DSP/BIOS UART Device Driver

Architecture/Design Document

Revision History

Document Version	Author(s)	Date	Comments
0.1	Kapil Bohra	June 19, 2006	Created the document
0.2	Rinkal Shah	December 22, 2006	Review comments closed and BIOS version changed to 5.31
0.3	Nagarajuna K Kapil Bohra	December 15, 2006	Modified for the DM648
0.4	Jayaprakash N	December 22, 2006	Modified for new UART architecture for DM648
0.5	Kapil Bohra	January 08, 2007	Modification as per the changed architecture
0.6	Nagarjuna K	January 29, 2007	Renamed DM64g to DM648
0.7	Nagarjuna K	June 15, 2007	Corrected version number for tools used
0.8	Nagarjuna K	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 UART 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
DDC	TI terminology for portion of device driver that is abstracted of any given OS
IOM	TI terminology for portion of device driver that is specific to target OS. This constitutes "adaptation" of the generic DDC to identified target OS.
IP	Intellectual Property
ISR	Interrupt Service Routine
OS	Operating System

1.2 Related Documents

1.	SPRU616	DSP/BIOS Driver Developer's Guide
3.	Uart format specifications version 1.3	UART Specs

1.3 Hardware

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

The UART module core used here has the following blocks:

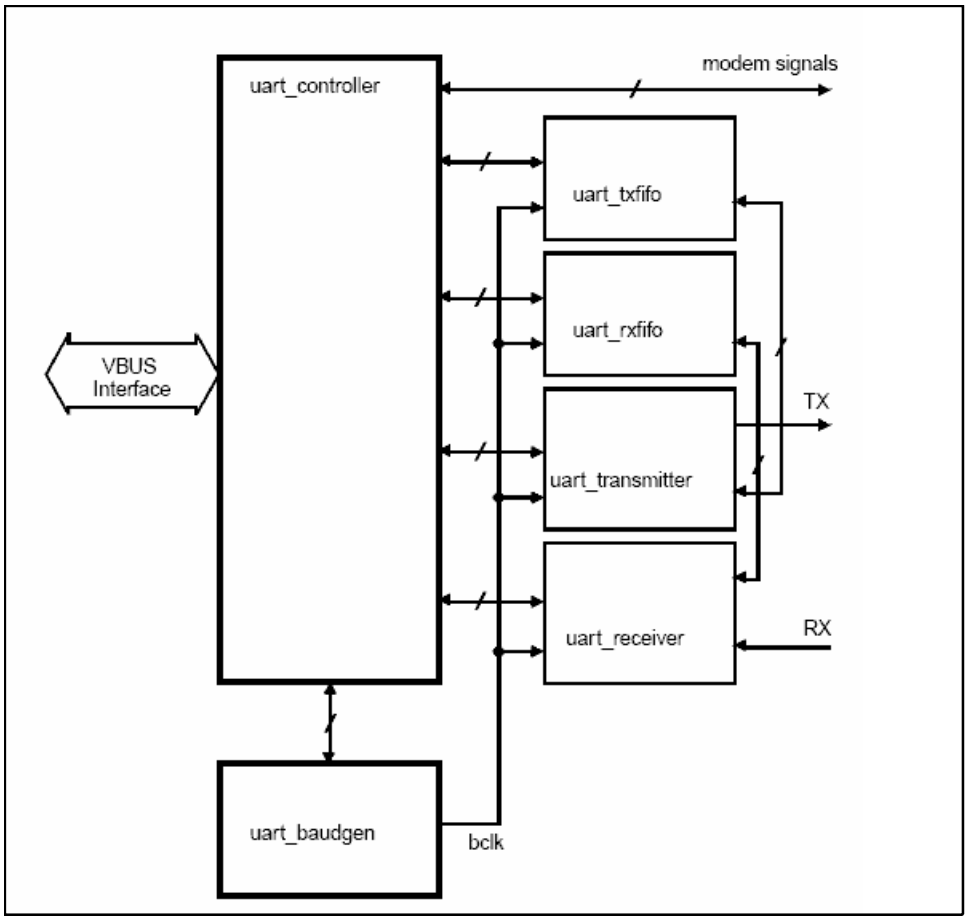

Figure 1 UART Block Diagram

1.4 Software

The UART mini-driver discussed here is running DSP/BIOS on the 64x+ DSP. However the UART 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 block diagram below shows the overall system architecture.

