



---

**SOFTWARE ARCHITECTURE TEMPLATE**

---

**UPP****Driver Design Document**

Rev No	Author(s)	Revision History	Date	Approval(s)
0.1	Imtiaz SMA	Created the document	11 Jan 2010	

Information in this document is subject to change without notice. Texas Instruments may have pending patent applications, trademarks, copyrights, or other intellectual property rights covering matter in this document. The furnishing of this document is given for usage with Texas Instruments products only and does not give you any license to the intellectual property that might be contained within this document. Texas Instruments makes no implied or expressed warranties in this document and is not responsible for the products based from this document

---

## TABLE OF CONTENTS

---

<b>1</b>	<b>Introduction.....</b>	<b>5</b>
1.1	Purpose & Scope .....	5
1.2	Terms & Abbreviations.....	5
1.3	References .....	5
1.4	Overview.....	6
1.4.1	<i>Hardware Overview .....</i>	<i>6</i>
1.4.2	<i>Software Overview .....</i>	<i>7</i>
<b>2</b>	<b>Requirements.....</b>	<b>8</b>
2.1	Assumptions.....	10
2.2	Constraints.....	10
<b>3</b>	<b>Design Description .....</b>	<b>11</b>
3.1	Design Philosophy .....	11
3.1.1	<i>The Instance Concept .....</i>	<i>11</i>
3.1.2	<i>The Channel Concept.....</i>	<i>11</i>
3.2	Static View .....	12
3.3	Functional Partition.....	12
3.3.1	<i>IOM Interface .....</i>	<i>12</i>
3.4	Dynamic view .....	14
3.4.1	<i>Driver Creation (Driver Initialization and Binding).....</i>	<i>15</i>
3.4.2	<i>Channel Creation.....</i>	<i>19</i>
3.4.3	<i>IO submit.....</i>	<i>21</i>
3.4.4	<i>Control Commands.....</i>	<i>24</i>
3.4.5	<i>Channel deletion .....</i>	<i>26</i>
3.4.6	<i>Driver unbinding/deletion.....</i>	<i>27</i>
3.4.7	<i>Asynchronous IO Mechanism .....</i>	<i>28</i>
3.5	Constants & Enumerations .....	29
3.5.1	<i>Upp_POLLED_RETRYCOUNT.....</i>	<i>29</i>
3.5.2	<i>Upp_MAXLINKCNT.....</i>	<i>29</i>
3.5.3	<i>Upp_DevMode .....</i>	<i>30</i>
3.5.4	<i>Data Structures .....</i>	<i>31</i>
3.5.5	<i>Driver Instance Object.....</i>	<i>31</i>
3.5.6	<i>Channel Object .....</i>	<i>32</i>
3.5.7	<i>Upp_Params.....</i>	<i>34</i>
3.5.8	<i>Upp_ChanParams.....</i>	<i>35</i>
3.5.9	<i>Upp_signalConfig .....</i>	<i>36</i>
3.6	API Definition .....	37
3.6.1	<i>Upp_init.....</i>	<i>37</i>

---

<b>4</b>	<b>Decision Analysis &amp; Resolution .....</b>	<b>39</b>
4.1	DAR Criteria .....	39
4.1.1	<i>Alternative 1</i> .....	39
4.1.2	<i>Alternative 2</i> .....	39
4.2	Decision.....	39
<b>5</b>	<b>Revision History .....</b>	<b>39</b>

---

## TABLE OF FIGURES

---

Figure 1 UPP Hardware Overview .....	6
Figure 2 UPP Software Overview .....	7
Figure 3 Dynamic view of the UPP driver .....	14
Figure 4 Driver Initialization .....	16
Figure 5 Driver Instance Binding .....	18
Figure 6 UPP channel create command Flow .....	20
Figure 7 UPP driver submit control flow.....	23
Figure 8 UPP Control command Flow.....	25
Figure 9 UPP Driver deletion .....	27
Figure 10 UPP driver Asynchronous IO .....	28

# 1 Introduction


This document describes the UPP DSP/BIOS device driver. The UPP driver conforms to the IOM driver model specified by the DSP/BIOS operating system. This document explains the design of the UPP driver. Also the data types, data structures and application programming interfaces provided by the UPP driver are explained in detail.


## 1.1 Purpose & Scope

This document explains the UPP driver design in the context of the DSP/BIOS operating system. It explains the various programming interfaces provided by the driver. Please note that the UPP driver design discussed here is applicable to the C6748/OMAPL138 SoCs. This document does not explain how to use the UPP device driver, for usage instructions please refer to the BIOSPSP user guide that is available along with the driver.

## 1.2 Terms & Abbreviations

Term	Description
API	Application Programming Interface.
CSL	Chip Support Layer.
DDK	Device Driver Development Kit.
EDMA	Enhanced Direct Memory Access Controller.
IOM	IO Mini Driver Model.
INTC	Interrupt Controller
IP	Intellectual Property
ISR	Interrupt Service Routine
UPP	Universal Parallel Port

 *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.*

 *All references to OMAPL138 are also applicable to C6748. They are used interchangeably throughout this document.*

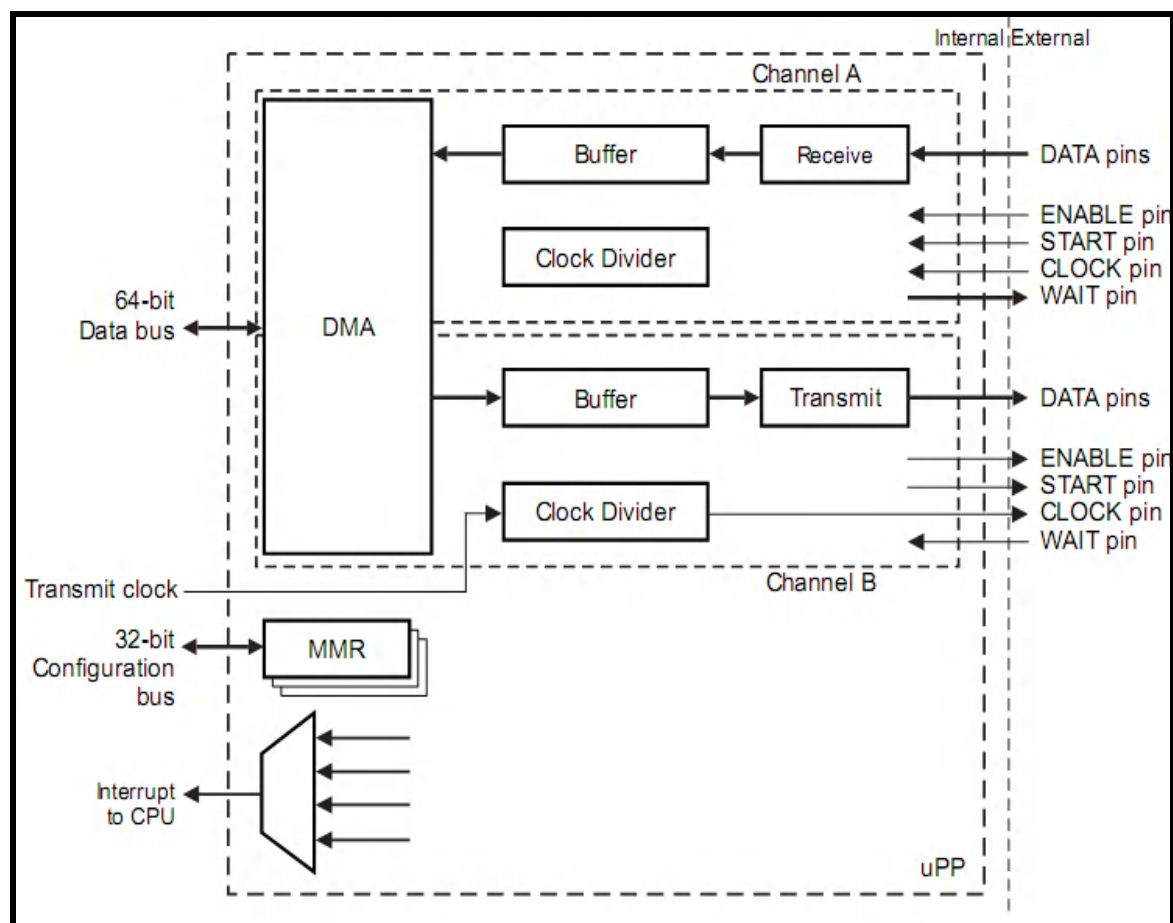
## 1.3 References

- EDMA3 User guide – SPRFUFL1.pdf
- Upp user guide
- DSP/BIOS™ API Reference Guide – spru403q.pdf.

## 1.4 Overview

The DSP/BIOS UPP device driver presented in this document is situated in the context of DSP/BIOS Operating System running on the OMAPL138. The following sub sections explain in detail the hardware and the software context of the UPP driver.

### 1.4.1 Hardware Overview



**Figure 1 UPP Hardware Overview**

The Figure 1 UPP Hardware Overview above shows the hardware overview of the UPP controller. The UPP contains 2 hardware channels (A& B), an internal DMA controller interface and configuration registers for configuring the UPP.

