

DSP/BIOS SPI Device Driver

Architecture/Design Document

Revision History

Document Version	Author(s)	Date	Comments
0.1	Chandan Kr Nath	September 10 2008	Created the document
0.2	Madhvapathi Sriram	October 15, 2008	Revised document for release 2.00.00.03
0.3	Chandan Kr Nath	December 9, 2008	Revised document for release 2.00.00.04
0.4	Imtiaz SMA	January 21, 2009	Updated the document for the IOM driver and IDriver contrast

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

Copyright ©. 2009, Texas Instruments Incorporated

Table of Contents

1	System Context.....	6
1.1	Terms and Abbreviations.....	6
1.1	Disclaimer.....	6
1.2	IOM driver Vs IDriver.....	6
1.3	Related Documents.....	8
1.4	Hardware	9
1.5	Software.....	10
1.5.1	Operating Environment and dependencies.....	10
1.5.2	System Architecture.....	10
1.6	Component Interfaces.....	11
1.6.1	IDriver Interface.....	11
1.6.2	CSLR Interface.....	13
1.7	Design Philosophy.....	13
1.7.1	The Module and Instance Concept.....	13
1.7.2	Design Constraints	14
2	SPI Driver Software Architecture.....	15
2.1	Static View	15
2.1.1	Functional Decomposition.....	15
2.1.2	Data Structures.....	16
2.2	Dynamic View.....	22
2.2.1	The Execution Threads.....	22
2.2.2	Input / Output using SPI driver	22
2.2.3	Functional Decomposition.....	23
2.3	Slave mode of operation.....	36
3	APPENDIX A – IOCTL commands	37

List Of Figures

Figure 1 SPI Block Diagram	9
Figure 2 System Architecture	10
Figure 3 Instance Mapping	14
Figure 4 SPI driver static view	16
Figure 5 instance\$static\$init() flow diagram	23
Figure 6 Spi_Instance_init () flow diagram	24
Figure 7 Spi_Instance_finalize () flow diagram.....	26
Figure 8 Spi_open () flow diagram.....	27
Figure 9 Spi_close () flow diagram	28
Figure 10 Spi_control () flow diagram	29
Figure 11 Spi_submit flow diagram()	31
Figure 12 spiIntrHandler ()......	33
Figure 13 Spi_localCallbankTransmit/Receive()	35

1 System Context

The purpose of this document is to explain the device driver design for SPI peripheral using DSP/BIOS operating system running on DSP OMAPL138. This driver is aimed at providing support for multiple SPI instances i.e. it can be used with other SPI supported SoC platforms.

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

1.1 Terms and Abbreviations

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 SPI driver for the DSP/BIOS operating system. Although the current design document explain the SPI 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 SPI 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.