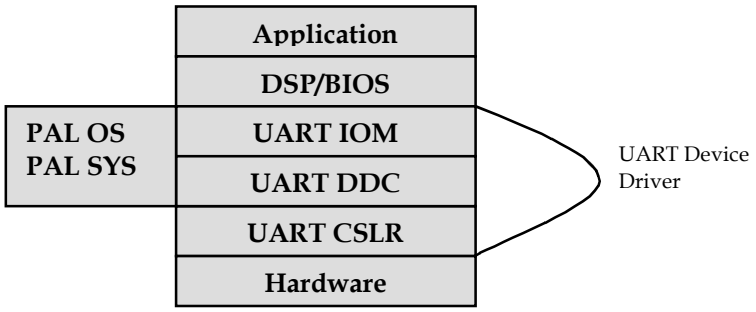

Figure 2 System Architecture

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 and performs all the OS related operations required for the driver core (DDC).

DDC is the driver core which actually performs the device specific operations.

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.

1.5 Component Interfaces

In the following subsections, the interfaces implemented by each of the sub-component are specified. Refer to UART 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 *uart_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 UART Driver implementation. The uart_mdUnbindDev () constitutes the “Un-loading” of the UART driver. The IOM mini-driver implements the following API interfaces to the class driver.

S.No	IOM Interfaces	Description
1	uart_mdBindDev ()	Allocates and configures the UART port specified by devid.
2	uart_mdUnbindDev ()	Removes the UART device from use
3	uart_mdCreateChan ()	Creates a communication channel in specified mode to communicate data between the application and the UART device instance.
4	uart_mdDeleteChan ()	Frees a channel and all its associated resources.
5	uart_mdControlChan ()	Implements the IOCTLS for UART IOM mini divers.
6	uart_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 following basic interfaces are implemented and exposed to the IOM layer by the DDC layer of UART driver.

S.No	DDC Interfaces	Description
1	PSP_uartCreate()	Initialize/Setup the UART hardware with the given configuration parameters.
2	PSP_uartDelete()	Does the reverse of *PSP_uartCreate.*
3	PSP_uartOpen()	Configure UART's TX/RX parameters to establish channel.
4	PSP_uartIoctl ()	Perform input/output control on UART Hardware.
5	PSP_uartClose ()	Does the reverse of *PSP_uartOpen.*
6	PSP_uartTransfer()	Submits IOP requests to perform input/output

1.5.3 CSLR Interface

The CSL register interface (CSLR) provides register level implementations. CSLR is used by the DDC to configure UART 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 UART 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 UART driver provides one read/write channel in order to perform IO operations.

The *Port Object* maintains the state of the UART 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. The following figure shows the generic port-channel-hardware mapping for UART driver.

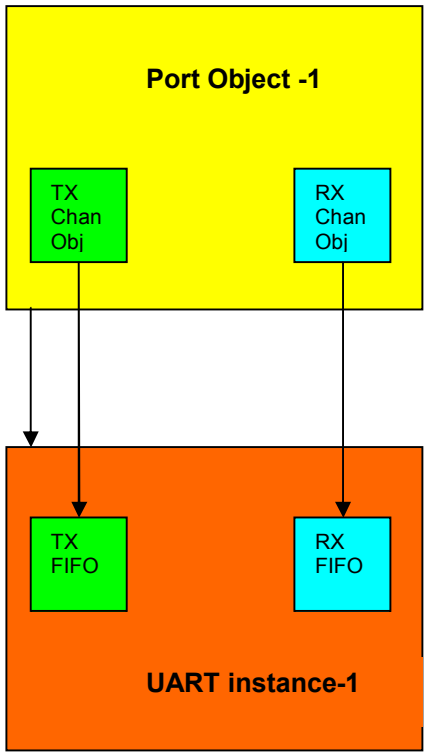

Figure 3 Port and Channel Object

1.6.2 Design Constrains

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

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

2 UART Driver Software Architecture

This section details the data structures used in the UART 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.