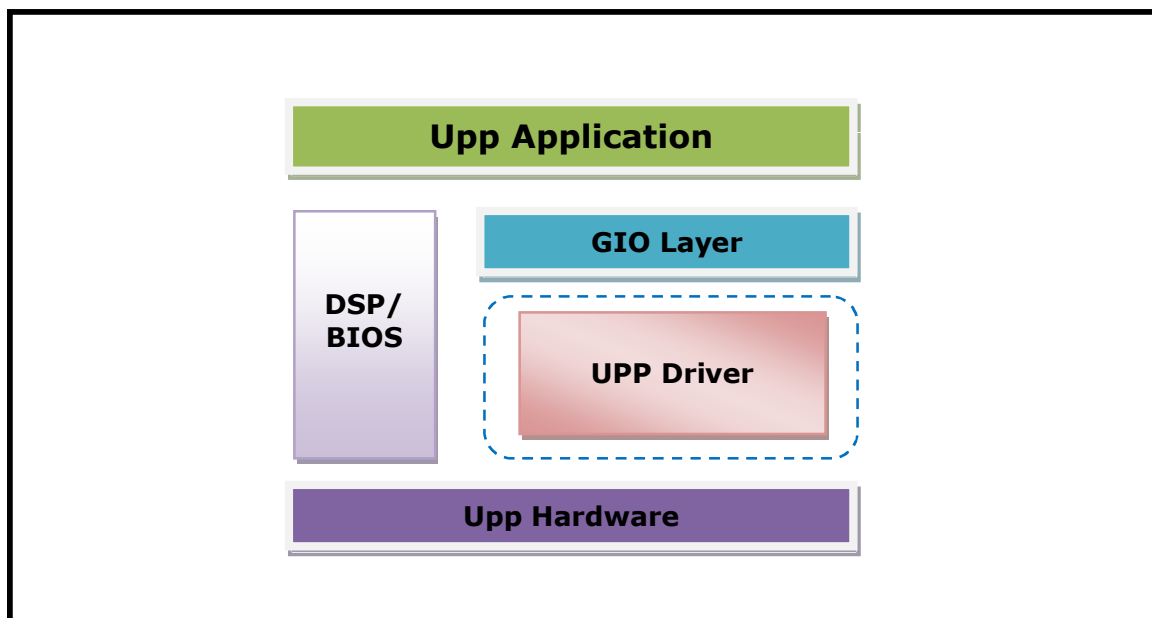
The UPP controller provides the hardware registers that allows the UPP to be configured for the parallel data transfer. The UPP Buffer FIFO provides additional data buffering required for the UPP to operate at high data rates.

The UPP contains a dedicated internal DMA (different from the EDMA interface available on the SoC) which allows parallel data movement from/ to the UPP and an external device. There are dedicated DMA channels I and Q available for the UPP to service the physical channels A and B.

**Note:** Some special modes (e.g. interleaved mode) will use both the DMA channels for servicing the channel A only, in such a scenario only one physical channel can be used.

### 1.4.2 Software Overview

This section describes in detail the UPP device driver architecture. The UPP driver described here conforms to the DSP/BIOS IOM driver model.



**Figure 2 UPP Software Overview**

Figure 2 UPP Software Overview depicts the various components involved in the transfer of parallel data when the UPP driver runs on the DSP core of OMAPL138 processor.

The UPP hardware allows parallel data to be transferred to connected devices like DAC or another UPP device. The UPP driver along with the DSP/BIOS operating system allows the applications to transfer and receive requests. The applications typically use GIO (or SIO) layer provided by the DSP/BIOS to submit requests to the driver.

The UPP driver allows a maximum of two channels to be created. Each channel corresponds to an actual physical channel of the UPP. Each of the physical channels can be configured as either a transmit channel or receive channel. When an application submits transmit request to the UPP driver, the driver programs the buffer address and the other parameters to the DMA and enables the required interrupts to transfer the data. Similarly when the application submits a data receive request, the driver programs the DMA parameters accordingly.

Note that the driver also performs the task of switching ON the UPP module instance in the PSC. But, it does not perform the required pinmux operations. Pinmux operations are the responsibility of the application.

## 2 Requirements

This section in brief lists the most important functional requirements of the UPP driver.

### SR247 uPP Driver for DSP/BIOS

The Driver shall conform to IOM Model of DSP/BIOS Operating System

### SR248 Channel Handles

The Driver shall allow the user to create two GIO\_Handles: one for each channel of the uPP peripheral. These two driver handles must interact gracefully with one another.

### SR249 Device initialization

The driver shall implement a simple driver handle initialization procedure (i.e. GIO\_create with a parameter selecting between channels)

### SR250 Device configuration

The driver shall provide straightforward device configuration to select between valid operating modes (i.e. IO\_CTLs for GIO\_control), callable at any time when no I/O transactions are active. The driver shall initialize its configuration parameters from a user-supplied C struct during system init. The driver shall support every valid combination of the following device parameters: Data Rate: single (SDR), double (DDR) Data Width: 8, 9, 10, ?, 16 bit (9-16 only valid for 16-bit interface) Data Interface Width: 8, 16 bit Data Packing (9-15 bit only): right justify zero extend, right justify sign extend, left justify zero fill Channel Count: 1, 2 Channel Direction: transmit, receive, duplex 0 (channel 1 receive, channel 2 transmit), duplex 1 (channel 1 transmit, channel 2 receive) Data Interleave: disabled, enabled (valid only for 1-channel transmit in SDR and 1-channel transmit or receive in DDR) Loopback: disabled, enabled (duplex mode only)

### SR251 Device configuration (cont'd)

Clock Divisor: 1, 2, ..., 16 (max clock speed: 75 MHz in SDR, 37.5 MHz in DDR) (special case, 2-channel transmit or 2-channel receive in 9-16 bit mode: 50 MHz in SDR, 25 MHz in DDR) Clock Polarity: normal, invert Honor Wait: honor, ignore (transmit only) Wait Polarity: normal, invert Watch Enable: watch, ignore (receive only) Enable/Write Polarity: normal, invert Honor Start: honor, ignore (receive only) Select/Start Polarity: normal, invert Tristate Data Lines While Inactive: tristate, drive (transmit only) DMA Burst Size: 64, 128, or 256 bytes (receive only) Transmit Threshold Size: 64, 128, or 256 bytes (transmit mode only) Driver Mode: Polled, Interrupt-driven Function Call Model (GIO\_submit): Synchronous, Asynchronous

### SR252 Driver modes

The driver shall operate in any of two modes: Synchronous Interrupt-driven: driver blocks caller. When interrupt is received indicating transfer complete, driver unblocks caller and returns. Asynchronous Interrupt-driven: driver does not block caller. When the interrupt happens, driver calls the user's callback to indicate transfer complete.

### SR253 Channel independence

All configuration settings must be applicable to each channel independent of the other. The only exception is if the device is configured in loopback mode: here, all settings should be identical except direction.

### SR254 Idle value in Transmit mode



The driver shall allow the user to specify an "idle" value for transmit mode for each channel (separately) when no I/O transactions are queued.

#### SR255 Start/Stop

The driver shall provide straightforward device start/stop functionality that removes all pending jobs (on stop).

#### SR256 Suspend/Resume

The driver shall provide straightforward device suspend/resume functionality that preserves transfers and queues currently in progress. The driver should finish any transfer currently in progress, then save the remaining queued transfers until operation is resumed.

#### SR257 I/O Callback

In interrupt-driven asynchronous mode, the driver must allow the user to specify a callback function which will be called at completion of I/O

#### SR258 Interrupts

The driver shall allow the user callback function to be triggered by one or more of the following interrupts: End-of-Line End-of-Transfer Underrun/Overrun Error CBA Error DMA Programming Error

#### SR259 Underrun/Overrun

The driver shall alert the user when an underrun/overrun condition arises. In asynchronous, polled mode, the driver must provide functionality for the user to check for these conditions.

#### SR260 DMA control

GIO\_submit shall accept as input a struct that contains the DMA parameters: start address, line length, line offset, line count.

#### SR261 DMA queueing

The driver shall allow the user to queue more than two successive I/O transfers (in asynchronous mode). The driver shall keep an internal list of pending transactions, and automatically begin the next transfer before calling the user callback function. The driver shall make use of the uPP internal DMA controller's inherent ability to queue a second transfer while the first is still running. The maximum queue size (if finite) must be documented in the driver user's guide. A single call to GIO\_submit must be capable of queueing up to 4 DMA transfers simultaneously.

#### SR262 DMA queue clearing

The driver shall support clearing the DMA queue for each channel (i.e. GIO\_abort, GIO\_flush). (Note: to clear the "first" item in each queue, the DMA descriptor registers must be cleared.)

#### SR263 Device reset

The driver shall implement a simple device reset procedure (i.e. reset IOCTL).

#### SR264 Device shutdown

The driver shall implement a simple device shutdown procedure (i.e. GIO\_delete).

#### SR265 Power Management

If power management is enabled: 1. During init, the driver calls power manager API to request all the power resources it requires. 2. When closing, the driver calls power manager API to release all the power resources it requires.

#### SR266 Instrumentation

The driver shall maintain the following statistics (per channel, separately) when compiled with instrumentation enabled. Number of bytes transferred Number of Underruns/Overruns Number of transfers complete Number of lines transferred Number of CBA errors Number of DMA programming errors Also the driver shall support IOCTLs to Query & Clear the statistics.

#### SR268 uPP sample application #1

uPP sample application will be 8-bit loopback (duplex) mode using SDR and asynchronous, interrupt-driven operation.

#### SR269 uPP sample application #2

TBD; utilize EVM hardware (DAC/ADCs).

#### SR270 Driver verification

Where possible, the driver may be verified in loopback mode. Non-loopback tests must be used to verify the following modes of operation: 2-channel transmit 2-channel receive data interleave enabled

#### SR271 Driver throughput: Transmit

60 MB/s (8-bit, single channel, SDR and DDR) 120 MB/s (16-bit, single channel, SDR and DDR)  
120 MB/s (8-bit, dual channel, SDR and DDR) 160 MB/s (16-bit dual channel, SDR and DDR)

#### SR272 Driver throughput: Receive

60 MB/s (8-bit, single channel, SDR and DDR) 120 MB/s (16-bit, single channel, SDR and DDR)  
120 MB/s (8-bit, dual channel, SDR and DDR) 160 MB/s (16-bit dual channel, SDR and DDR)

#### SR273 Driver throughput: Duplex

60 MB/s (per direction) (8-bit, SDR and DDR) 120 MB/s (per direction) (16-bit, SDR and DDR)

#### SR274 Driver throughput: Loopback

60 MB/s (per direction) (8-bit, SDR and DDR) 120 MB/s (per direction) (16-bit, SDR and DDR)

## 2.1 Assumptions

None

## 2.2 Constraints

The UPP device has one "EN" (enable) bit to enable and disable the peripheral, it is required that the device is configured before the peripheral is enabled by setting the EN bit. Because of this limitation, when using the driver in two channel mode, both the channels have to be created before an IO can be programmed on any one of the channels. Otherwise, the UPP does not perform the IO transfer.

---

## 3 Design Description

This chapter deals with the overall architecture of DSP/BIOS UPP 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 Design Philosophy

This device driver is written in conformance to the DSP/BIOS™ IOM device driver model and handles communication to and from the UPP hardware. The driver is designed keeping a device, also called instance, and channel concept in mind.


