DSP/BIOS I2C Device Driver

Architecture/Design Document

Revision History

Document Version	**Author(s)**	**Date**	**Comments**
0.1	Chandan Kr Nath	September 9 2008	Created the document
0.2	Madhvapathi Sriram	October 15, 2008	Revised for release 2.00.00.03
0.3	Chandan Kr Nath	December 9, 2008	Revised for release 2.00.00.04
0.4	Imtiaz SMA	January 21, 2009	Updated the document for the IOM driver and IDriver contrast sections

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 OMAPL138

Note: The usage of structure names and field names used throughout this design document is only for indicative purpose. These names shall not necessarily be matched with the names used in source code.

1.1 Terms and Abbreviations

Term	Description
API	Application Programmer's Interface
CSL	TI Chip Support Library – primitive h/w abstraction
IP	Intellectual Property
ISR	Interrupt Service Routine
OS	Operating System

1.1 Disclaimer

This is a design document for the I2C driver for the DSP/BIOS operating system. Although the current design document explain the I2C driver in the context of the BIOS 6.x driver implementation, the driver design still holds good for the BIOS 5.x driver implementation as the BIOS 5.x driver is a direct port of BIOS 6.x driver. The BIOS 5.x drivers conform to the IOM driver model whereas the BIOS 6.x drivers confirm to the IDriver model. The subsequent section explains how this document can be used to understand and modify the IOM drivers found in this product. Please note that all the flowcharts, structures and functions described here in this document are equally applicable to the I2C driver 5.x.

1.2 IOM driver Vs IDriver

The following are the main difference between the BIOS 5.x and BIOS 6.x driver. Please refer to the reference documents for more details in the IOM driver model.

1. All the references to the stream module should be treated as references to a module that provides data streaming. In BIOS 5.x the equivalent modules are SIO and GIO.
2. This document refers to the IDriver model supported by the BIOS 6.x.All the references to the IDriver should be assumed to be equivalent to the IOM driver model.

3. The BIOS 6.x driver uses a module specifications file (*.xdc) for the declaration of the enumerations, structures and various constants required by the driver. The equivalent of this xdc file is the header file XXX.h and the XXXLocal.h.

   Note: The XXXLocal.h file contains all the declaration specified in the “internal” section of the corresponding xdc file.

4. In BIOS 6.x creation of static driver instances follow a different flow and cause functions in module script files to run during build time. In BIOS 5.x creation of driver (for both static and dynamic instances) result in the execution of the mdBindDev function at runtime. Therefore any references to module script files (*.xs files) can be ignored for IOM drivers.

5. The XXX_Module_startup function referenced in this document can be ignored for IOM drivers.

6. IOM drivers have an XXX_init function which needs to be called by the application once per driver. This XXX_init function initializes the driver data structures. This application needs to call this function in the application initialization functions which are usually supplied in the tci file.

7. The functionality and behavior of the functions is the same for both driver models. The mapping of IDriver functions to IOM driver functions are as follows:

IDriver	IOM driver
XXX_Instance_init	mdBindDev
XXX_Instance_finalize	mdUnbindDev
XXX_open	mdCreateChan
XXX_close	mdDeleteChan
XXX_control	mdControlChan
XXX_submit	mdSubmitChan

8. All the references to module wide config parameters in the IDriver model map to macro definitions and preprocessor directives (#define and #ifdef etc) for the IOM drivers. e.g.

   a. XXX_edmaEnable in IDriver maps to –D XXX_EDMA_ENABLE for IOM driver.

   b. XXX_FIFO_SIZE in IDriver maps to #define FIFO_SIZE in IOM driver header file

9. In BIOS 6.x a cfg file is used for configuring the BIOS and driver options whereas in the BIOS 5.x the “tcf” and “tci” files are used for configuring the options.

10. In BIOS 5.x driver support for multiple devices is implemented as follows. A chip specific compiler define is required by the driver source files (-DCHIP_OMAPL138). Based on the include a chip specific header file (e.g soc_OMAPL138) which contains chip specific defines is used by the driver. In order to support a new chip, a new soc_XXX header file is required and driver sources files need to be changed in places where the chip specific define is used.

1.3 Related Documents

1.	TBD	DSP/BIOS Driver Developer’s Guide
3.	SPRUFL9	I2C Specs

1.4 Hardware

The I2C device driver design is in the context of DSP/BIOS running on DSP OMAPL138 core

The I2C module core used here has the following bocks:

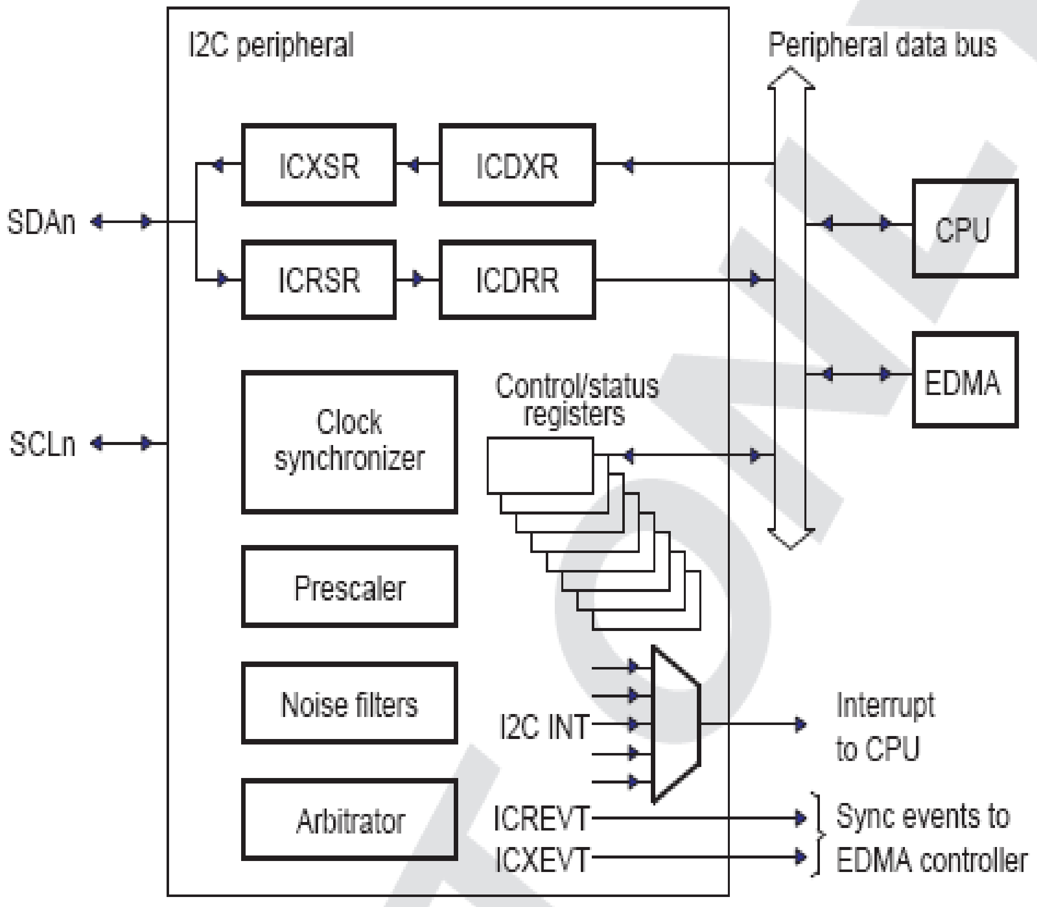


Figure 1 I2C Block Diagram

1.5 Software

The I2C driver discussed here is running DSP/BIOS on the OMAPL138 DSP.