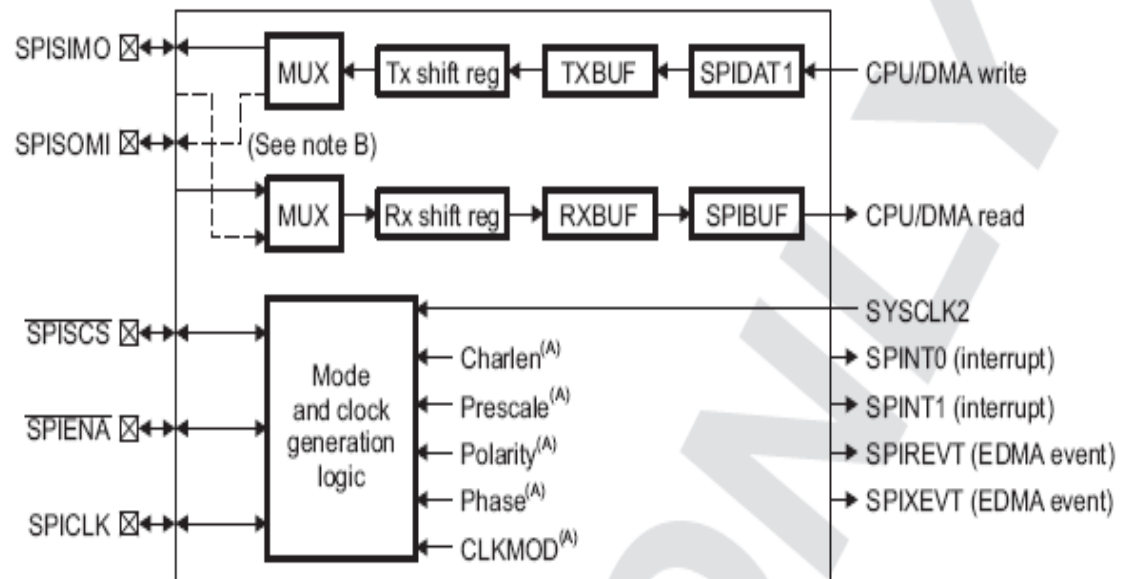
1.3
Related Documents

1.	TBD	DSP/BIOS Driver Developer's Guide
3.	SPRUFM4	SPI User Guide (DRAFT)

1.4 Hardware

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

The SPI module core used here has the following blocks:



A Indicates the log controlled by SPI register bits

B Solid line represents data flow for SPI master mode. Dashed line represents data flow for SPI slave mode.

Figure 1 SPI Block Diagram

The SPI device supports following features.

- SPI device supports full duplex support.
- SPI works in both Master and Slave modes.
- SPI device supports 3, 4 and 5 pin modes of operation.

1.5 Software

The SPI driver discussed here is intended to work with 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 Architecture*

The block diagram below shows the overall system architecture.

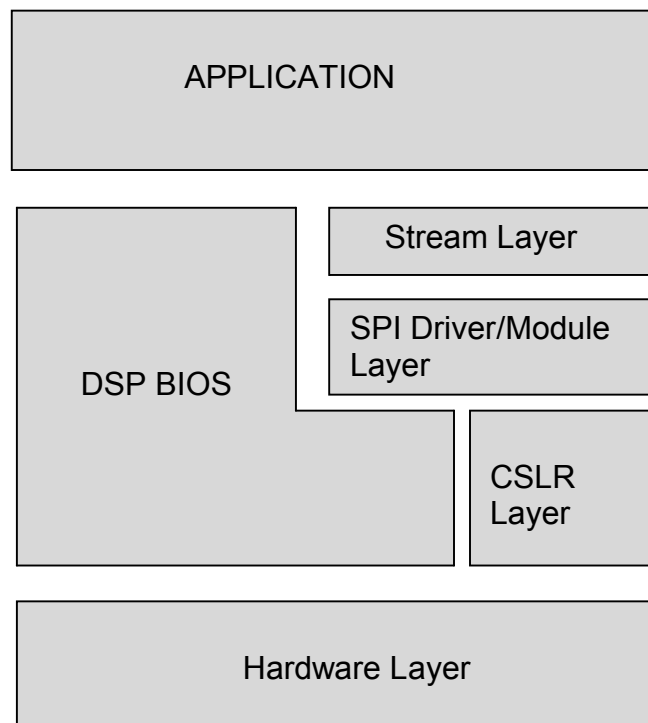


Figure 2 System Architecture

Driver module which this document discusses lies below the stream layer, 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

SPI 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 SPI driver module is object of IDriver class one may need to refer the IDriver documentation to access the spi 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 SPI driver is intended to be an XDC module and this module would inherit the IDriver interfaces. Thus the SPI 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 <module>.xdc file, <module>.xs file and source file as a minimum.

SPI module specification file (Spi.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.), Channel Parameters.

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 (Spi.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.