#### 3.1.1 The Instance Concept

The IOM model provides the concept of the Instance for the realization of the device and its communication path as a part of the driver implementation. The instance Object maintains the state of the UPP device. The instance word usage refers to the driver as one entity. Any detail, configuration parameter or setting which shall apply across the driver shall be an instance variable. For example, loopback mode setting is an instance wide variable. Each hardware instance shall map to one UPP instance. The "instance object" will be instantiated per physical instance of the peripheral. The lifetime of the instance is between its creation using `uppMdBindDev()` and its deletion using `uppMdUnBindDev()`.

#### 3.1.2 The Channel Concept

There are two physical channels available on the UPP. Each of the physical channel corresponds to an channel in the driver. Each of these channels is configurable independently and can be controlled independent of each other. The UPP driver supports up to a maximum of 2 channels. The lifetime of the channel is between its creation using `uppMdCreateChan()` and its deletion using `uppMdDeleteChan()`.

The UPP peripheral needs the instance to maintain its state. The channel object holds the IOM channel state during execution. The software channels created for UPP driver are bound with physical channel of the peripheral.

 *Please note that when the driver is configured to use more than one channel, it is required to create both the channels before attempting to perform an IO operation on any one of the channel. This limitation is because of the fact that the UPP device will be enabled by the driver only after the second channel is created. Also note that this limitation is applicable only in 2 channel modes, in a single channel mode the UPP will be enabled as soon as the channel is created.*

## 3.2 Static View

## 3.3 Functional Partition

The device driver is partitioned into distinct sub-components, consistent with the roles and responsibilities it is expected to perform. In the following sub-sections, each of these functional sub-components of the device driver is further elaborated.

As per the design philosophy, the UPP driver shall be a single layer driver and coupled tightly with the DSP/BIOS operating system. The driver is fully compliant with the IOM driver model of the DSP/BIOS operating system.

### 3.3.1 IOM Interface

A driver which conforms to the IOM driver model exposes a well defined set of interfaces (as given below)

- Driver initialization function.
- IOM Function pointer table.

Hence the UPP driver (because of the virtue of its IOM driver model compliance) exposes the following interfaces.

- Upp\_init()
- Upp\_IOMFXNS

The Upp\_init () is a startup function that needs to be called by the user (application) to initialize all the data structures of the Upp driver. This function also initializes all the instance specific information for the UPP instance like the Base address, interrupt number etc.

***Note:** The working of the UPP driver will be affected if this function is not called by the application prior to accessing the UPP driver APIs.*

The UPP driver exposes an IOM function pointer table which contains the various APIs provided by the UPP driver. The functions that need to be supported by an IOM driver are explained below.

The following table outlines the basic interfaces published by UPP IOM layer.

Function	Description
uppMdBindDev	<p>The mdBindDev function is called by the DSP/BIOS after the bios initialization. The mdBindDev should typically perform the following actions.</p> <ul style="list-style-type: none"> <li>❖ Acquire the device Handle for the specified instance of the Upp on the SoC.</li> <li>❖ Configure the UPP device with the specified parameters( A set of default parameters are also provided).</li> </ul>
uppMdUnBindDev	<p>The mdUnBindDev function is called to destroy an instance of the UPP driver.</p> <ul style="list-style-type: none"> <li>❖ It will unroll all the changes done during the bind</li> </ul>

	operation and free all the resources allocated to the UPP driver.
uppMdControlChan	<p>The mdControlChan function is used to issue a control command to the UPP driver. Please refer to the list of control commands supported by the UPP driver.</p> <p>❖ Typical commands supported are STOP, START etc.</p>
uppMdCreateChan	<p>The mdCreateChan () function is executed in response to the SIO_create() or GIO_create() API call by the application.</p> <p>Application has to specify the mode in which the channel has to be created through the "mode" parameter. The UPP driver supports only two modes of channel creation (input and output) mode for every device instance.(though it will allow both the channels to be configured as transmit or receive simultaneously).</p> <p>❖ The required TX or RX sections (data width, data rate, frequency clock divider values etc.) are setup.</p>
uppMdDeleteChan	<p>The mdDeleteChan () is invoked in response to the GIO_delete () or SIO_delete API call by the application.</p> <p>❖ It frees all the resources allocated during the creation of the channel.</p>
uppMdSubmitChan	<p>The mdSubmitChan () is invoked in response to the GIO or SIO layer provided read/write API calls with the appropriate channel handle and IOM packet containing the operation to be performed and required parameters needed for programming the DMA channels.</p>

### 3.4 Dynamic view

The figure below presents the dynamic view of the driver when processing data.

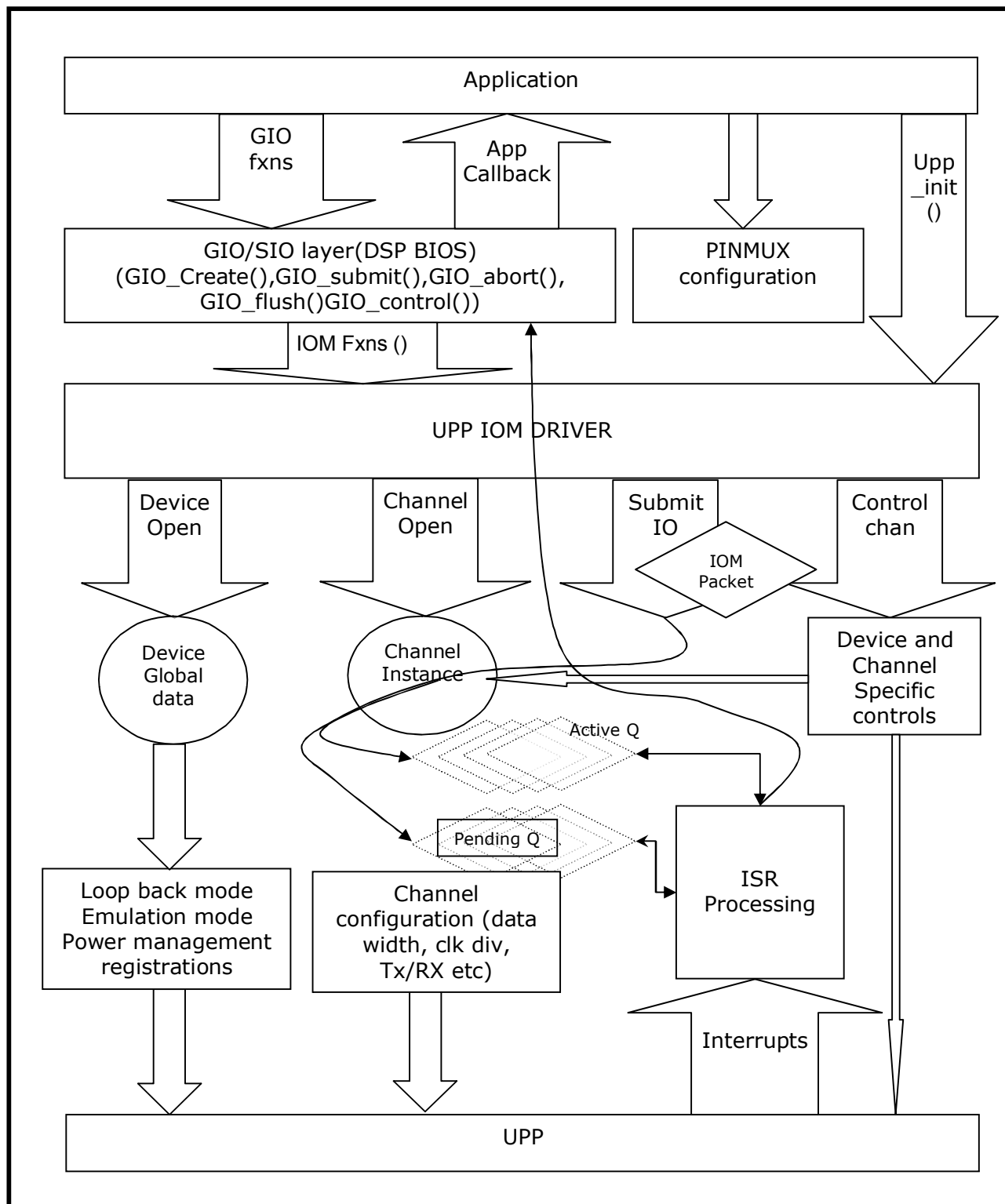


Figure 3 Dynamic view of the UPP driver

The `Upp_init ()` function of the UPP driver is invoked first and is responsible for initializing the device object and channel object structure of the UPP IOM driver. After the driver object is initialized, the driver is instantiated, in response to driver create functions which is the call to `uppMdBndDev()` of UPP IOM driver.

The figure above shows the flow of data from the application to the driver to the underlying physical device. The IO packet shown in the figure is standard structure used to submit the I/O requests to the IOM layer of the UPP driver. It contains pointer to the data buffer, size of the buffer and the status of the request. The pointer to the data buffer is a pointer to a special structure that contains the information required for programming the packet in to the UPP internal DMA descriptors.

Before data communication between an application and a device can begin, a channel instance handle must be obtained by the application by a call to `GIO_create ()` API. The channel handle represents a unique communication path between the application and UPP device driver. All subsequent operations that communicate to the driver shall use this channel handle. A channel object typically maintains data fields related to a channel's mode, I/O request queues, and possibly driver state information. Application should relinquish channel resources by deleting all channel instances when they are no longer needed through a call to `GIO_delete ()`.

Application shall call `GIO_submit ()` API to submit read/write I/O request to driver. The Device Independent layer shall construct an I/O packet and submits the packet to the IOM layer to do the I/O operation. When a mini-driver completes its processing, usually in an ISR context, it calls a previously registered callback function to pass the IO packet back to the device independent layer of the UPP driver and the device independent layer of the driver in turn calls the application specified callback for that particular I/O request(if registered). The submit/callback function pair handles the passing of IO packets between the application and the UPP IOM layer of the driver. Before an IO packet is passed back to the upper layer driver, the mini-driver must set the completion status field and the data size field in the IO Packet. This status value and size are returned to the application call that initially made the I/O request.