1.5.1 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.

1.5.2 System Architechture

The block diagram below shows the overall system architecture.

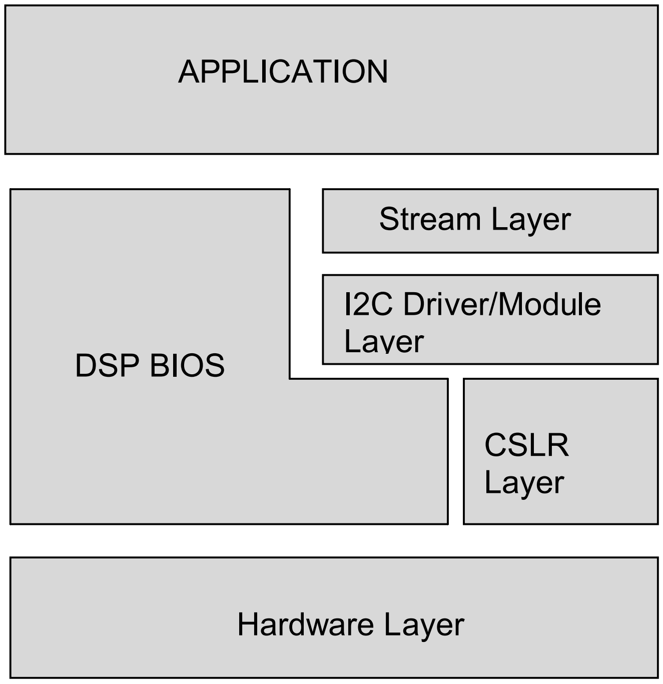


Figure 2 System Architecture

Driver module which this document discusses lies below the steam layer, which is an class driver layer provided by DSP BIOS™ (please note that the stream layer is designed to be OS independent with appropriate abstractions).The I2C driver would use the rCSL (register overlay) to access the Hardware and would use the DSP BIOS™ APIs for OS services. Also as the IDriver is supposed to be a asynchronous driver (to the stream layer), we plan not to use semaphores in IO path. The Application would use the Stream APIs to make use of driver routines

Figure 2 shows the overall device driver architecture. For more information about the IDriver model, see the DSP/BIOS™ documentation. The rest of the document elaborates on the architecture of the Device driver by TI.

1.6 Component Interfaces

In the following subsections, the interfaces implemented by each of the sub-component are specified. The I2C driver module is object of IDriver class one may need to refer the IDriver documentation to access the i2c driver in raw mode or could refer the stream APIs to access the driver through stream abstraction. The structures and config params used would be documented in CDOC format as part of this driver development.

1.6.1 IDriver Interface

The IDriver constitutes the Device Driver Manifest to Stream (and hence to application). This I2C driver is intended to an XDC module and this module would inherit the IDriver interfaces. Thus the I2C driver module becomes an object of IDriver class. Please note that the terms “Module” and “Driver or IDriver” would be used in this document interchangeably.

As per xdc specification, an module should feature a xdc file, xs file and souce file as a minimum.

I2C module specification file (I2C.xdc)

The XDC file defines the following in its public section: the data structures, enums, constants, IOCTLS, error codes and module wide config variables that shall be exposed for the user.

These definitions would include

ENUMS: Operation modes, Serial port settings (Baud, num bits etc), Fifo trigger level

STRUCTURES: Communication status (no. of bytes transferred, errors etc.), Chanparms,

CONSTANTS: error ids and ioctls

Also this files specifies the list of configurable items which could be configured/specified by the application during instantiation (instance parameters)

In its private section it would contain the the data structures, enums, constants and module wide config variables. The Instance object (the driver object) and channel objects which contain all the info related to that particular IO channel. This information might be irrelevant to the User. The instance object is the container for all driver variables, channel objects etc. In essence, it contains the present state of the instance being used by the application

The XDC framework translates this into the driver header file (I2c.h) and this header file shall be included by the applications, for referring to any of the driver data structures/components. Hence, XDC file contains everything that should be exposed to the application and also accessed by the driver.

Please note that by nature of the specification of the xdc file, all the variables (independent or part of structure) need to be initialized in xdc file itself.

I2C module script file (I2c.xs)

The script file is the place where static instantiations and references to module usage are handled. This script file is invoked when the application compiles and refers to the I2C module/driver. The I2c.xs file contains two parts

1. Handling the module use references

   When the module use is called in the application configuration for the I2c module, the module use function in the I2c.xs file is used to initialize the hardware instance specific details like base addresses, interrupt numbers, frequency etc. This data is stored for further use during instantiation. This gives the flexibility to design, to handle multiple SOC with single c-code base (as long as the IP does not deviate).

2. Handling static instantiation of the I2C instance

   When the I2C instance is instantiated statically in the application configuration file, (please note that dynamic instantiation is also possible from a Cfile) the instance static init function is called. If a particular instance is configured with set of instance parameters (fron CFG file) they are used here to configure the instance state. The instance state is populated based on the instance number, like the default state of the driver, channel objects etc

The I2C IDriver module implements the following interfaces

Serial Num	Interfaces	Description
1	I2c_Instance_init()	Handle dynamic calls to module instantiation. This shall be a duplication of the tasks done in instance static init in the module script file.
2	I2c_Instance_finalize()	Unregister interrupt, reset hardware and driver state and all deinitialization foes here. Effectively removes the usage of I2C instance.
3	I2c_open ()	Creates a communication channel in specified mode to communicate data between the application and the I2C module instance.
4	I2c_close ()	Frees a channel and all its associated resources.
5	I2c_control ()	Implements the IOCTLS for I2C IDriver module. All control operations go through this interface.
6	I2c_submit ()	Submit an I/O packet to a channel for processing. Used for data transfer operations

1.6.2 CSLR Interface

The CSL register interface (CSLR) provides register level implementations. CSLR is used by the I2C IDriver module to configure I2C registers. CSLR is implemented as a header file that has CSLR macros and register overlay structure.

1.7 Design Philosophy

This device driver is written in conformance to the DSP/BIOS IDriver model and handles communication to and from the I2C hardware.

1.7.1 The Module and Instance Concept

The IDriver model, conforming to the XDC framework, provides the concept of the *module and instance* for the realization of the device and its communication path as a part of the driver implementation. The module word usage (I2C module) refers to the driver as one entity. Any detail, configuration parameter or setting which shall apply across the driver shall be a module variable. For example, mode of operation (interrupt/polled/EDMA) setting is a module wide variable. Default parameters for any channel opened shall be a module wide variable since, it should be available during module start up or use call, to initialize all the channels to their default values.

This instance word usage (I2C instance 1) refers to every instantiation of module due to a static or dynamic create call. Each instance shall represent the device directly by holding info like the TX/RX channel handles, device configuration settings, hardware configuration etc. Each hardware instance shall map to one I2C instance. This is represented by the Instance_State in the I2C module configuration file.

