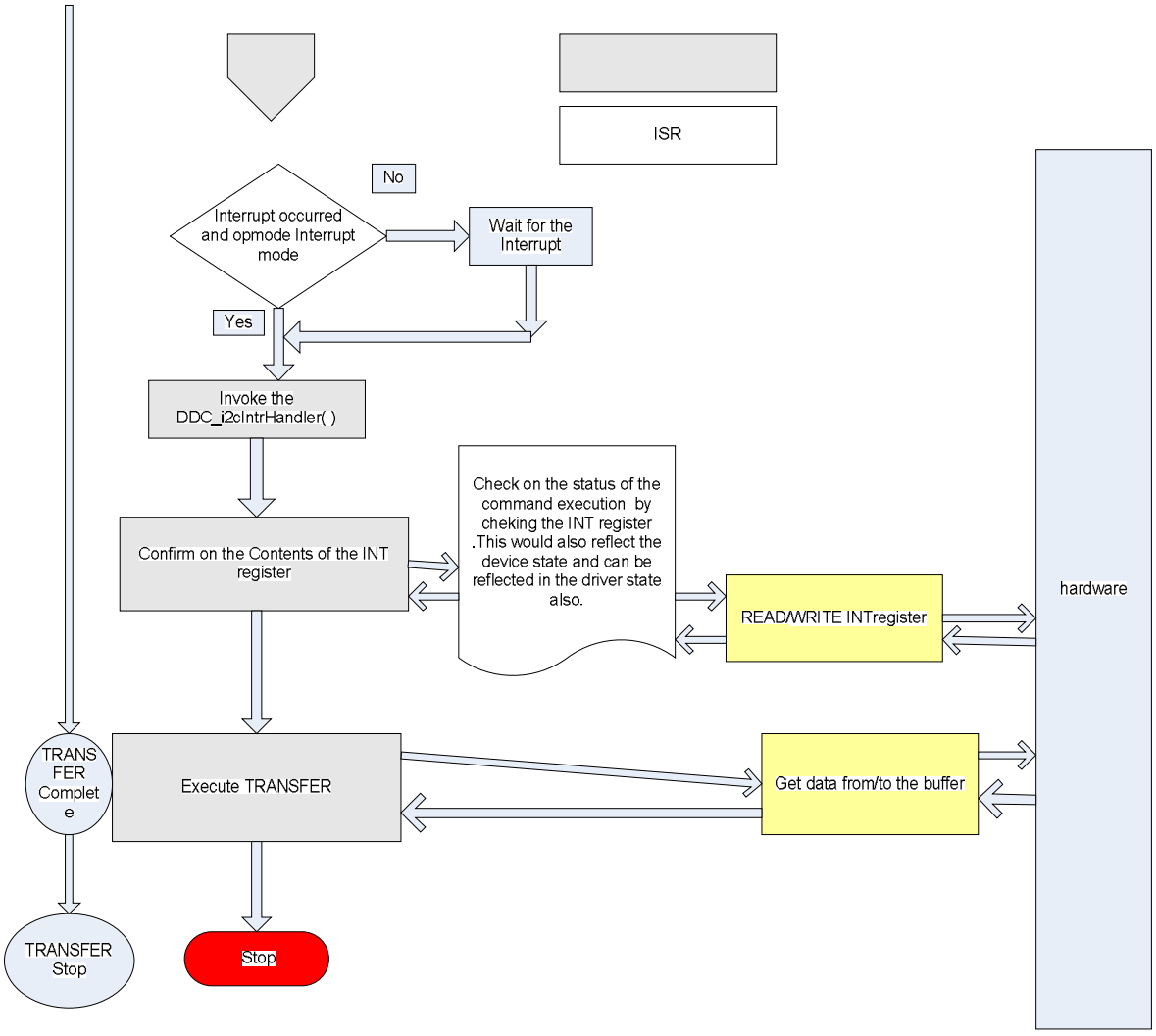

Figure 16: Driver transfer in ISR Detailed Flow Diagram

2.2.5.5 Driver Close

Application or Adaptation layer invokes the PSP_i2cClose () function to close the opened instance of the I2C device.

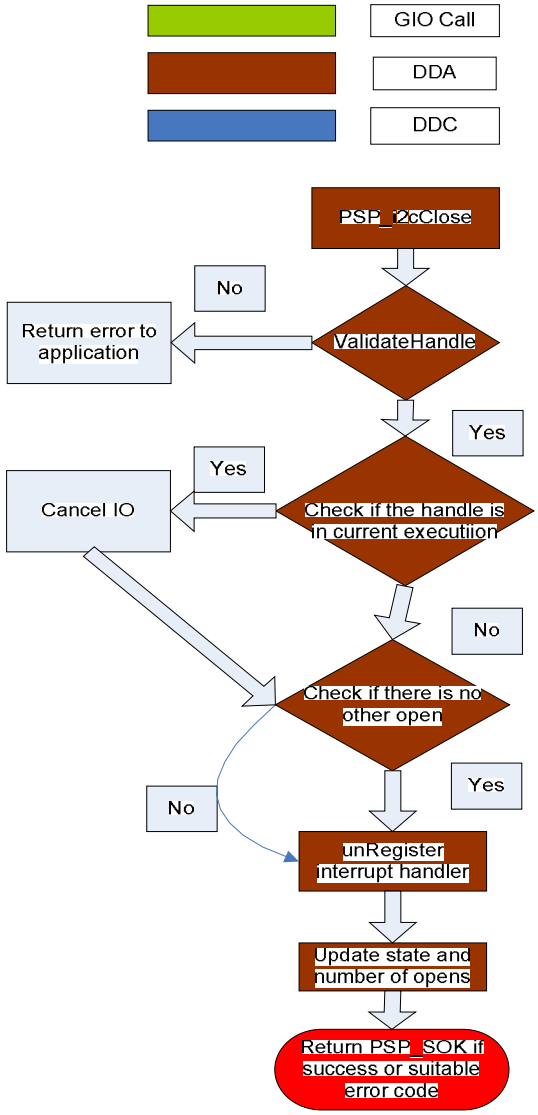

Figure 17 : Driver Close Detailed Flow Diagram – 2

2.2.5.6 Driver Teardown

Following the call of PSP_i2cDelete () function, it de-initialize the I2C DDC and delete if any OS resources originally allocated through PSP_i2cCreate ().