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

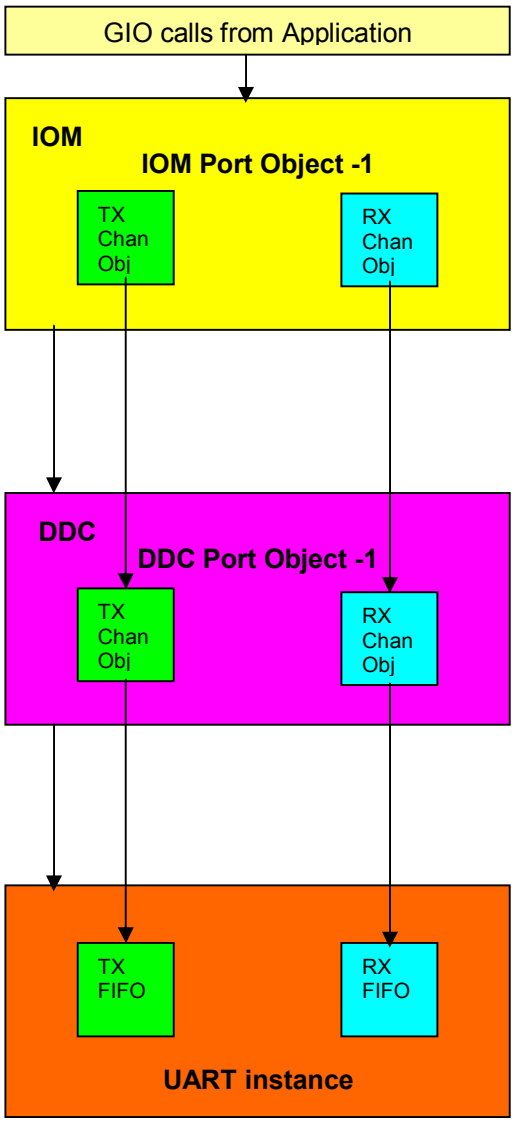

Figure 4 UART driver static view

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 (uart_*portObj)*	Description
1	*instNum*	Preserve port or instance number of UART
2	*inUse*	*Flag to the object source is used or free*
3	*Chan[]*	Holds all channel objects for this port

2.1.2.2 The Channel Object(IOM)

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

2.1.2.3 The Device Params

The application passes a data structure UART_devParams that is used to initialization function of the driver. The params are explained below:

S.No	Structure Elements	Description
1	opMode	Operational mode of the driver
2	inputFreq	Input Frequency to UART Module.
3	hEdma	EDMA channel handle
4	fifoEnable	FIFO Mode is enabled or disabled
5	loopbackEnabled	Loop Back is enabled or disabled
6	xferConfigParams	Hardware configuration parameters

2.1.2.4 The DDC Device Params

The application passes a data structure UART_devParams that is used to initialization function of the driver. The params are explained below:

S.No	Structure Elements	Description
1	appHandle	Handle for the IOM Device structure
2	devState	State of the DDC
3	ddcIntrNum	Interrupt Number for the instance of the UART
4	hwInfo	Handle for the hardware registers for the Uart
5	txIO	Transmitter Channel Handle
6	rxIO	Receiver Channel Handle
7	instanceId	Instance of the Uart being worked upon
8	opMode	Mode of operation for the Uart
9	configParams	Configuration Parameters for the Uart

2.1.2.5 The DDC Channel Params

The params are explained below:

S.No	Structure Elements	Description
1	stats	Stats for the channel
2	queuePendingList	Pending IOPs for the channel
3	queueFreeList	List of the free IOPs
4	activeIOP	Packet Under progress in the Channel
5	activeBuf	Buffer address of the active packet
6	bytesRemaining	Request of bytes remaining for the active packet
7	fifoEnabled	If FIFO is enabled
8	chunkSize	Data Transfer needed for the one set of buffer
9	hDev	Handle for the Uart DDC Device
10	cbFxn	Callback for the application
11	cbArg	Arguments to be passed to the Application
12	mode	Mode of operation
13	iopPool	Static pool of IO Packets
14	inUse	If the channel is initialized
15	flush	used to find out the Flush Command
16	flushSyncSem	Semaphore taken for the flushing