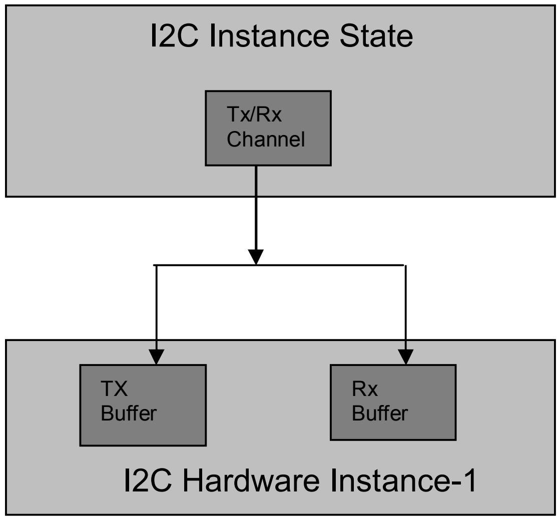


Figure 3 Instance Mapping

Hence every module shall only support the as many number of instantiations as the number of I2C hardware instances on the SOC

1.7.2 Design Constraints

I2C IDriver module imposes the following constraint(s).

- I2C driver shall not support dynamically changing modes between Interrupt, Polled and DMA modes of operation.
- Only one channel for slave operation is possible

2 I2C Driver Software Architecture

This section details the data structures used in the I2C IDriver module and the interface it presents to the Stream layer. A diagrammatic representation of the IDriver module 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 instance, and channel concept in mind.

This driver uses an internal data structure, called channel, to maintain its state during execution. This channel is created whenever the application calls a Stream create call to the I2C IDriver module. The channel object is held inside the Instance State of the module. However, this instance state is translated to the I2c_Object structure by the XDC frame work. The data structures used to maintain the state are explained in greater detail in the following *Data Structures* sub-section. The following figure shows the static view of I2C driver.

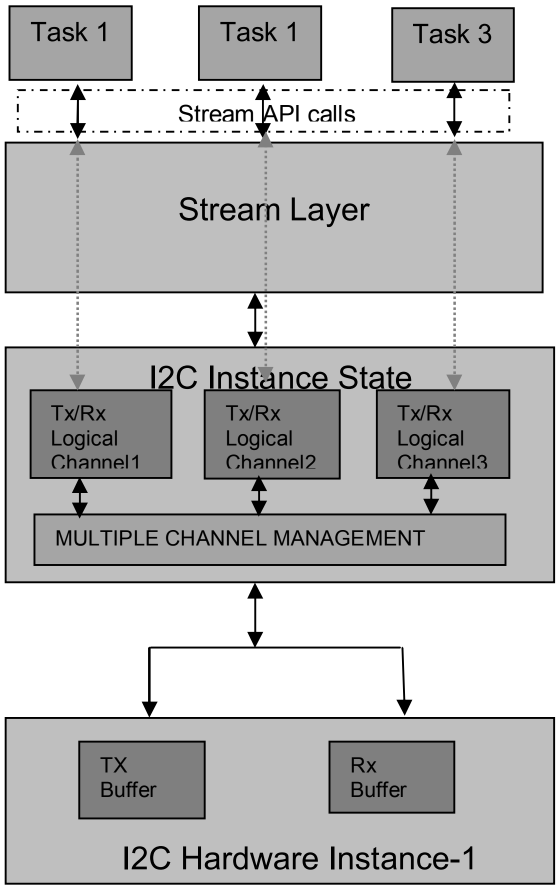


Figure 4 I2C driver static view

2.1.2 Data Structures

The IDriver employs the Instance State (I2c_Object) and Channel Object structures to maintain state of the instance 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 module start up or initialization. The channel params structure is used to specify required characteristics while creating a channel. For current implementation channel parameters only contain the EDMA driver handle, when in EDMA mode of operation.

The following sections provide major data structures maintained by IDriver module and the instance.

2.1.2.1 The Instance Object (I2c_Object)

The instance state comprises of all data structures and variables that logically represent the actual hardware instance on the hardware. It preserves the input and output channels for transmit and receive, parameters for the instance etc. The handle to this is sent out to the application for access when the module instantiation is done via create statically (application CFG file) or dynamically (C file during run time). The parameters that are to be passed for this call is described in the section Device Parameters

S.No	Structure Elements (I2c_*Object)*	Description
1	*instNum*	Instance number for instance handle reference
2	*devStatus*	Status of the I2c handle
3	opMode	Driver operating mode polled, interrupt, dma etc
4	chans[]	channels for the I2C
5	ownAddr	This specifies I2c devices own address (7 or 10 bit)
6	numOpens	Number of active opens
7	hwEventCallback	callback when accessed as slave
8	appSlvHandle	App handle which will be provided while registering callback and will be returned with slave callback
9	hEdma	Flag to indicate if char length greater than 8 bits
10	dmaChaAllocated	Flag to inidicate EDMA channels allocation status
11	edma3EventQueue	Edma event Q to be requested, while requesting an EDMA3 channel
12	hwiNumber	Hardware interrupt Number
13	enablecache	Submitted buffers are in cacheable memory

14	deviceInfo	device instance information
15	polledModeTimeout	Timeout for polled mode of operation
16	currentActiveChannel	Pointer to the channel whose request is in progress
17	isSlaveChannelOpened	Boolean to indicate if the slave channel for this instance is already opened

2.1.2.2 The Channel Object

The interaction between the application and the device is through the instance object and the channel object. While the instance object represents the actual hardware instance, the channel object represents the logical connection of the application with the driver (and hence the device). It is the channel which represents the characteristic/types of connection the application establishes with the driver/device and hence determines the data transfer capabilities the user gets to do to/from the device. For example, the channel could be input/output channel. This capability provided to the user/application, per channel, is determined by the capabilities of the underlying device. The I2C device is a half duplex device and can thus either receive or transmit at a given instant. Also, a give channel at any instant can transmit and receive. Thus, per instance we have one channel for transmit and receive. The number of channels an instance is permitted to support is a policy decision based on system resources available, like the memory, load on the device etc.

S.No	Structure Elements (I2c*_ChanObj)*	Description
1	*mode*	Channel mode of operation: Input or Output
2	*cbFxn*	In case the driver is in any interrupt mode of operation and the application needs to be notified of any completion this callback register by the application is called

3	*cbArg*	In case the driver is in any interrupt mode of operation and the application needs to be notified of any completion this callback register by the application is called
4	*instHandle*	I2c Handle to access the i2c channel params
6	busFreq	I2C Bus Frequency
7	numBits	Number of bits/byte to be sent/received
8	addressing	This is used to choose 7bit/10bit Addressing mode
9	loopbackEnabled	This is used to enable/disable Digital Loob Back (DLB) mode for I2c driver
11	pendingState	Shows whether io is in pending state or not
12	cancelPendingIO	Shows whether IO has to cancel or not
15	currError	This variable is used to store the Error value
16	currFlags	Current Flags for read/write
17	currBuffer	User buffer for read/write
18	currBufferLen	User buffer length
19	OwnAddress	Own address of this instance
20	activeIOP	The IOP being processed
21	dataParam	Parameters for current data transfer
22	queuePendingList	List of pending IOPs
23	tempICMDRValue	Variable to store the mode control register settings across the channel while instance is being readied for

		data transfer
24	taskPriority	Priority of the task owning this channel
25	channelState	Current state of this channel
26	masterOrSlave	If this channel is operating the instance as a master or a slave mode

2.1.2.2.1 Selection of next IO request

Since, there are multiple channels and channels can be created in tasks of different priority (note that a single channel cannot be shared between tasks), the driver should take care of scheduling the transfers. The request from the highest priority task should get a chance to be serviced next.