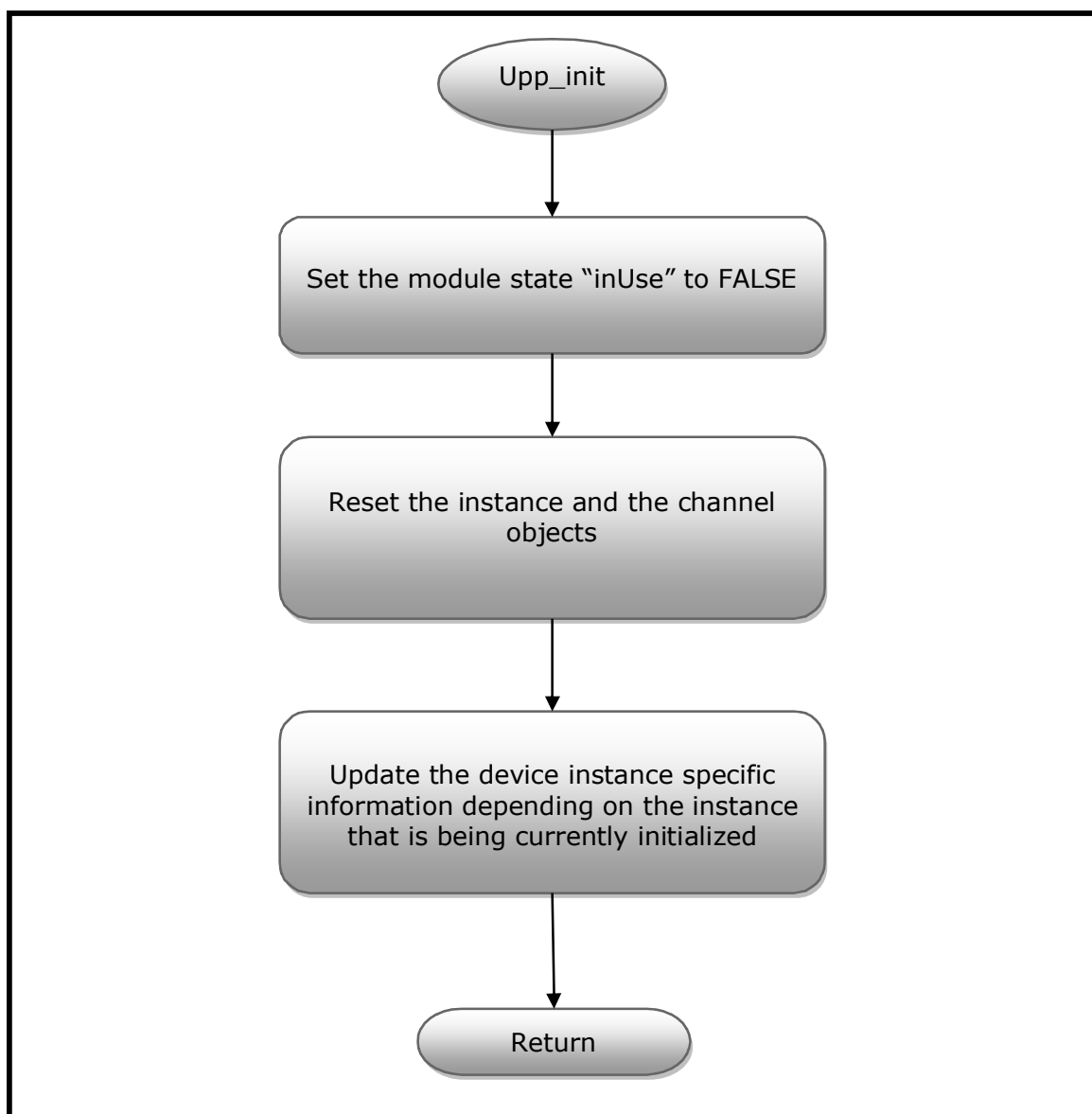
### **3.4.1 Driver Creation (Driver Initialization and Binding)**

The UPP IOM driver needs the global data used by the UPP driver to be initialized before the driver can be used. The initialization function for the UPP driver is not included in the `IOM_Fxns` table, which is exported by the UPP driver; instead a separate extern is created for use by the DSP/BIOS. The initialization function is responsible for initialization of the following instance specific information

- Base address for the instance
- CPU event numbers
- Module clock value
- LPSC number for the module in the Power sleep controller.

The function also sets the "inUse" field of the UPP instance module object to FALSE so that the instance can be used by an application which will create it. It will also initialize all the module level global variables.

Please refer to the figure below for the typical control flow during the initialization of the driver. Refer to the section 3.6 for the API reference for the initialization function.



**Figure 4 Driver Initialization**

The binding function (uppMdBindDev) of the UPPP IOM mini-driver is called in case of a static or dynamic creation of the driver. In case of dynamic creation application will call DEV\_createDevice () API to create the device instance. Otherwise, the instance could be created statically through a tcf file. Each driver instance corresponds to one hardware instance of the UPP. This function shall typically perform the following actions:

- Check if the instance being created is already in use by checking the Module variable "inUse".
- Update the instance object with the user supplied parameters.
- Initialize all the channel objects with default information.
- Initialize the queues used to hold the pending packets and currently executing packets (active queue).



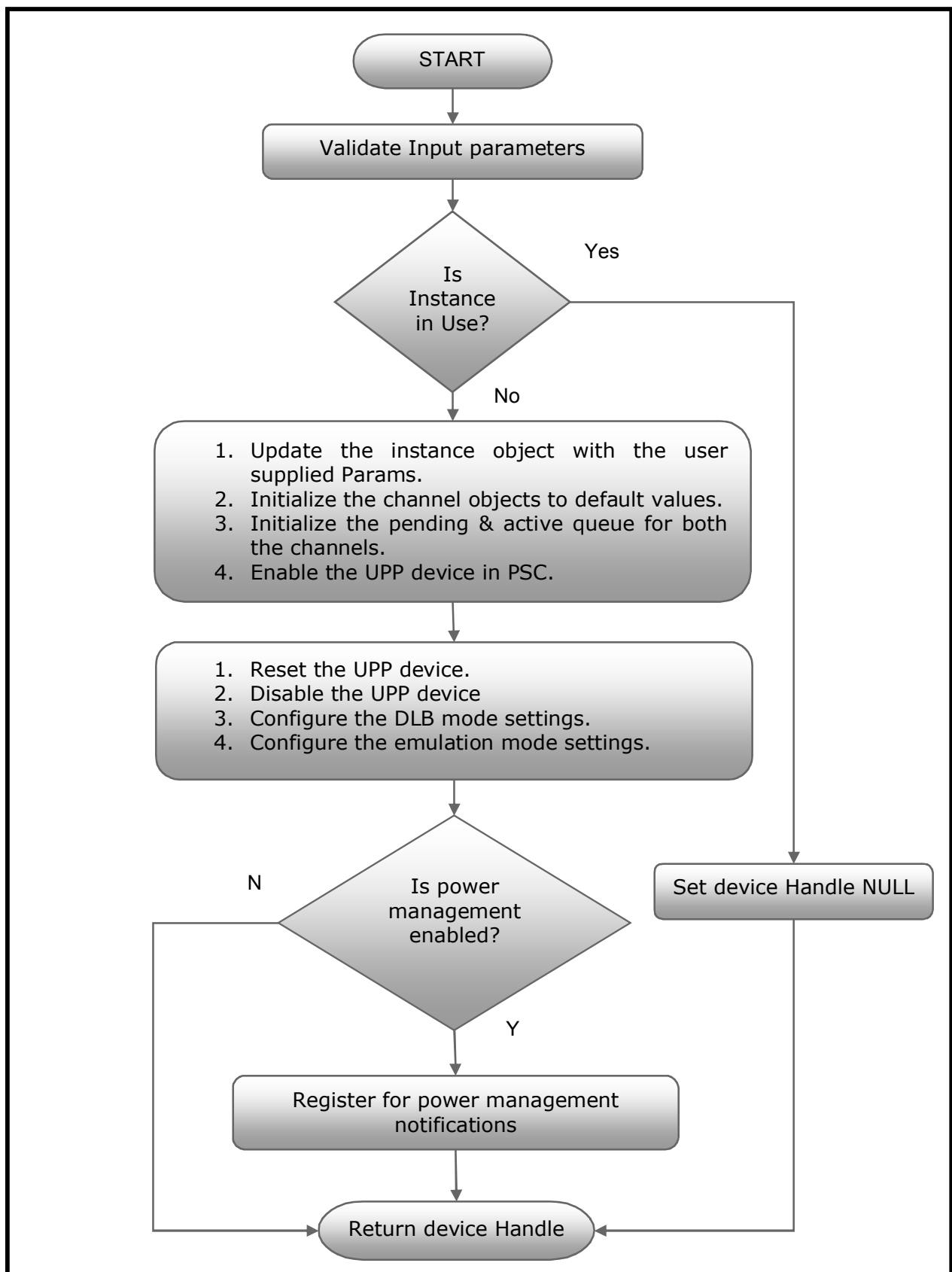
- 
- Enable the UPP device instance in the PSC module.
  - Reset the UPP device and disable the device.
  - Configure the UPP for the DLB mode if applicable and also to set the emulation mode settings.
  - Switch off the module if power management is enabled.
  - Registration of the driver with the PWRM module of DSP BIOS if power management support is requested. Constraints if any will be registered during the creation of the driver channel.
  - Return the device handle to the GIO/SIO layer.

**Note:** The Driver binding operation expects the following parameters

1. Pointer to hold the device handle.
2. Instance number of the instance being created.
3. Pointer to the user provided device parameter structure required for the creation of the device instance.

The user provided device parameter structure will be of type "**Upp\_Params**". Refer to Upp\_Params section for more details.

Please refer to the Figure below for the control flow in the driver during the Bind operation.


**Figure 5 Driver Instance Binding**

### 3.4.2 Channel Creation

The application once it has created the device instance, needs to create a communication channel for transactions with the underlying hardware. As such a channel is a logical communication interface between the driver and the application. Also, note that a channel will correspond to a physical Channel on the UPP. Hence an application can create at most 2 channels.

The UPP IOM driver allows at most two channels to be created. They are

1. CHANNEL A – physical channel A (can be configured as transmit or receive).
2. CHANNEL B – physical channel B (can be configured as transmit or receive).

The application can create a communication channel by calling `GIO_create()`/`SIO_create()` API which in turn calls UPP IO mini driver's `uppMdCreateChan()` function. The application shall call `GIO_create()`/`SIO_create()` with the appropriate "mode" (`IOM_INPUT` or `IOM_OUTPUT`) parameter for the type of the channel to be created (Reception or transmission).