17	hEdma	Edma Handle
18	Tcc	Transfer complete code number
19	ChId	DMA Channel

2.1.2.6 The DDC IO Packet Structure

The params are explained below:

S.No	Structure Elements	Description
1	iopLink	Link for the next IO Packet in the Queue
2	cmd	IO commnad read/write
3	buf	Application supplied IO buffer address
4	xferRequest	Number of bytes being requested in this transaction
5	timeout	Amount of time given to this iop to complete
6	appContext	Application supplied application context
7	status	Status of IO transactions
8	xferActual	Number of bytes actually transferred
9	appCbk	Call back function for this request
10	appData	Call back function argument if any

2.1.2.7 The PSP IO Packet Structure

The params are explained below:

S.No	Structure Elements	Description
1	cmd	IO request command. Read/Write
2	addr	Pointer to the application Buffer
3	size	Size of the application Buffer
4	actualSize	Acutal bytes transmitted by the driver
5	cbFxn	UART Application call back per request
6	appPtr	Pointer to application specified data
7	status	Status of the transfer. This data is filled by the driver
8	timeout	timeout in millisecs

2.2 Dynamic View

2.2.1 The Execution Threads

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

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

Application thread: Creation of channel, Control of channel, deletion of channel and processing of UART 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 UART driver

In UART, the application can perform IO operation using GIO_submit () calls (corresponding IOM function is uart_mdSubmitChan ()) to receive transmission parameters like buffers. 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 UART channel transfer is enabled upon submission of the IO request. Once the IOP is submitted, the driver configures the appropriate registers 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 Functional Decomposition

2.2.3.1 uart_mdBindDev

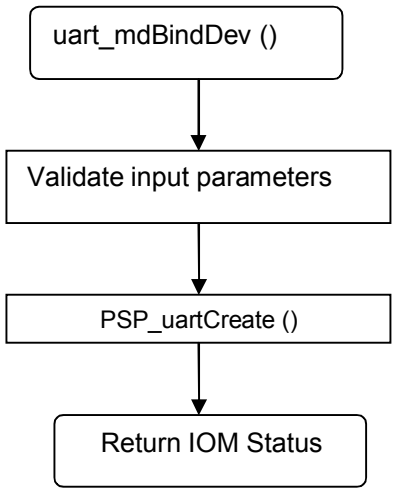

Figure 5 uart_mdBindDev () flow diagram

What is required from the application?

- Valid structure to setup the UART
- Valid device parameters structure

During *uart_mdBindDev*, the mini-driver has access only to pointers of device parameters. Memory for these structures is to be allocated outside the driver by the application.

2.2.3.2 uart_mdUnBindDev

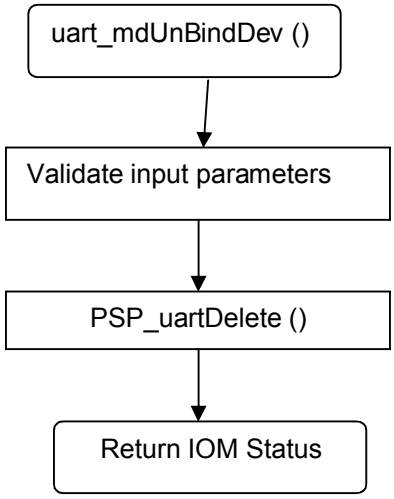

Figure 6 uart_mdUnBindDev () flow diagram

The *uart_mdBindDev* and *uart_mdUnbindDev* functions are called by the DSP-BIOS if driver is created using TCF configuration. Otherwise these functions are called by device driver BIOS APIs like *DEV_createDevice ()* etc. Refer BIOS GIO/IOM model device driver guide for more details. These functions will not be used by the application directly to interface with the UART driver.