Hence,

- For each channel in the list
    - If the channel is opened
        - If the priority task of the task owning this channel is greater than the previous channel
            - If the channel has pending request
                - Set this channel as the next channel

2.1.2.3 The Device Parameters

During module instantiation a set of parameters are required which shall be used to configure the hardware and the driver for that operation mode. This is passed via create call for the module instance. Please note that there are two ways to create an instance (static-using CFG file and dynamic using application c file) and each of these methods have different way of passing devparams. For further information of these methods please refer xdc documentation. These parameters are preserved in the DevParams structure and are explained below:

S.No	Structure Elements	Description
1	*instNum*	Instance number for instance handle reference

2	mode	Driver operating mode polled, interrupt, dma etc
3	ownAddr	This specifies I2c devices own address (7 or 10 bit)
4	numBits	Number of bits/byte to be sent/received
5	busFreq	I2C Bus Frequency
6	addressing	This is used to choose 7bit/10bit Addressing mode
7	dlb	This is used to enable/disable Digital Loob Back (DLB) mode for I2c driver
8	edma3EventQueue	Edma event Q to be requested, while requesting an EDMA3 channel
9	hwiNumber	Hardware interrupt Number

2.1.2.4 The Channel Parameters

Every channel opened to the device may need some setting by the user. These parameters may be passed through to the driver module by chanParams.Currently the I2C module requires the handle to the EDMA driver when operating in the DMA interrupt mode and the option of communication mode as a slave or master

The parameters are explained below:

S.No	Structure Elements	Description
1	hEdma	EDMA handle. Used in DMA opmode
2	masterOrSlave	Communication mode as slave or master

2.2 Dynamic View

2.2.1 The Execution Threads

The I2C IDriver module involves following execution threads:

Application thread: Creation of channel, Control of channel, deletion of channel and processing of I2C data will be under application thread.

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

Edma call back context: The callback from EDMA LLD driver (in case of EDMA mode of operation) on the completion of the EDMA IO programmed, (this would actually be in the CPU interrupt context)

2.2.2 Input / Output using I2C driver

In I2C, the application can perform IO operation using Stream_read/write() calls (corresponding IDriver function is I2c_submit()). The handle to the channel, buffer for data transfer, sizeof data transfer,

The I2c channel transfer is enabled upon submission of the IO request.

2.2.3 Functional Decomposition

The I2C driver, seen in the RTSC framework, has two methods of instantiation, or instance creation – Static and Dynamic. By static instantiation we mean, the invocation of the create call for the module in the configuration file of the application. This is called so because, the creation of the instance is at build time of the application. By dynamic instantiation we mean, the invocation of the create call for the module during runtime of the application.

The two types of instantiation of the module are handled in different ways in the module. The static instantiation of the module is handled in the module script file, and the dynamic instantiation is handled in the C file.

This design concept explained in the sections to follow.

2.2.3.1 *module$use() of XS file*

One important feature in the RTSC framework design of this package is that we have designed to have a soc.xs capsule file, instead of a soc.h file, which will have the SoC specific information, like interrupt/event numbers, base addresses, CPU/module frequency values etc. This move is adopted to keep the driver C code free from compiler switches and everything of this sort in the form of either configuration variables for the module or loading the platform specific data from the soc.xs capsule. This loading of data is done in the module use function.

The module shall have an array of device instance configuration structures (default DeviceInfo), which shall contain, base address, event number, frequency and such instance specific details. The length of this is defined by the array member of this instance in the soc.xs file.

The default device information is populated for all the instance numbers in this function since this information is needed to later prepare the instances in instance static init and the instance init functions.

2.2.3.2 *Instance$static$init()*

This function context is the where the instance statically created is initialized. Please note that the instance params provided by the applications (from the CFG file) would override the default value of those parameters from XDC file.

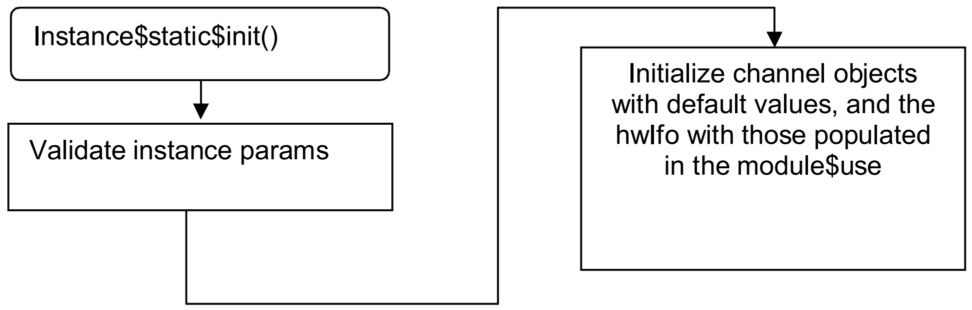

Instance$static$init()
Validate instance params
Initialize channel objects with default values, and the hwIfo with those populated in the module$use

Figure 5 instance$static$init() flow diagram

2.2.3.3 I2c_Instance_init()

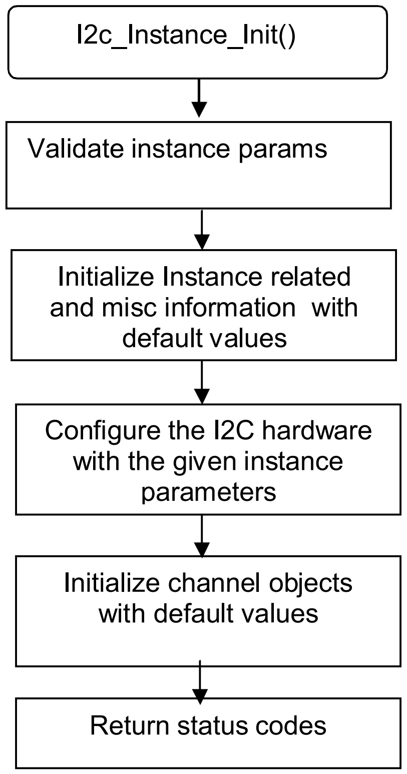

I2c_Instance_Init()
Validate instance params
Initialize Instance related and misc information with default values
Configure the I2C hardware with the given instance parameters
Initialize channel objects with default values
Return status codes

Figure 6 I2c_Instance_init() flow diagram

The instance init function is called when the module is dynamically instantiated. This is the only context available for initialization per instance, when doing dynamic (application C file during run time) instantiation. Hence, this function should be including all the initialization done in the instance static init in the module script file and the module startup function. The return, value of this function represents the extent to which the instance (and hence its resources) were initialized. For example, we could use return value of 0 for complete (successful) initialization done, return value of 1 for failure at the stage of a resource allocation and so on. This return value is preserved by the RTSC/BIOS framework and passed to the Instance_finalize function, which does a clean up of the driver during instance removal accordingly.