SPI module script file (Spi.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 SPI module/driver. The Spi.xs file contains two parts

1. Handling the module use references
2. When the module use is called in the application cfg for the Spi module, the module use function in the Spi.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).
3. Handling static instantiation of the SPI instance

When the SPI instance is instantiated statically in the application cfg file, (please note that dynamic instantiation is also possible from a C file) the instance static init function is called. If a particular instance is configured with set of instance parameters (from 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 SPI IDriver module implements the following interfaces

S.No	IOM Interfaces	Description
1	Spi_Module_startup()	Register interrupts, configure hardware and initialization needed before opening a channel. Effectively makes the SPI module ready for use

2	<code>Spi_Instance_init()</code>	Handle dynamic calls to module instantiation. This shall be a duplication of the tasks done in instance static init in the module script file.
3	<code>Spi_Instance_finalize()</code>	Unregister interrupt, reset hardware and driver state and all deinitialization foes here. Effectively removes the usage of SPI instance.
4	<code>Spi_open ()</code>	Creates a communication channel in specified mode to communicate data between the application and the SPI module instance.
5	<code>Spi_close ()</code>	Frees a channel and all its associated resources.
6	<code>Spi_control ()</code>	Implements the IOCTLs for SPI IDriver module. All control operations go through this interface.
7	<code>Spi_submit ()</code>	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 SPI IDriver module to configure SPI 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 SPI hardware. The driver shall support both master and slave mode of operation.

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 (SPI 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/pollled/EDMA) setting is a module wide variable. However, there can also be module wide constants. 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 (SPI 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 SPI instance. This is represented by the Instance_State in the SPI module configuration file.

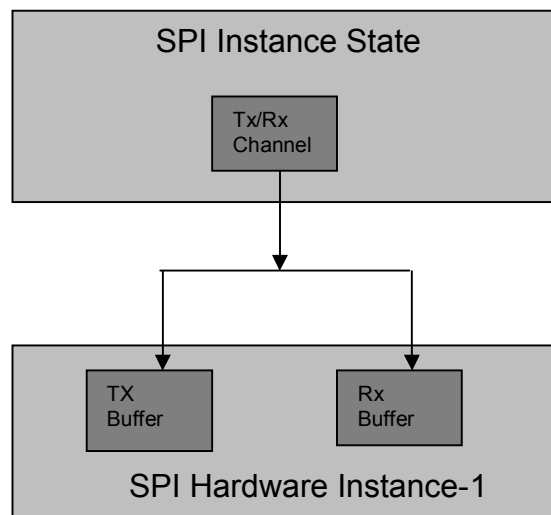


Figure 3 Instance Mapping

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

1.7.2 Design Constraints

SPI IDriver module imposes the following constraint(s).

- SPI driver shall not support dynamically changing modes between Interrupt, Polled and DMA modes of operation.

- There shall be only one channel for slave mode of operation
- An instance can be configured as a slave or master and not a channel
- The data word length can be configured only as 8 bits or 16-bits

2 SPI Driver Software Architecture

This section details the data structures used in the SPI 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 SPI IDriver module. The channel object is held inside the Instance State of the module. However, this instance state is translated to the Spi_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 SPI driver.