2.2.3.3 uart_mdCreateChan

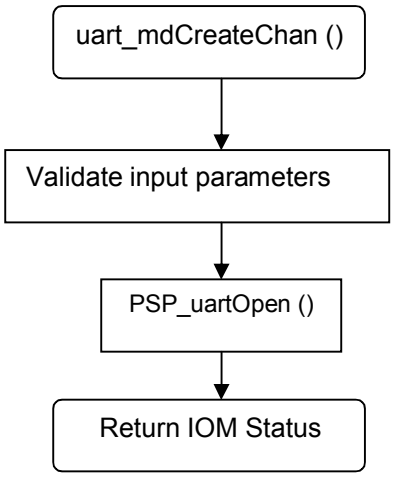

Figure 7 uart_mdCreateChan () flow diagram

2.2.3.4 uart_mdDeleteChan

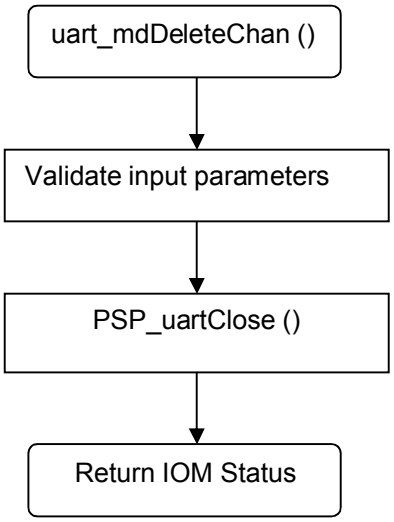

Figure 8 uart_mdDeleteChan () flow diagram

2.2.3.5 uart_mdControlChan

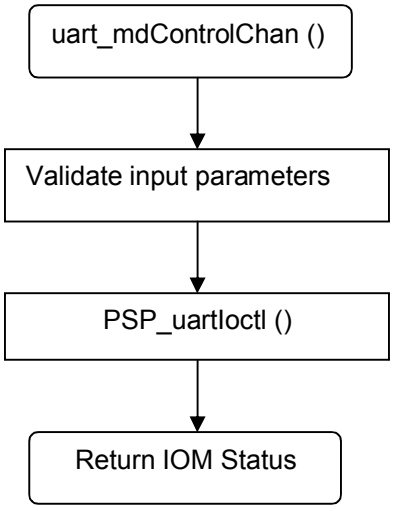

Figure 9 uart_mdControlChan () flow diagram

2.2.3.6 uart_mdSubmitChan

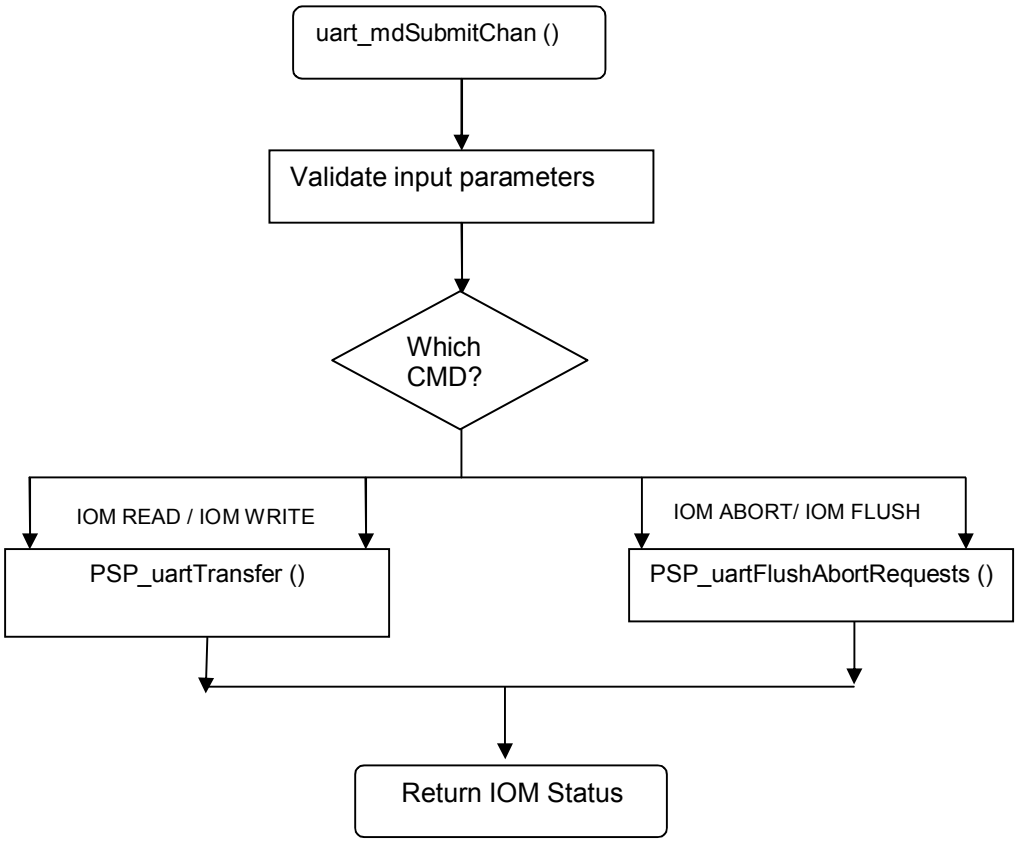

Figure 10 uart_mdSubmitChan () flow diagram

2.2.3.7 psp_uartCreate

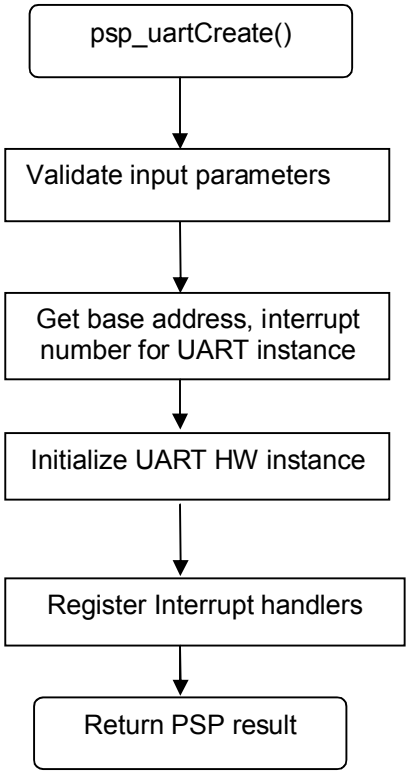