2.2.3.4 I2c_Instance_finalize()

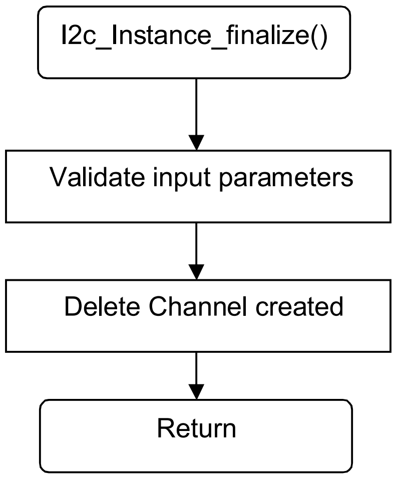


Figure 7 I2c_Instance_finalize() flow diagram

The I2c_Instance_init and I2c_Instance_finalize functions are called when by the XDC/BIOS framework.

The instance init function does a start up initialization for the driver. This function is called when the module is instantiated dynamically by the I2c_create call by the application. At this module instantiation, the instance related information given by the application and instance related miscellaneous information should be done here which form a pre-requisite before the actual functioning of the driver (viz channel creation and then data transfers).

The instance finalize function does a final clean up before the driver could be relinquished of any use. Here, all the resources which were allocated during instance initialization shall be unallocated. After this the instance no more is valid and needs to be reinitialized. Please note that the input parameter for this function is the initialization status returned from the instance)init function. This helps in de-allocation of resources only that were actually allocated during instance_init.

2.2.3.5 *I2c_open()*

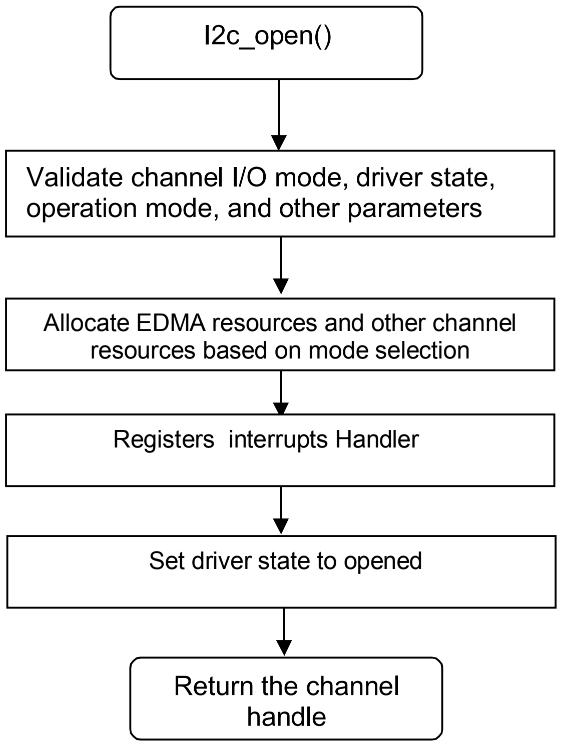

I2c_open()
Validate channel I/O mode, driver state, operation mode, and other parameters
Allocate EDMA resources and other channel resources based on mode selection
Registers interrupts Handler
Set driver state to opened
Return the channel handle

Figure 8 I2c_open () flow diagram

2.2.3.6 *I2c_close ()*

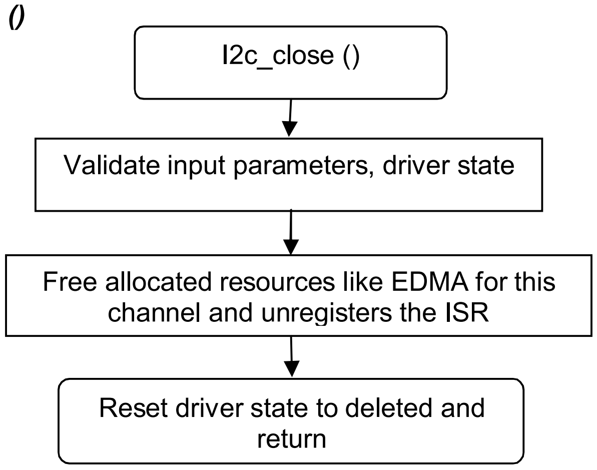

I2c_close ()
Validate input parameters, driver state
Free allocated resources like EDMA for this channel and unregisters the ISR
Reset driver state to deleted and return

Figure 9 I2c_close () flow diagram

2.2.3.7 *I2c_control ()*

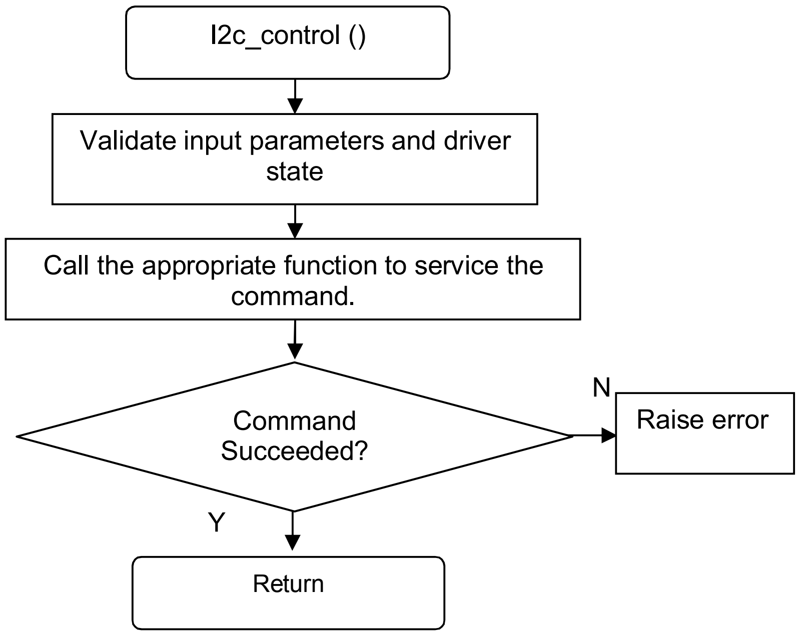
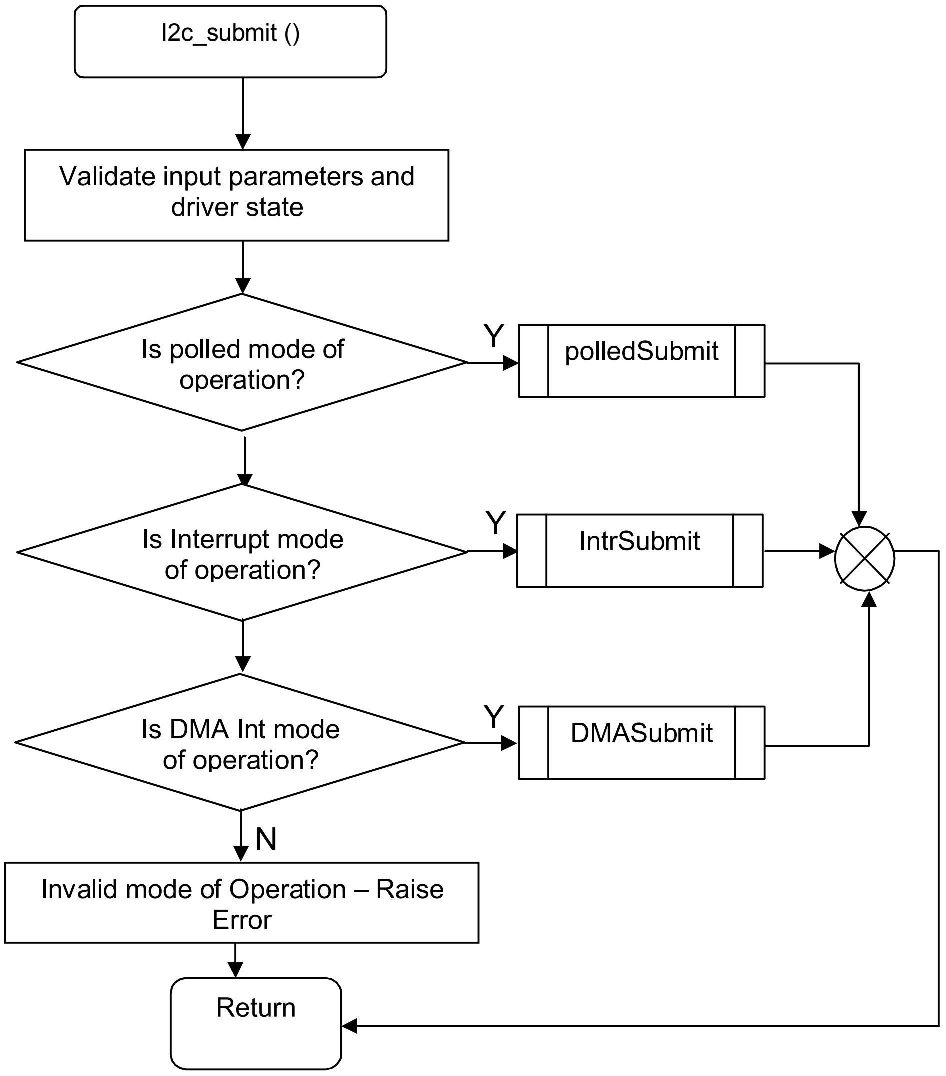