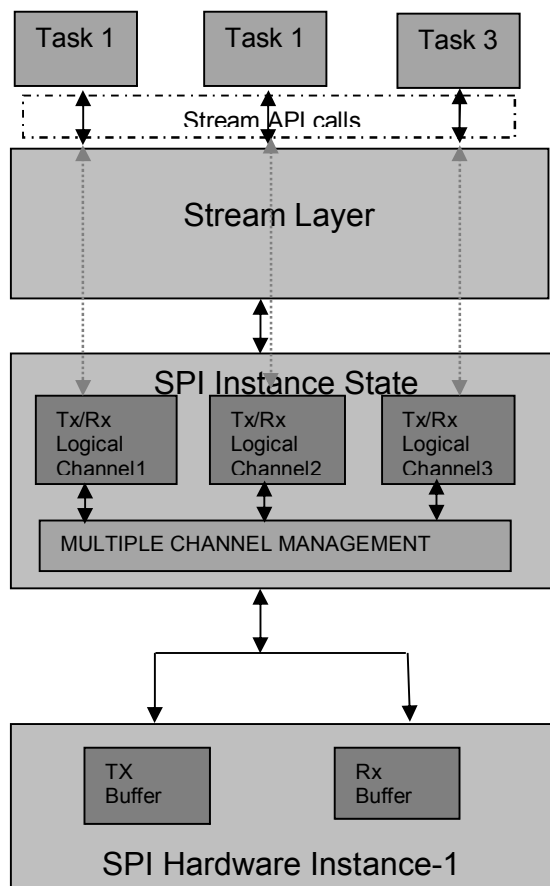


Figure 4 SPI driver static view

2.1.2 Data Structures

The IDriver employs the Instance State (Spi_Object) and Channel Object structures to maintain state of the instance and channel respectively.

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

2.1.2.1 The Instance Object (Spi_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 (Spi_Object)	Description
1.	<i>instNum</i>	Preserve port or instance number of SPI
2.	<i>devState</i>	State of the SPI Either created or deleted
3.	<i>opMode</i>	Mode of operation
4.	<i>deviceInfo</i>	Instance specific information
5.	<i>chanObj[]</i>	Channel objects for the SPI
6.	<i>numOpens</i>	The number of channels opened on this instance
7.	<i>hwiNumber</i>	Hardware interrupt number
8.	<i>spiHWconfig</i>	SPI Hardware configurations
9.	<i>numOpens</i>	Number of channels opened
10.	<i>csHighPolarity</i>	Chip Select Polarity
11.	<i>hEdma</i>	Handle used for Edma
12.	<i>dmaChanAllocated</i>	Flag to indicate EDMA channels allocation status
13.	<i>edmaCbCheck</i>	Use to check occurrence of EDMA callback
14.	<i>polledModeTimeout</i>	Variable to hold Timeout to for the io operation
15.	<i>currentActiveChannel</i>	The channel for which the current IO operation is in

		progress
16.	isSlaveChannelOpened	This boolean track for having only one slave channel

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 SPI device is a full duplex device and can thus receive and transmit at a given instant. Also, a given channel at any instant can transmit and receive. Thus, per instance we have channels, each being able to 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 (Spi_ChanObj)	Description
1.	<i>mode</i>	Channel mode of operation: Input or Output
2.	<i>channelState</i>	state of the SPI Either created or deleted
3.	<i>cbFxn</i>	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.	<i>cbArg</i>	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

5.	<i>instHandle</i>	Spi Handle to access the spi channel params
6.	<i>busFreq</i>	SPI Bus Frequency of operation
7.	<i>loopbackEnabled</i>	Boolean to check if loopback is enabled
8.	<i>charLength16Bits</i>	Boolean to indicate if the data word length is greater than 8 bits
9.	<i>activeIOP</i>	The pointer to the current active packet (being processed)
10.	<i>dataParam</i>	The parameters for the data transfer
11.	<i>pendingState</i>	Shows whether io is in pending state or not
12.	<i>cancelPendingIO</i>	Shows whether IO has to cancel or not
13.	<i>currError</i>	current error flag
14.	<i>currFlags</i>	Current Flags for read/write
15.	<i>transcieveFlags</i>	flag for transcieve operation
16.	<i>currBuffer</i>	Current User buffer for read/write
17.	<i>transBuffer</i>	Buffer used for transieve operation of spi
18.	<i>currBufferLen</i>	User buffer length
19.	<i>queuePendingList</i>	pending lop List head
20.	<i>taskPriority</i>	this will hold the priority of the task that created this channel

2.1.2.2.1 Selection of next IO request to be processed

Since, there are multiple channels and channels can be created in tasks of different priority (note that a single channel handle cannot be shared between tasks), the driver should take care of scheduling the transfers after an IO request is processed. The request that was queued 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

The device parameters structure is used to pass on data to initialize the driver during module start up or initialization.