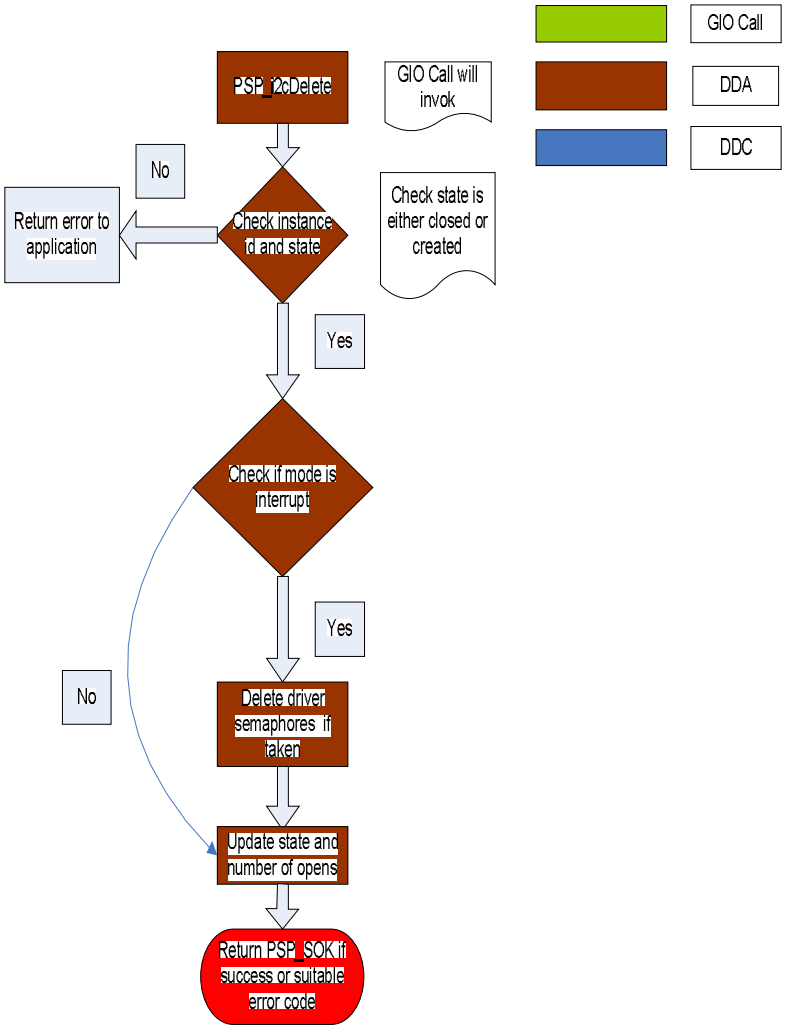

Figure 18: Driver Delete Detailed Flow Diagram – 2

3 APPENDIX A – IOCTL commands

Applications that use I2C device driver is expected to use the data types and IOCTL commands specified in psp_i2c.h file to perform runtime configurations.

S.No	ERROR Code	Description
1	PSP_I2C_IOCTL_SET_BIT_RATE	Set the I2C clock
2	PSP_I2C_IOCTL_GET_BIT_RATE	Get the I2C clock
3	PSP_I2C_IOCTL_CANCEL_PENDING_IO	To cancel pending IO
4	PSP_I2C_IOCTL_BIT_COUNT	To set bit Count value
5	PSP_I2C_IOCTL_NACK	To enable or disable NACK dynamically
6	PSP_I2C_IOCTL_SET_OWN_ADDR	To change I2C own address dynamically
7	PSP_I2C_IOCTL_GET_OWN_ADDR	To Get I2C own address.

4 APPENDIX B – Error Codes

Driver returns following errors if it encounters any error conditions.

S.No	ERROR Code	Description
1	PSP_I2C_BUS_BUSY_ERR	When driver finds bus to be busy
2	PSP_I2C_ARBITRATION_LOSS_ERR	When driver lost its arbitration
3	PSP_I2C_NACK_ERR	Error when slave doesn't acknowledges
4	PSP_I2C_TRANSMIT_UNDERFLOW_ERR	Transmit under-run indication
5	PSP_I2C_RECEIVE_OVERFLOW_ERR	Receive overrun indication
6	PSP_I2C_CANCEL_IO_ERROR	When cancel IO command is issued

5 APPENDIX C – Flags

Application while calling GIO_READ/GIO_WRITE/GIO_SUBMIT has to send appropriate flags as a part of its parameters to configure the hardware.

S.No	Flag	Description
1	PSP_I2C_READ	To issue read command
2	PSP_I2C_WRITE	To issue write command
3	PSP_I2C_MASTER	To enable i2c driver as master
4	PSP_I2C_START	To Generate start bit before transmission
5	PSP_I2C_STOP	To generate STOP bit after transmission
6	PSP_I2C_RESTART	To user restart condition
7	PSP_I2C_REPEAT	To enable repeat mode
8	PSP_I2C_IGNORE_BUS_BUSY	To ignore when bus is busy