I2c_control ()
Validate input parameters and driver state
Call the appropriate function to service the command.
Command Succeeded?
N
Raise error
Y
Return

Figure 10 I2c_control () flow diagram

2.2.3.8 I2c_submit ()


I2c_submit ()
Validate input parameters and driver state
Is polled mode of operation?
Y
polledSubmit
Is Interrupt mode of operation?
Y
IntrSubmit
Is DMA Int mode of operation?
Y
DMASubmit
N
Invalid mode of Operation – Raise Error
Return

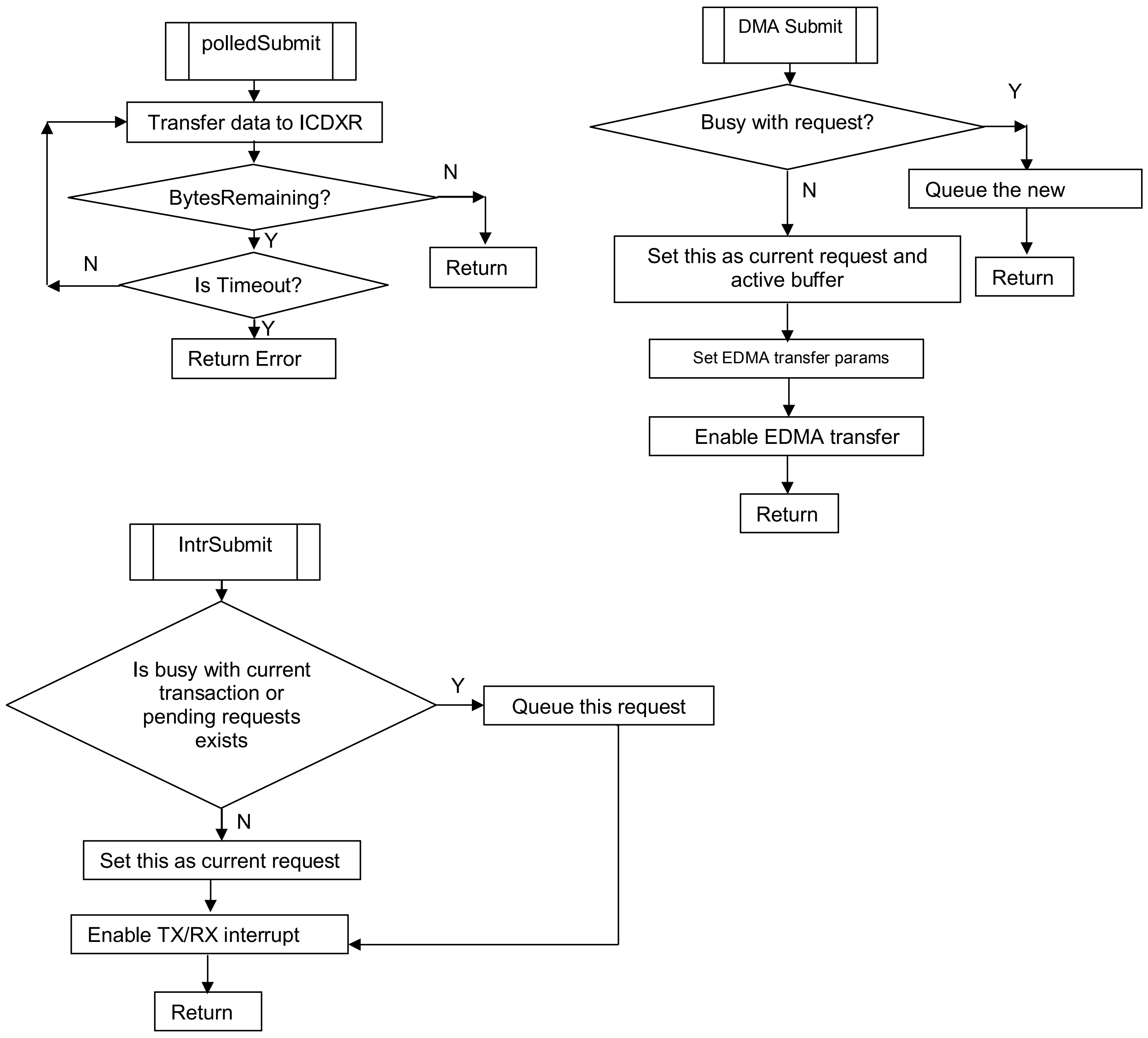
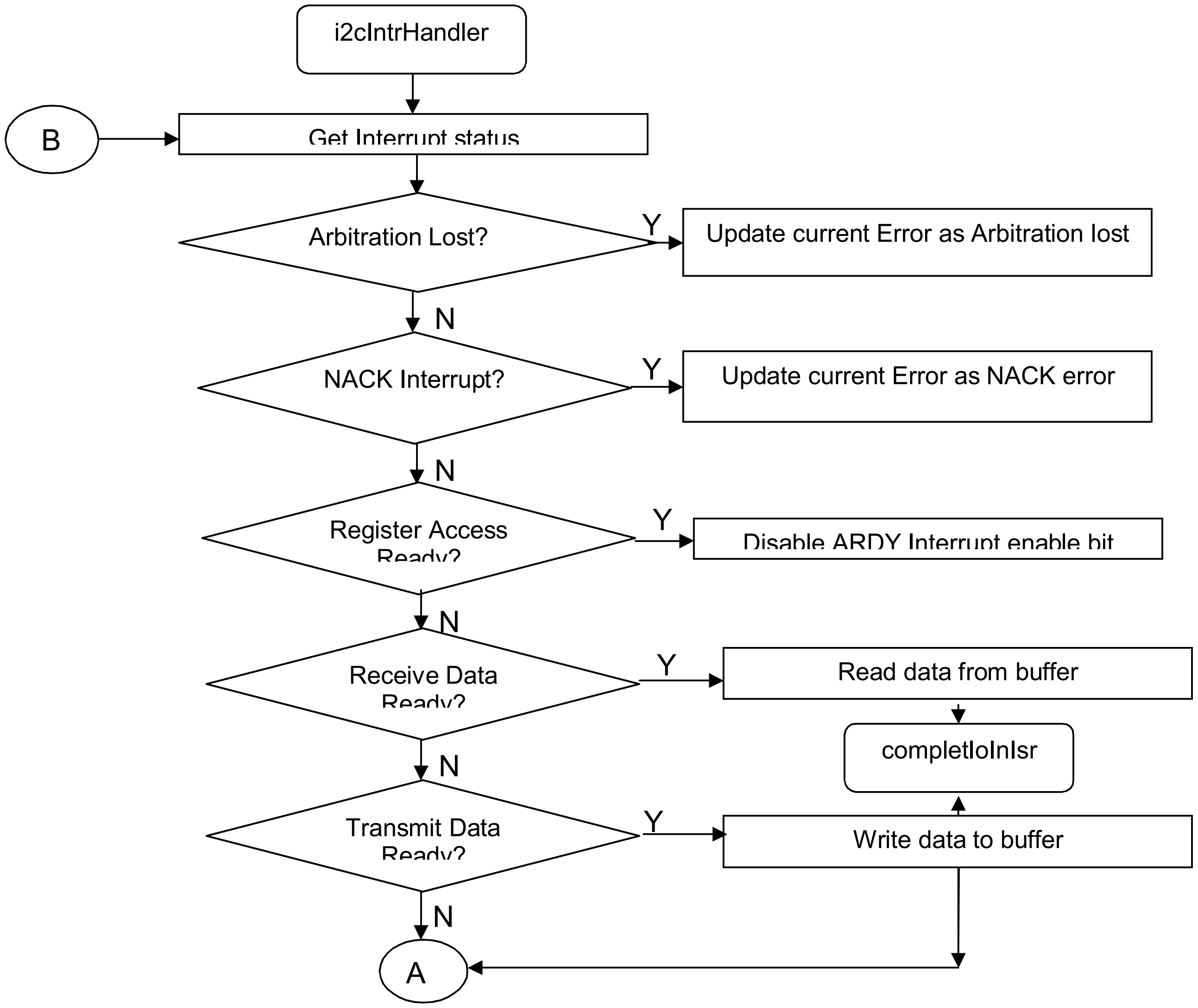