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 device parameters. These parameters are preserved in the DevParams structure and are explained below:

S.No	Structure Elements	Description
1	<i>instNum</i>	Instance Number
2	<i>opMode</i>	Mode of operation
3	<i>outputClkFreq</i>	OutPut Clock Frequency
4	<i>loopbackEnabled</i>	Enable/Disable loop back mode
5	<i>spiHWCfgData</i>	SPI Hardware Data Configuation
7	<i>hwiNumber</i>	Hardware interrupt number
8	<i>polledModeTimeout</i>	Variable to hold Timeout to for the io operation

2.1.2.4 The Device Hardware Configuration Params

The SPI instance needs some more hardware specific information like default chip select control, interrupt level as to which interrupt line the SPI should use, data word length for transmit and receive, master or slave mode of operation, chip select and transmit active delays etc. These are supplied via the SPI hardware configuration structure explained below.

SPI Hardware Configuration Structure

S.No	Structure Elements spiHWconfig	Description
1	intrLevel	This variable is used to map interrupt line - Intr1 and Intr0
2	masterOrSlave	This variable is used to configure SPI device either in Master or in Slave mode
3	clkInternal	Used to select clock, Internal or External.
4	enableHighZ	whether ENA signal should be tristated when inactive
5	pinOpModes	Spi Operation Modes
6	delay	SPI delay registers value
7	waitDelay	Enable format delay between 2 consecutive transfers
8	csDefault	Default chip select pattern
9	configDatafmt[]	Data Format Configuration values

2.1.2.5 The Channel Parameters

The channel parameters 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.

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. However, currently the SPI module requires only one channel parameter which is the handle to the EDMA driver when operating in the DMA interrupt mode.

The params are explained below:

S.No	Structure Elements	Description
1	hEdma	EDMA handle. Used in DMA opmode

2.2 Dynamic View

2.2.1 The Execution Threads

The SPI IDriver module involves following execution threads:

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

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

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

2.2.3 Functional Decomposition

The SPI 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* of XS file

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.