Channel created with mode `IOM_OUTPUT` will be used for transmission of data whereas the channel created with mode `IOM_INPUT` will be used for receiving data. The user can supply the parameters which will characterize the features of the channel (e.g. data rate, bit width etc). The user can use the "Upp\_ChanParams" structure to specify the parameters to configure the channel.

*⚠ The driver will not perform any check on the number of channels created and the operation mode of the driver. it is the responsibility of the application to do so. Failure to do so can result in problems with the driver operation. e.g. if the driver is created as Channel\_A\_XMT mode then only a single channel should be created. if more than a channel is required then an appropriate driver mode should be selected.*

The `uppMdCreateChan()` function typically does the following.

- It validates the input parameters given by the application.
- It checks if the requested channel is already opened or not. If it is already opened the driver will flag an error to the application else the requested channel will be allocated.
- It updates the appropriate channel objects with the user supplied parameters.
- The UPP is configured with the appropriate parameters for the channel.
- If the driver supports power management, then the constraints for the various set points are calculated and registered.
- The clock divider settings for the channel are configured.
- The UPP device is enabled if this is the last channel to be created.
- Registration of the interrupt handler is performed.
- If the complete process of channel creation is successful, then the application will be returned a unique Handle. This Handle should be used by the application for further transactions with the channel. This Handle will be used by the driver to identify the channel on which the transactions are being requested.

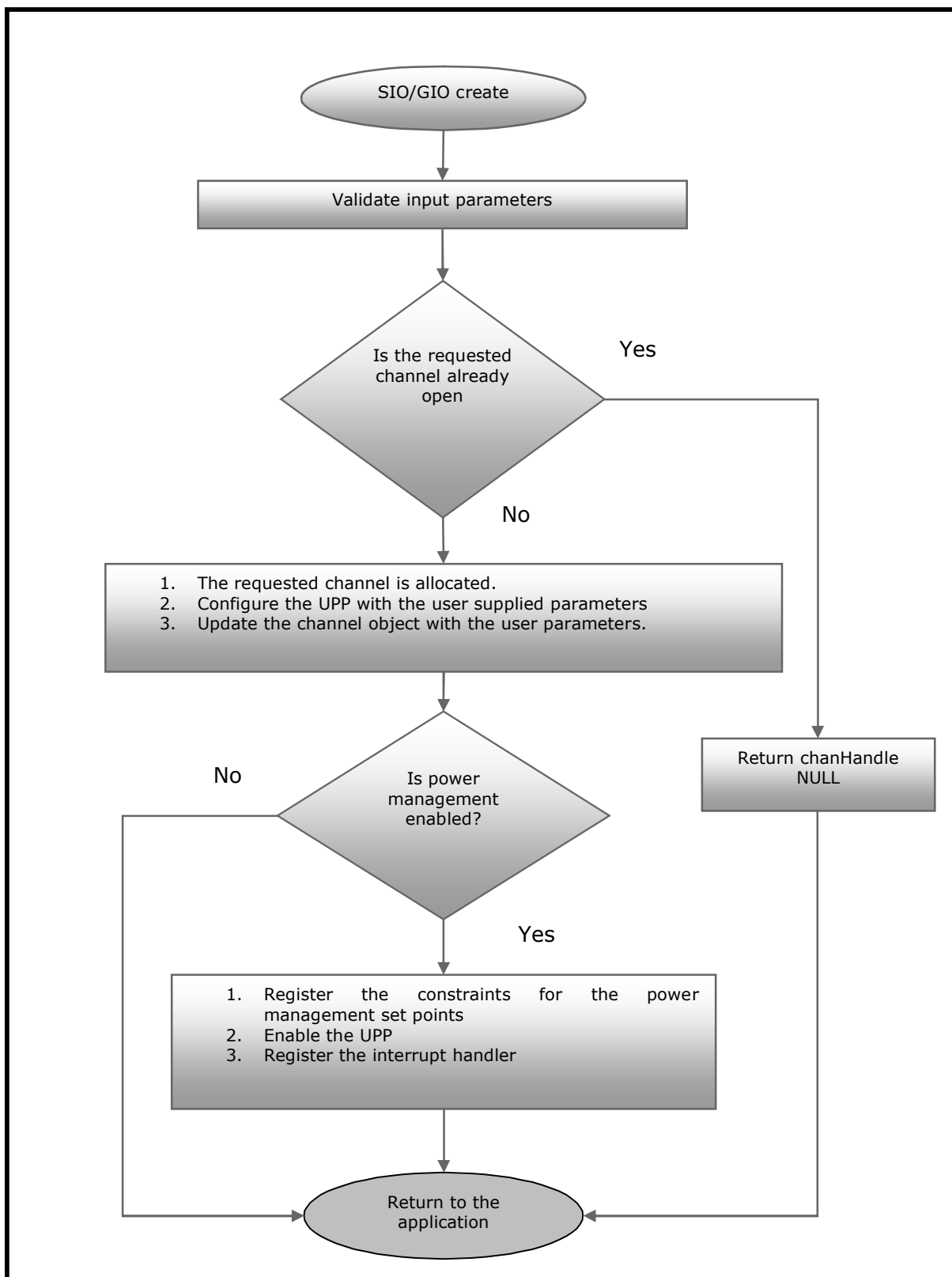


Figure 6 UPP channel create command Flow

### 3.4.3 IO submit

UPP IOM driver provides an interface to submit IO packets for the IO transactions to be performed. Application invokes `GIO_read ()` and `GIO_write ()` APIs for data transfer using UPP device. These APIs in turn creates and submits an IOM packet containing the all the transfer parameters needed by the IOM driver to program the underlying hardware for data transfer. The `mdSubmitChan` function of the UPP IOM driver must handle all the command codes passed to it as part of the `IOM_Packet` structure.

The command codes to be supported by the UPP IOM mini-driver are: `IOM_READ`, `IOM_WRITE`, `IOM_ABORT`, and `IOM_FLUSH`.

- **IOM\_READ.** Drivers that support input channel must implement `IOM_READ`.
- **IOM\_WRITE.** Drivers that support output channel must implement `IOM_WRITE`.
- **IOM\_ABORT and IOM\_FLUSH.** To abort or flush I/O requests already submitted, all I/O requests pending in the mini-driver must be completed and returned to the device independent layer. The `mdSubmitChan` function should dequeue each of the I/O requests from the mini driver's channel queue. It should then set the size and status fields in the `IOM_Packet`. Finally, it should call the callback function registered for the channel for the channel.

✎ *The behavior of the driver will be same for both the ABORT and FLUSH i.e. all the packets will be aborted and returned back to the application.*

The `mdSubmitChan` function of the UPP driver typically performs the following activities.

1. The input packet is validated.
2. If the driver supports power management then the PSC dependency count for the module is increased.
3. If the driver already has sufficient packets then the current IO packet is loaded in to the pending queue.
4. Otherwise the IOP is programmed in to the descriptors of the internal DMA.

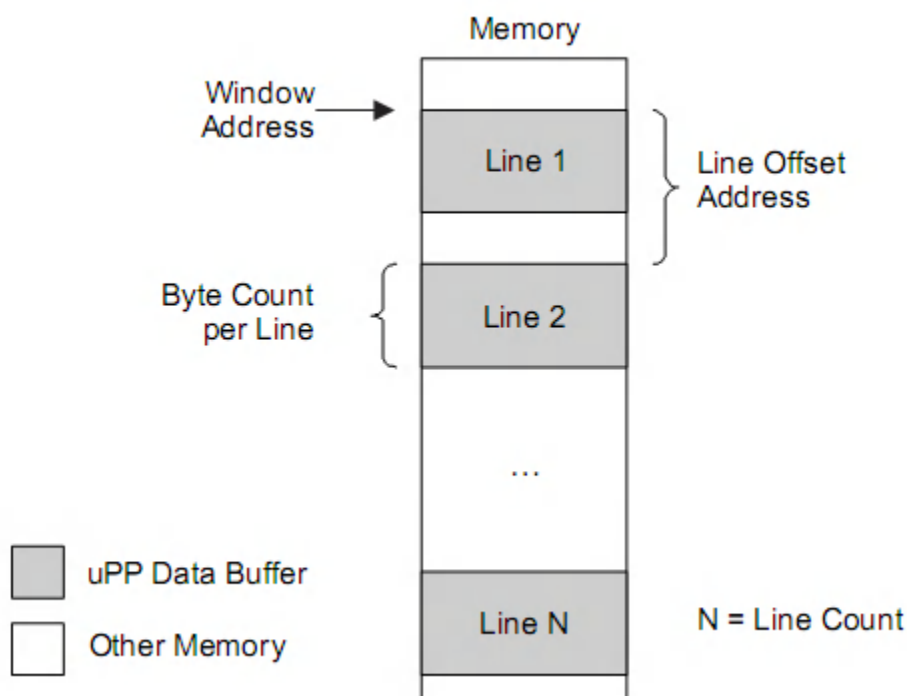
The UPP driver expects the IO packets to be submitted in the following format

```
typedef struct Upp_tranParam_t
{
    Ptr          windowAddr;
    Uint16       bytesPerLine;
    Uint16       lineCount;
    Uint16       lineOffset;
}Upp_transParam;
```

The UPP device has two internal DMA channels I and Q each servicing the physical channels A and B respectively (unless the channel A is configured in interleaved mode where both the DMA channels service only channel A). Each of the DMA channels has two set of descriptors which can be programmed with two IO requests. Hence each of the DMA channel can hold a maximum of 2 packets allowing UPP to transfer data continuously.

### **DMA programming concepts**

The figure below shows a typical buffer located in the memory.



A typical request for this kind of Upp data to be transferred by the application will be as given below

```
Upp_transParam transParam;

transParam.windowAddr = Window address; /* start data */
transParam.bytesPerLine = Byte Count per line;
transParam.lineCount = N;
transParam.lineOffset = line address offset;

SIO_issue(channel Handle, &transParam, (SIZE * N), NULL));
```

### **Special cases**

1. If the data buffer is continuous then the line offset address is equal to the byte count per line.
2. If the data buffer consists of a single line then the line offset address is set to Zero.