Figure 11 psp_uartCreate

2.2.3.8 psp_uartDelete

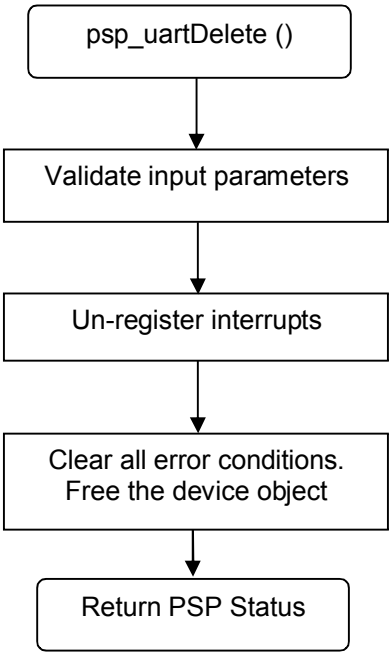

Figure 12 psp_uartDelete

2.2.3.9 psp_uartOpen

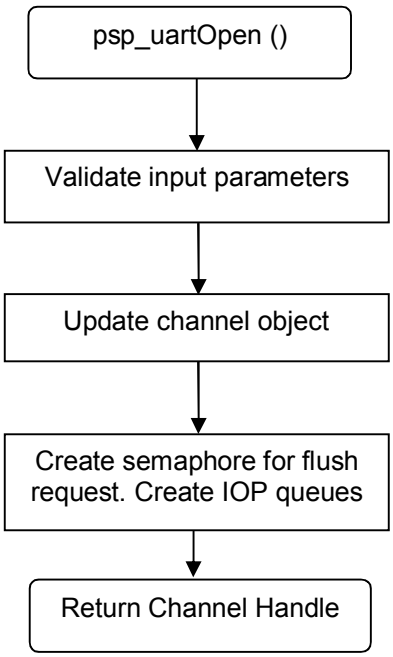

Figure 13 psp_uartOpen

2.2.3.10 psp_uartClose

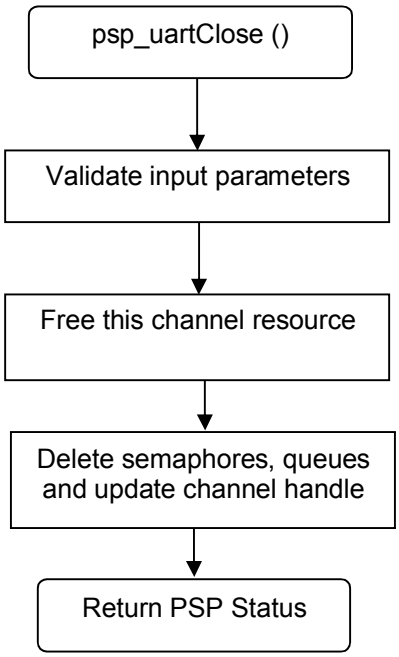

Figure 14 psp_uartClose

2.2.3.11 psp_uartIoctl

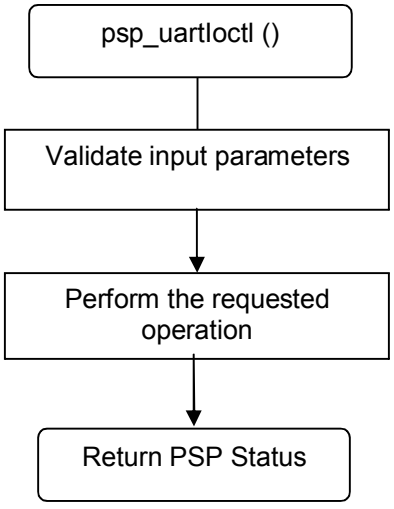

Figure 15 psp_uartIoctl

2.2.3.12 psp_uartRead or psp_uartWrite

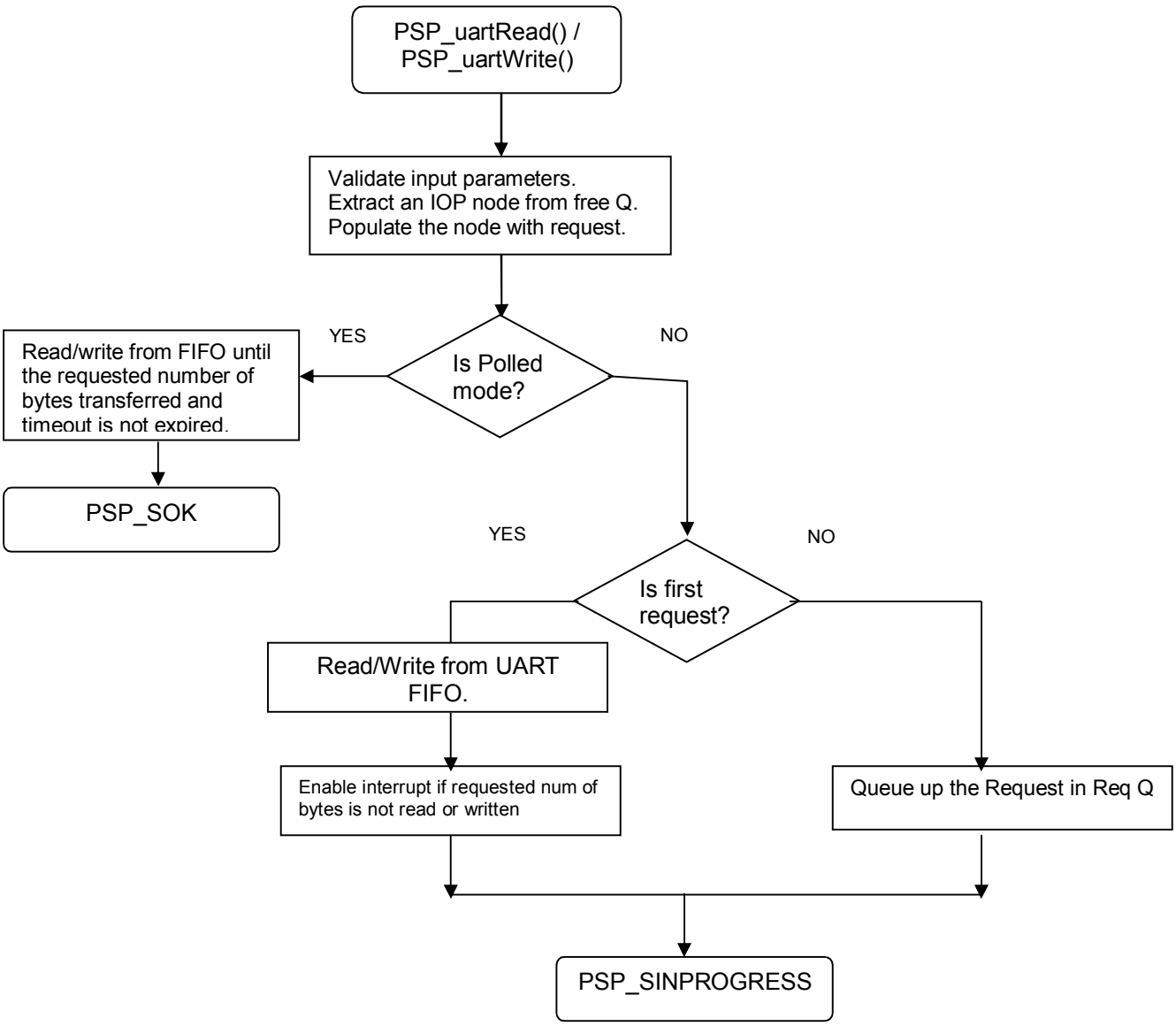

Figure 16 psp_uartRead / psp_uartWrite