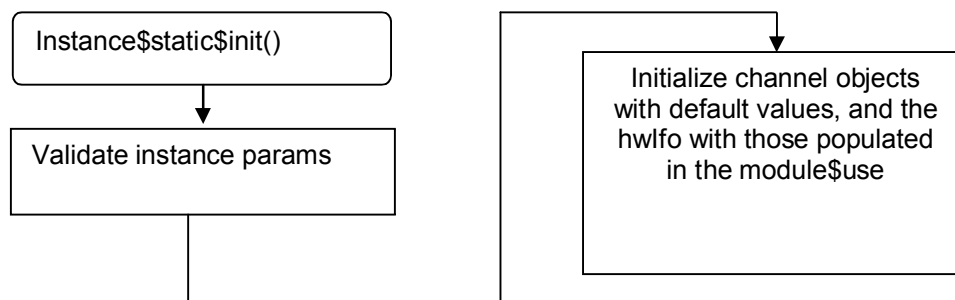


Figure 5 instance\$static\$init() flow diagram

2.2.3.3 *Spi_Instance_init()*

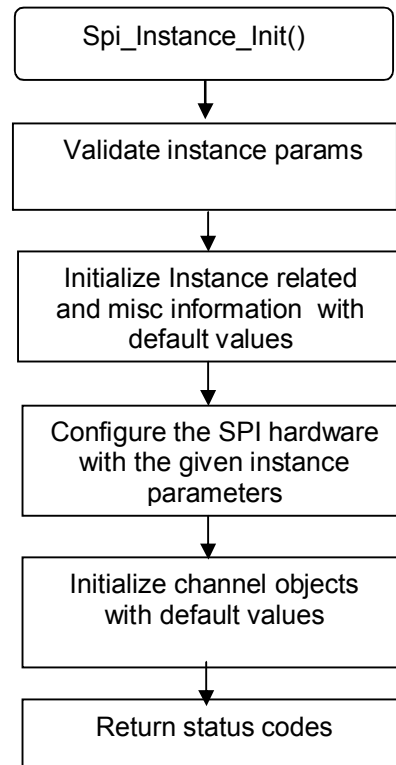


Figure 6 Spi_Instance_init () flow diagram

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 *Spi_Instance_finalize ()*

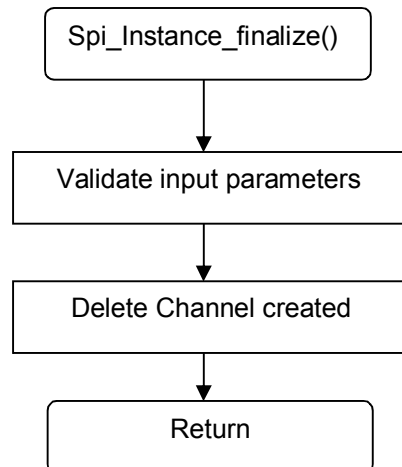


Figure 7 Spi_Instance_finalize () flow diagram

The Spi_Instance_init and Spi_Instance_finalize functions are called 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 Spi_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 *Spi_open*