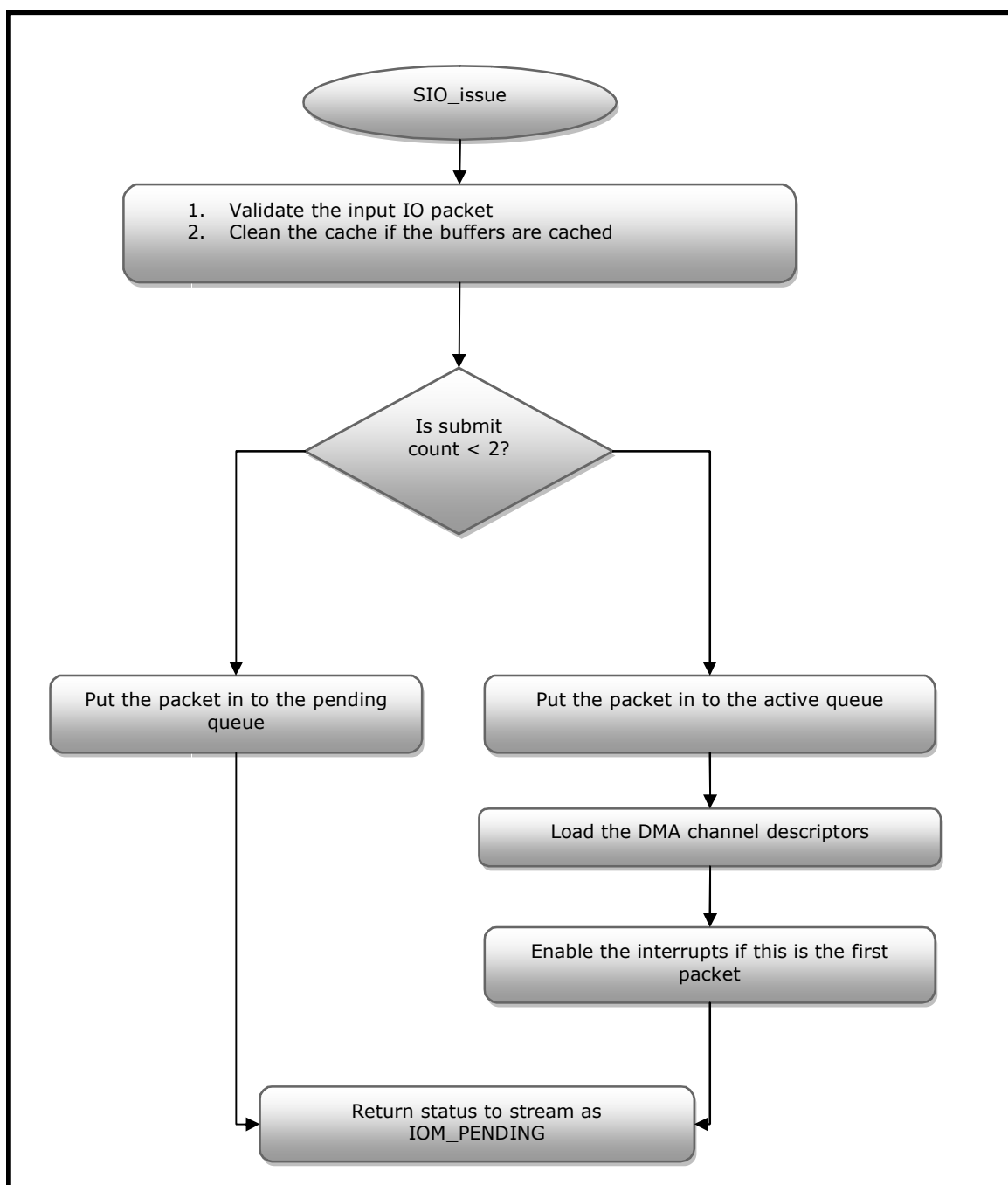


Figure 7 UPP driver submit control flow

### 3.4.4 Control Commands

UPP IOM driver implements device specific control functionality which may be useful for any application, which uses the UPP IOM driver. Application may invoke the control functionality through a call to `GIO_control ()`. UPP IOM driver supports the following control functionality.

The typical control flow for the UPP control function is as given below.

- Validate the command sent by the application.
- Check if the appropriate arguments are provided by the application for the execution of the command.
- Process the command and return the status back to the application.

The below table lists the control commands supported by the UPP driver

Command	Command Argument	Explanation
<code>Upp_Ioctl_START</code>	None	Starts the Upp device and makes it ready for transactions.
<code>Upp_Ioctl_STOP</code>	None	Stops the Upp device.
<code>Upp_Ioctl_QUERY_STATS</code>	<code>Upp_devStats *</code>	Retrieves the device statistics from the driver.
<code>Upp_Ioctl_CLEAR_STATS</code>	None	Clears all the driver internal statistics.
<code>Upp_Ioctl_SET_TIMEOUT</code>	<code>Uint32 *</code>	Configures the genric timeout used by the Upp driver to a user supplied value.

The basic control flow for the handling of the control commands for the driver is shown below. Please not that the individual command handling is not detailed here.



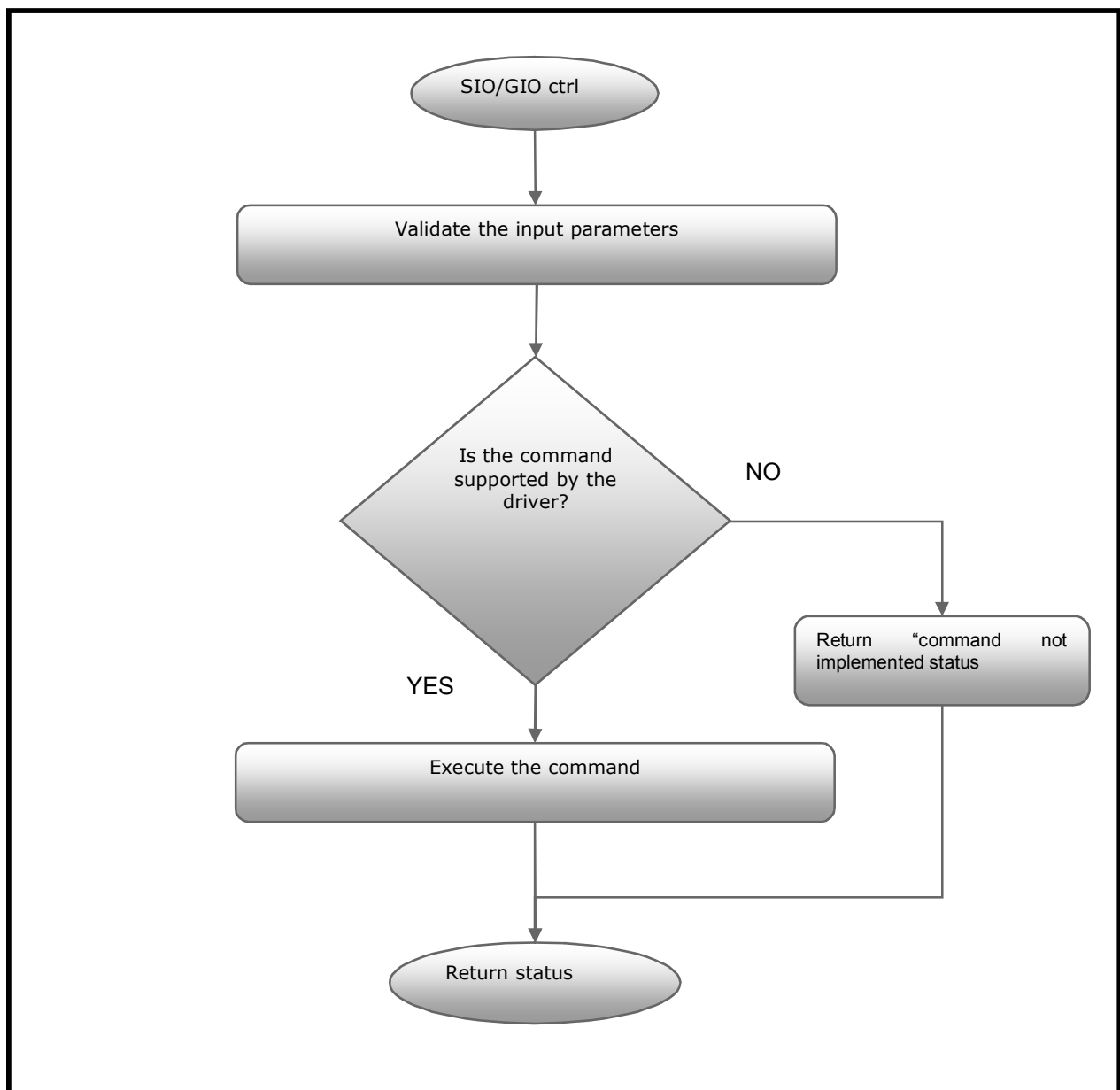


Figure 8 UPP Control command Flow

#### **3.4.5 Channel deletion**

The channel once it has completed all the transaction can close the channel so that all the resources allocated to the channel are freed. The Upp driver provides the "uppMdDeleteChan" API to delete a previously created UPP channel.