Figure 11 I2c_submit ()

The driver by inheritance of the IDriver module is an asynchronous driver. The driver shall not anytime pend for status of the current request. It shall queue the requests, if they are already any in progress and shall return IOM_PENDING status to the stream layer. The stream takes care of the pending status. The interrupt handler or the EDMA callback functions call the application callback (essentially the Stream layer callback here) is then called to indicate completion. One exception is the polled mode of operation. Here the caveat is that unlike in the interrupt mode or the EDMA mode of operation, there is no other context where in the queued packet can be processed and then status posted to the Stream. Hence, in the polled mode the status is always returned immediately as completed or error (timeout). Also, when multiple channels are present it needs to be taken care, which channel gets the chance to transfer. This logic is shown in section "Selection of next channel" while explaining the channel object

2.2.3.9 i2cIntrHandler



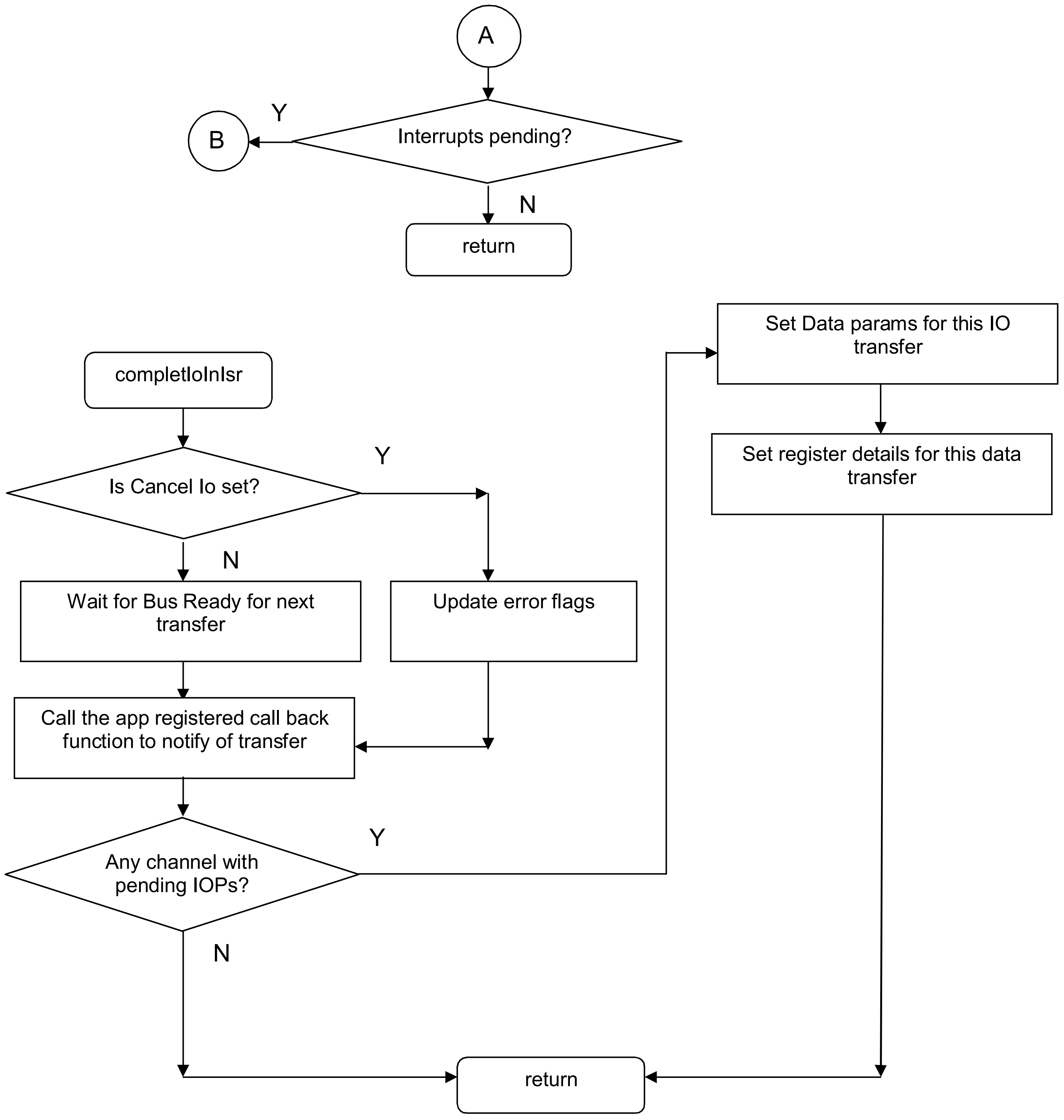

A
Y
B
Interrupts pending?
N
return
Set Data params for this IO transfer
completIoInIsr
Set register details for this data transfer
Y
Is Cancel Io set?
N
Wait for Bus Ready for next transfer
Update error flags
Call the app registered call back function to notify of transfer
Y
Any channel with pending IOPs?
N
return