2.2.3.13 PSP_uartAbort

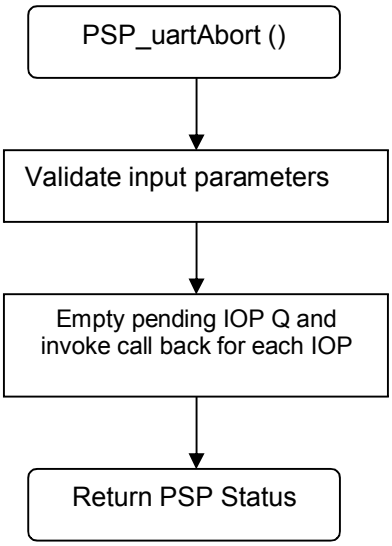

Figure 17 psp_uartAbort

2.2.4 Synch-IO Mechanism

UART 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.

3 APPENDIX A – IOCTL commands

The application can perform the following IOCTL on the channel. All commands shall be sent through TX or RX channel except for specifics to TX /RX.

S.No	IOCTL Command	Description
1	PSP_UART_IOCTL_SET_BAUD	Set the Baud rate.
2	PSP_UART_IOCTL_SET_STOPBITS	Set number of stop bits.
3	PSP_UART_IOCTL_SET_DATABITS	Set number of data bits.
4	PSP_UART_IOCTL_SET_PARITY	Set parity Odd/Even
5	PSP_UART_IOCTL_SET_FLOWCONTROL	Set flow control HW or SW
6	PSP_UART_IOCTL_SET_TRIGGER_LEVEL	Set trigger level for FIFO
7	PSP_UART_IOCTL_RESET_RX_FIFO	Clear RXFIFO
8	PSP_UART_IOCTL_RESET_TX_FIFO	Clear TXFIFO
9	PSP_UART_IOCTL_CANCEL_IO	Cancel the current IO
10	PSP_UART_IOCTL_GET_STATS	Get statistics information
11	PSP_UART_IOCTL_CLEAR_STATS	Clear statistics information
12	PSP_UART_IOCTL_MAX_IOCTL	Book keep – Max ioctl