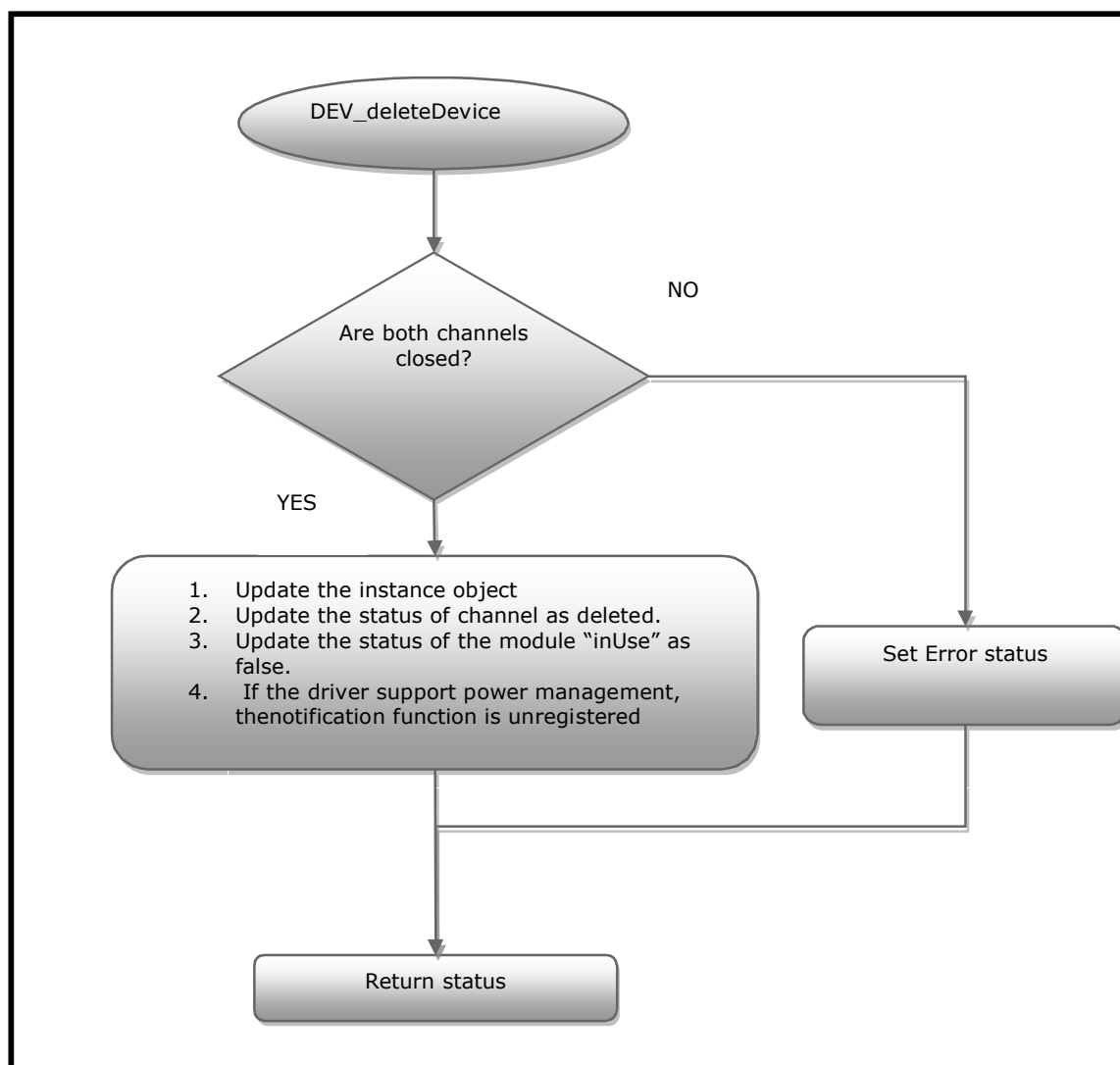
The typical activities performed during the channel deletion are as follows

1. Checking if the channel is already open.
2. If the driver supports power management then all the constraints registered by this channel are unregistered.
3. The state of the channel is set to "closed".
4. If the other channel is already closed then the interrupt handler is unregistered.

### 3.4.6 Driver unbinding/deletion.

The UPP driver provides the interfaces for deleting the driver instance. The `uppUnBindDev` function de-allocates all the resources allocated to the driver during the driver binding operation. The typical operations performed by the unbind operation are as listed below.

- Check if both channels A and B are closed.
- Update the instance object.
- Set the status of the driver to "DELETED".
- Set the status of the module "inUse" to FALSE (so that it can be used again).
- Switch off the module in the PSC.
- Unregister the notification registered with the BIOS power management module(only if the driver supports power management)

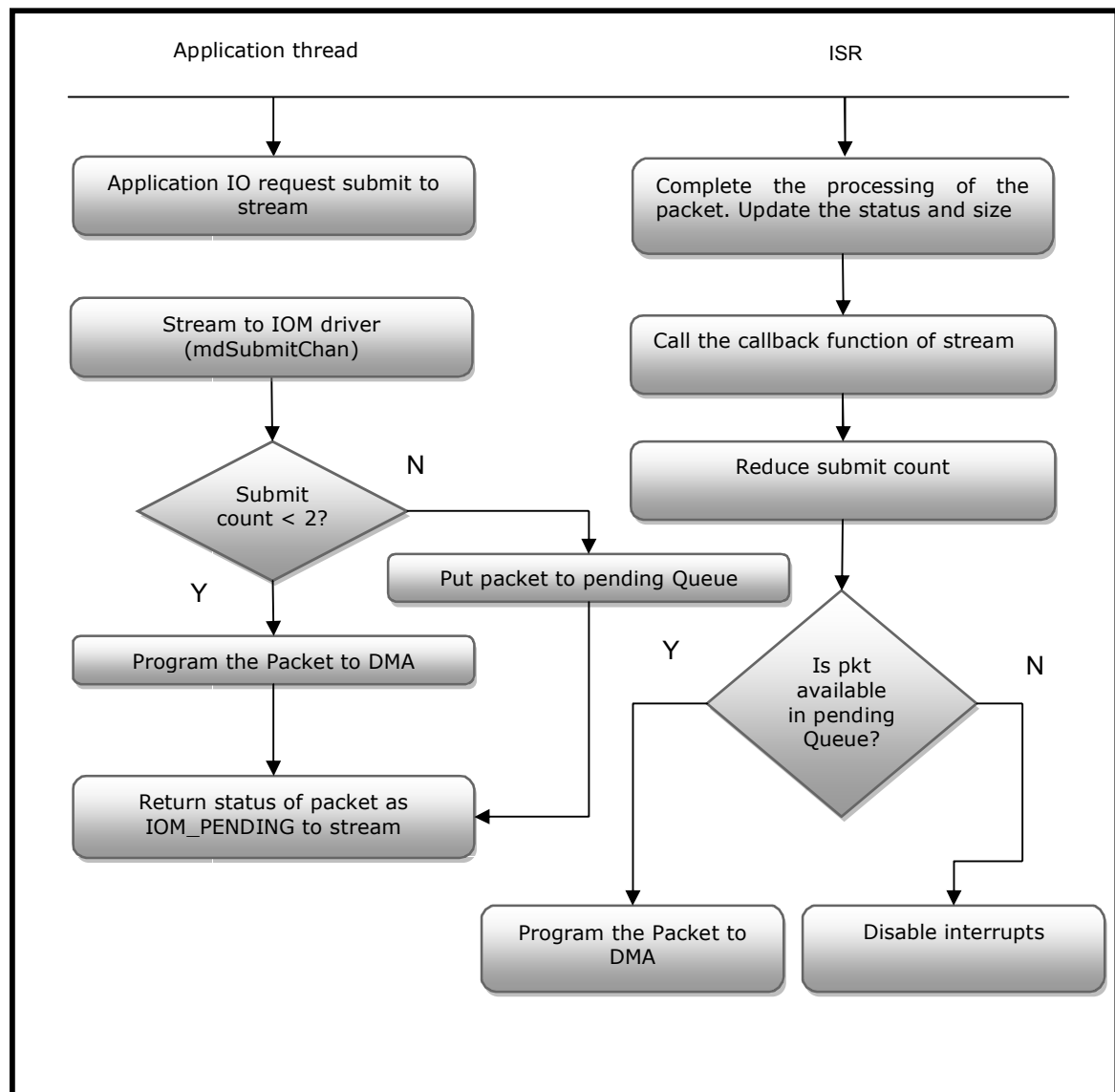


**Figure 9 UPP Driver deletion**

### 3.4.7 Asynchronous IO Mechanism

The UPP IOM driver supports only asynchronous IO mechanism. In Asynchronous IO mechanism multiple IO requests can be submitted by the application thread without causing it to block while waiting for the IO request to complete. Application can submit multiple I/O requests using the "GIO\_read ()" or "GIO\_write ()" (also SIO) APIs and specify a callback function to be called after the IO transfer is completed.

The driver programs the IO requests in to the DMA descriptors and once all the DMA descriptors are programmed the remaining packets are placed in to the pending queue. Whenever an IO request is completed, the interrupt handler then calls the callback function of the stream layer which in turn calls the callback function registered by the application to inform about the completion of the packet processing. Then the driver loads the next packet from the pending queue if available.



**Figure 10 UPP driver Asynchronous IO**

---

## 3.5 Constants & Enumerations

### 3.5.1 Upp\_POLLED\_RETRYCOUNT

This constant defines the retry count to be used by the driver when it is waiting for some operation to complete. Usually the driver will give an error if the operation does not complete within the timeout specified.

#### Definition

```
#define Upp_POLLED_RETRYCOUNT    (0xFFFFu)
```

#### Comments

None

#### Constraints

Please check that this value should be minimum of 200 CPU clock cycles as this is during the reset of the UPP device

#### See Also

None

### 3.5.2 Upp\_MAXLINKCNT

This constant defines the max number of packets that will be available at any given time in the active queue of the driver.

#### Definition

```
#define Upp_MAXLINKCNT            (2u)
```

#### Comments

None

#### Constraints

As the UPP internal DMA supports queuing a maximum of 2 packets this Macro value will also be constant.

#### See Also

None

### 3.5.3 Upp\_DevMode

This Enumeration defines the operating mode of the UPP device driver.

#### Definition

```
typedef enum Upp_DevMode_t
{
    Upp_DevMode_CHAN_A_RCV      = (0u) ,
    Upp_DevMode_CHAN_A_XMT      = (1u) ,
    Upp_DevMode_CHAN_AB_RCV     = (4u) ,
    Upp_DevMode_CHAN_AB_XMT     = (5u) ,
    Upp_DevMode_DUPLEX_0        = (6u) ,
    Upp_DevMode_DUPLEX_1        = (7u)
}Upp_DevMode;
```

#### Comments

None

#### Constraints

Device should be created with appropriate mode to enable the use of proper physical channels.

#### See Also

Upp\_Params

### 3.5.4 Data Structures

### 3.5.5 Driver Instance Object

This structure is the Upp driver's internal data structure. This data structure is used by the driver to hold the information specific to the instance. There will be one unique instance object for every instance of the Upp controller supported by the driver.

This data structure holds the information pertaining to an instance of the Upp device. It holds information like the current device state, handle to the UPP channels etc. The data structure is initialized during "mdBindDev", which is called during DSP-BIOS initialization, and is persistent till it is invalidated by "mdUnBindDev".