Figure 12 i2cIntrHandler

2.2.3.10 *i2cRx/TxCallback*

When I2c is operating in the DMA interrupt mode for data transfer, the I2c driver during the channel open requests EDMA channels and registers a callback (i2cRx/TxCallback())for notification of the transfer completion status. Before starting any transfer, I2c driver sets the parameters (like the source and destination addresses, option fields etc) for the transfer in the EDMA parameter RAM sets allocated. It then enables the transfer in triggered event mode. When the EDMA driver completes the transfer, it calls the registered callback during open to notify about the status. It is now up to the I2c driver to take act upon the status. This is done in the i2cTx/RxCallback() as shown in the figure below.

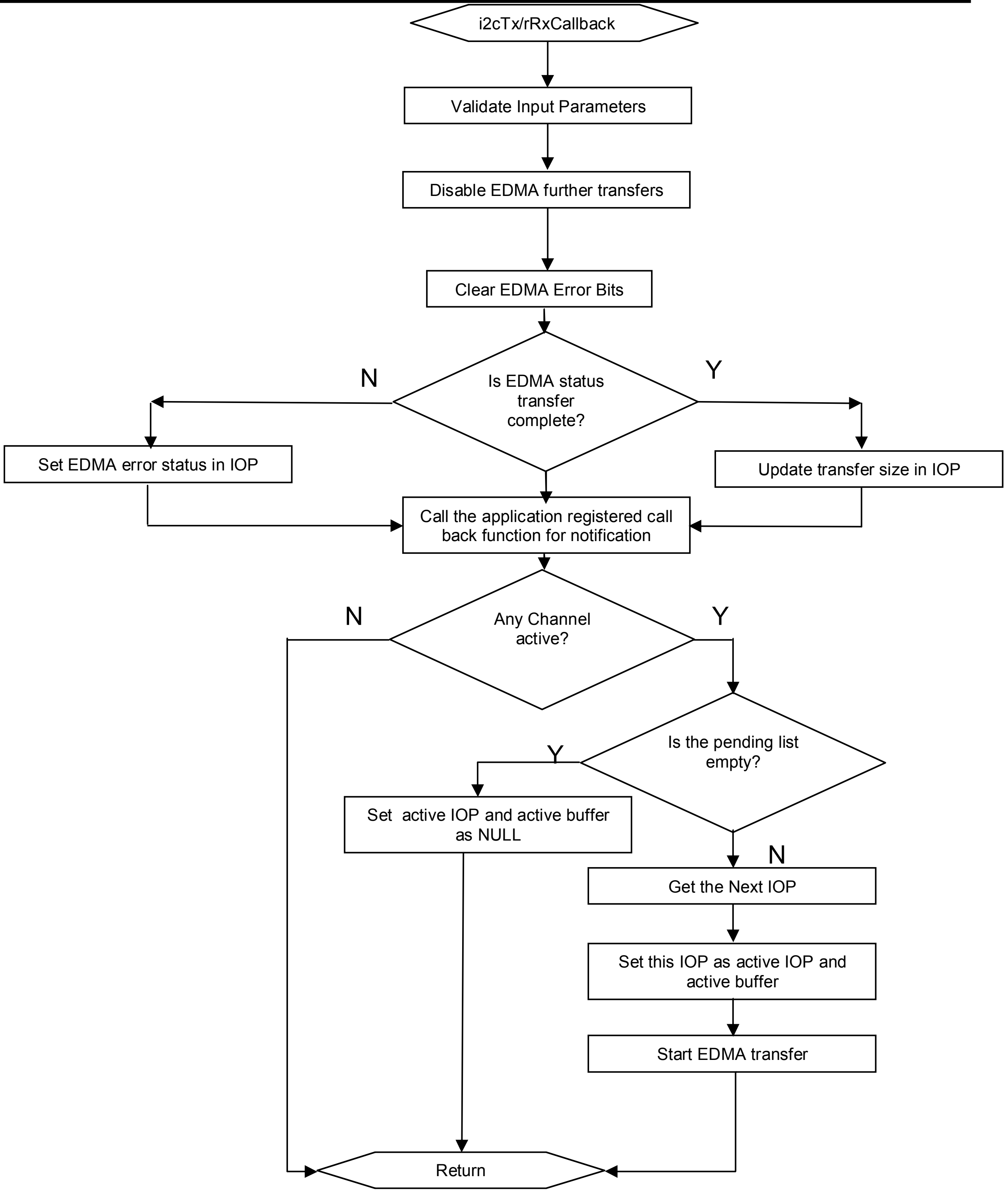

i2cTx/rRxCallback
Validate Input Parameters
Disable EDMA further transfers
Clear EDMA Error Bits
Is EDMA status transfer complete?
N
Y
Set EDMA error status in IOP
Update transfer size in IOP
Call the application registered call back function for notification
Any Channel active?
N
Y
Is the pending list empty?
Y
Set active IOP and active buffer as NULL
N
Get the Next IOP
Set this IOP as active IOP and active buffer
Start EDMA transfer
Return

Figure 13 i2cTx/RxCallback

2.3 Slave mode of operation

The I2c instance to operate in slave mode should be configured with specific details in the register as mentioned in the peripheral user guide and some of the details the driver shall configure in the master mode shall not be valid for the slave mode of operation. For example, in the slave mode the clock settings like prescalar are not valid since, the clock is received by this slave generated from the master..

The option of slave mode (or master mode) of operation, should be supplied along with the channel parameters structure (masterOrSlave field) during Stream_create. The choice mode of operation as slave/master can be per channel.

Note:

For polled mode of operation the driver does not implement the task sleeping in in between checks for data ready status, during data transfer. This is because, while in sleep the data may arrive and the data may go unread. This can be more prevalent with increasing data clock frequencies. This non use of task sleep result in a tight while loop for checking data ready status during transfers and may block out other tasks in the system from executing, for the timeout duration set by the user. Hence, it is advised that in slave mode interrupt mode of operation may be used.

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	IOCTL_SET_BIT_RATE	IOCTL for setting the i2c bit rate
2	IOCTL_GET_BIT_RATE	IOCTL for getting the current bit rate of the I2c bus
3	IOCTL_CANCEL_PENDING_IO	IOCTL for cancelling the current pending IO
4	IOCTL_BIT_COUNT	IOCTL for cancelling the current pending IO
5	IOCTL_NACK	IOCTL for setting the I2c NACK
6	IOCTL_SET_OWN_ADDR	IOCTL for setting the I2c device own address
7	IOCTL_GET_OWN_ADDR	IOCTL for getting the I2c device own address
8	IOCTL_SET_POLLEDMODETIMEOUT	IOCTL for setting the I2c device polled mode timeout value.