#### **Definition**

```
typedef struct Upp_Object_t
{
    Int32                instNum;
    Upp_DriverState      devState;
    Upp_Params           devParams;
    Upp_HwInfo           hwInfo;
    Uint32               retryCount;
    Upp_ChanObj          chanA;
    Upp_ChanObj          chanB;
    Uint32               upicrRegVal;
    Bool                isrRegistered;
    volatile Bool        uppSmState;
    Upp_pwrMInfo         pwrMInfo;
} Upp_Object;
```

#### **Fields**

<i>instNum</i>	Instance number of the Upp.
<i>devState</i>	Current state of the driver (Created/Deleted).
<i>devParams</i>	Device creation parameters supplied by the application.
<i>hwInfo</i>	Structure holding the hardware information related to the instance (e.g. interrupt numbers, base address etc).
<i>retryCount</i>	Retry count to be used by the driver when waiting in indefinite loops. (E.g. waiting for the device to be out of reset etc.)
<i>chanA</i>	channel object for the physical channel A.
<i>chanB</i>	channel object for the physical channel B.

<i>upicrRegVal</i>	Value written to the UPICR register.
<i>isrRegistered</i>	Variable to indicate if the ISR is registered already.
<i>uppSmState</i>	Upp driver state machine state.(TRUE = Stopped, FALSE = Running).
<i>pwrMInfo</i>	Power management information for the Upp device.

### **Comments**

1. The Upp Driver works only in the interrupt - internal DMA mode of operation.
2. One instance object represents one instance of the driver.
3. "pwrMInfo" is used only if power management support is enabled for the driver.

### **Constraints**

None

### **See Also**

*Upp\_ChanObj*, *pwrMInfo*, *hwInfo*

## **3.5.6 Channel Object**

This structure is the Upp driver's internal data structure. This data structure is used by the driver to hold the information specific to the channel. There will be at most two channels supported per instance(one for channel A and one for channel B).it is used to maintain the information pertaining to the channel like the current channel state, callback function etc. This structure is initialized by mdCreateChan and a pointer to this is passed down to all other channel related functions. Lifetime of the data structure is from its creation by mdCreateChan till it is invalidated by mdDeleteChan.

### **Definition**

```
typedef struct Upp_ChanObj_t
{
    Int16                                mode;
    Upp_DriverState                      chanState;
    volatile Bool                        flushAbort;
    Ptr                                  devHandle;
    IOM_TiomCallback                    cbFxn;
    Arg                                  cbArg;
    Uint32                               submitCount;
    QUE_Obj                             queActiveList;
    QUE_Obj                             quePendList;
    Upp_ChanParams                      chanParams;
    Upp_devStats                        stats;
} Upp_ChanObj;
```



### **Fields**

<i>mode</i>	Current operating mode of the channel (INPUT/OUTPUT).
<i>chanState</i>	Current state of the channel (opened/closed).
<i>flushAbort</i>	Variable to indicate if the channel is currently executing a flush or abort command.
<i>devHandle</i>	Pointer to the instance object.
<i>cbFxn</i>	Callback function pointer
<i>cbArg</i>	Callback function argument
<i>submitCount</i>	Number of IO packets currently held by the channel.
<i>queuePendList</i>	Queue for holding the pending packets.
<i>queueActiveList</i>	Queue for Holding the currently executing packets.
<i>chanParams</i>	Application supplied configuration for the channels.
<i>stats</i>	Structure to hold the statistics for the physical channel.

### **Comments**

1. Only 2 channels are supported (corresponding to channel A and channel B).
2. "stats" to be supported requires "Upp\_STATISTICS\_ENABLE" to be enabled.

### **Constraints**

1. **If the device is to be operated in two channel mode, then both the channels need to be created before the IO can be processed on any one of the channels. This is because the Upp peripheral is enabled only after both the channels are configured.**

### **See Also**

*Upp\_Object, Upp\_chanParams.*

### 3.5.7 Upp\_Params

This structure is used to supply user parameters during the creation of the driver instance. During the creation of the driver using the static creation or dynamic creation the user needs to supply the above structure with the required parameters. The structure is as defined below

#### Definition

```
typedef struct Upp_Params_t
{
    Upp_EmuMode           emulationMode;
    Upp_Loopback          dlbMode;
    Upp_DevMode           devMode;
    Bool                  pscPwrMEnable;
    Uint32                 inputFreq;
    Upp_pllDomain         pllDomain;
} Upp_Params;
```

#### Fields

<i>emulationMode</i>	Emulation mode setting to be used.
<i>dlbMode</i>	Loop back mode selection
<i>devMode</i>	Device operating mode (1/2 channels,xmt,rcv etc)
<i>pscPwrMEnable</i>	Power management support is to be enabled or not.
<i>inputFreq</i>	Input clock frequency
<i>pllDomain</i>	PLL domain where the Upp source clock is configured to.

#### Comments

1. The Upp Driver works only in the Interrupt internal DMA mode of operation.
2. Please refer to different operating modes supported by the driver given below

1. *Upp\_DevMode\_CHAN\_A\_RCV*
2. *Upp\_DevMode\_CHAN\_A\_XMT*
3. *Upp\_DevMode\_CHAN\_AB\_RCV*
4. *Upp\_DevMode\_CHAN\_AB\_XMT*
5. *Upp\_DevMode\_DUPLEX\_0*
6. *Upp\_DevMode\_DUPLEX\_1*

#### Constraints

None

### See Also

*Upp\_chanParams*

### 3.5.8 Upp\_CharParams

This structure is used to supply user parameters during the creation of the channel instance. During the creation of the channel, user needs to supply the above structure with the appropriate parameters as per his mode of operation. The structure is as defined below

#### Definition

```
typedef struct Upp_CharParams_t
{
    Bool                enableCache;
    Uint16              hwiNumber;
    Upp_chanSel         chanSel;
    Upp_bitWidth         bitWidth;
    Upp_dataRate         dataRate;
    Upp_ChanMode         chanMode;
    Upp_dataPackFmt      dataPackFmt;
    Uint32              opFreq;
    Uint16              idleVal;
    Upp_ErrCallback     userCbFxn;
    Arg                 userCbArg;
    Upp_fifoThreshold    fifoThresHold;
    Upp_signalConfig     signalCfg;
} Upp_CharParams;
```

#### Fields

<i>enableCache</i>	Option to enable the cache operations on the user supplied buffers.
<i>hwiNumber</i>	HWI number for the ECM group in which the event is configured
<i>chanSel</i>	Physical channel to be used (A or B)
<i>bitWidth</i>	Width of the data to be received or transmitted.
<i>dataRate</i>	Data rate to be used by the channel(single or double)
<i>dataPackFmt</i>	Data packing format to be used(left justified, right justified etc)
<i>opFreq</i>	Output bit clock frequency.

---

<i>idleVal</i>	Idle pattern to be transmitted when no data is available with the Upp
<i>userCbFxn</i>	User specified callback function to be called in case of interrupts.
<i>userCbArg</i>	User specified argument to be passed to the callback function.
<i>fifoThreshold</i>	FIFO threshold for this channel
<i>signalCfg</i>	Configuration of the START, WAIT and ENABLE signals.

#### **Comments**

1. "userCbFxn" function will be called in the ISR context hence appropriate care should be taken that the function confirms to the ISR coding guidelines.
2. "hwiNumber" needs to be specified according to the ECM event group that the channel being configured falls in to.
3. Please note that all the different combination of channel parameters may not be valid. Refer to the Upp SPRU for the valid configurations.

#### **Constraints**

Check the SPRU for the allowed limits for each of the channel parameters.

#### **See Also**

SIO\_create (), GIO\_Create ()

### **3.5.9 Upp\_signalConfig**

This structure is used to configure the WAIT, ENABLE and START signals for a given UPP channel. The individual signal can be enabled or disabled. Also, we can specify the polarity of the required signals. Polarity of the signals not used will be don't care for the driver.

#### **Definition**

```
typedef struct Upp_signalConfig_t
{
    Bool                startEnable;
    Upp_polarity        startPinPol;
    Bool                enaEnable;
    Upp_polarity        enablePinPol;
    Bool                waitEnable;
    Upp_polarity        waitPinPol;
    Upp_clkPol          clkPol;
    Upp_PinIdleState    pinIdleState;
} Upp_signalConfig;
```

**Fields**

<i>startEnable</i>	START Pin to be used or not
<i>startPinPol</i>	start pin polarity
<i>enaEnable</i>	Option to select if ENABLE pin is to be used or not
<i>enablePinPol</i>	Polarity for the enable pin
<i>waitEnable</i>	Option to select/disable the WAIT pin
<i>waitPinPol</i>	Wait pin polarity
<i>clkPol</i>	clock polarity
<i>pinIdleState</i>	Idle state of the pin (tri-stated or idle value transmitted etc)

**Comments**

1. START, ENABLE, WAIT pins can be enabled or disabled as per the system requirements and the operational mode of the channels
2. If the Idle pin is selected to transmit an idle value, the idle value to be transmitted can be specified in the channel parameters.

**Constraints**

None

**See Also**

Upp\_chanParams

## 3.6 API Definition

### 3.6.1 Upp\_init

**Syntax**

```
Void Upp_init(Void);
```

**Arguments**

IN	None
	No input arguments

**Return Value**

None	No return value
------	-----------------

**Comments**

Function to initialize the Upp driver data structures. This function initializes the hardware information per instance like CPU event numbers, base address etc.

It also initializes the instance and the channel objects of the required instance.

**Constraints**

- This function should be called by the application before creating any Upp driver instance.
- This function should be called only once in the life time of the Application.

**See Also**

None

**Note:** Please refer to the section 3.2.1 for the other Interfaces provided by the Upp driver.

## **4 Decision Analysis & Resolution**

### **4.1 DAR Criteria**

1. Handling of the PEND bit and loading of the next packet in the UPP driver

#### **4.1.1 Alternative 1**

- Wait for EOW interrupt and then wait for PEND bit to get reset to load the next packet.

#### **4.1.2 Alternative 2**

- Enable EOL interrupt and check if the PEND bit is reset in the EOL and EOW interrupt.

### **4.2 Decision**

It has been decided to go ahead with the alternative 2. It is not advisable to wait for the PEND bit to be reset in the ISR context. Hence it is decided that EOL interrupt will be enabled during the loading of the packet and subsequently when the packet is being completed. During the loading of new packet in ISR the EOL interrupt will be disabled.

## **5 Revision History**

<b>Version #</b>	<b>Date</b>	<b>Author Name</b>	<b>Revision History</b>
0.1	11 Jan 2010	Imtiaz SMA	Created Newly for the OMPL138 UPP Driver.

««« § »»»»