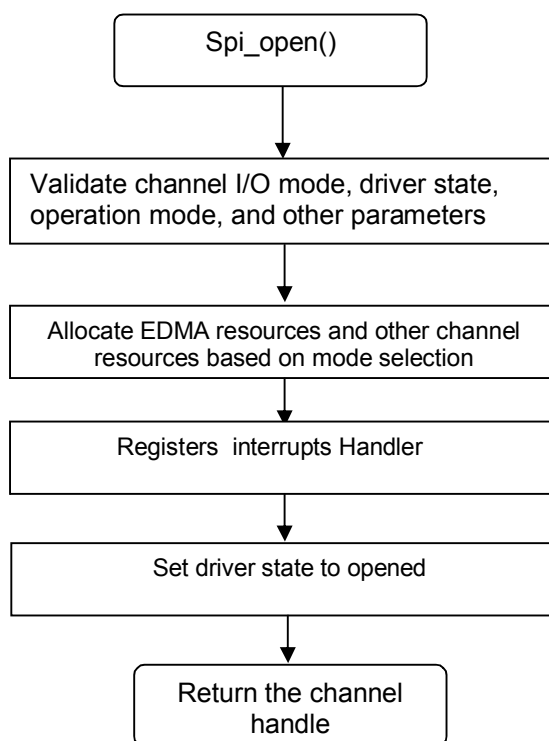
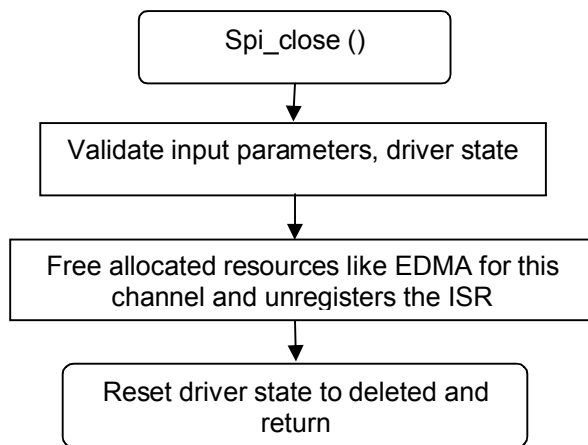
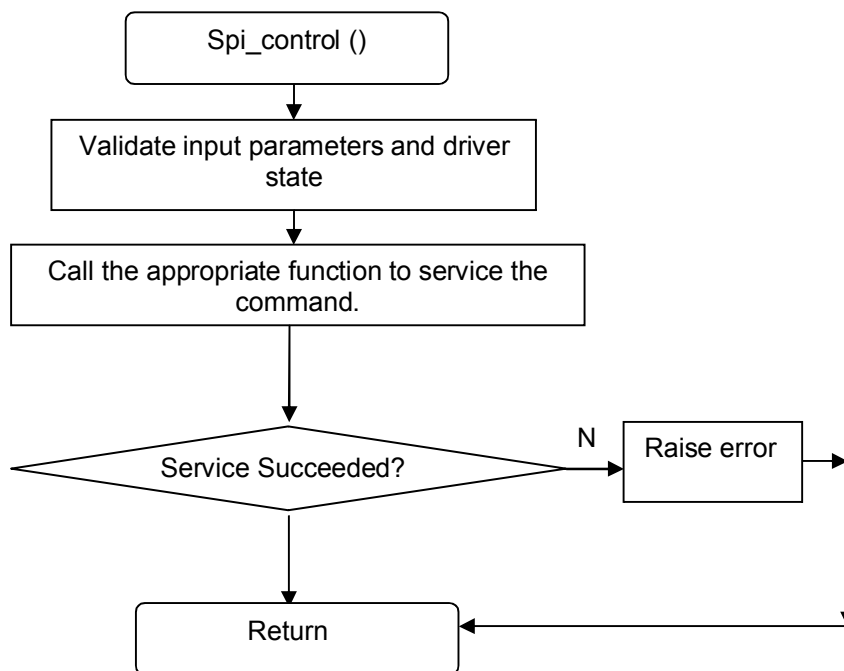
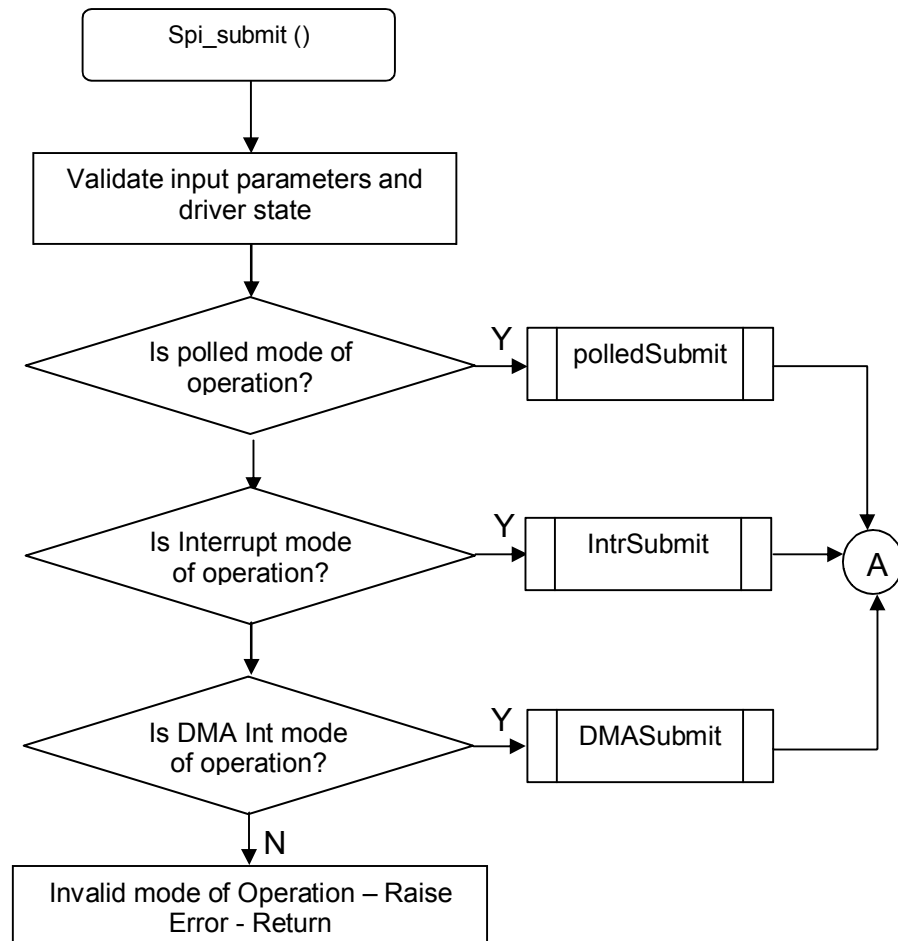


Figure 8 Spi_open () flow diagram

2.2.3.6
Spi_close ()

Figure 9 Spi_close () flow diagram

2.2.3.7
Spi_control

Figure 10 Spi_control () flow diagram

2.2.3.8
Spi_submit


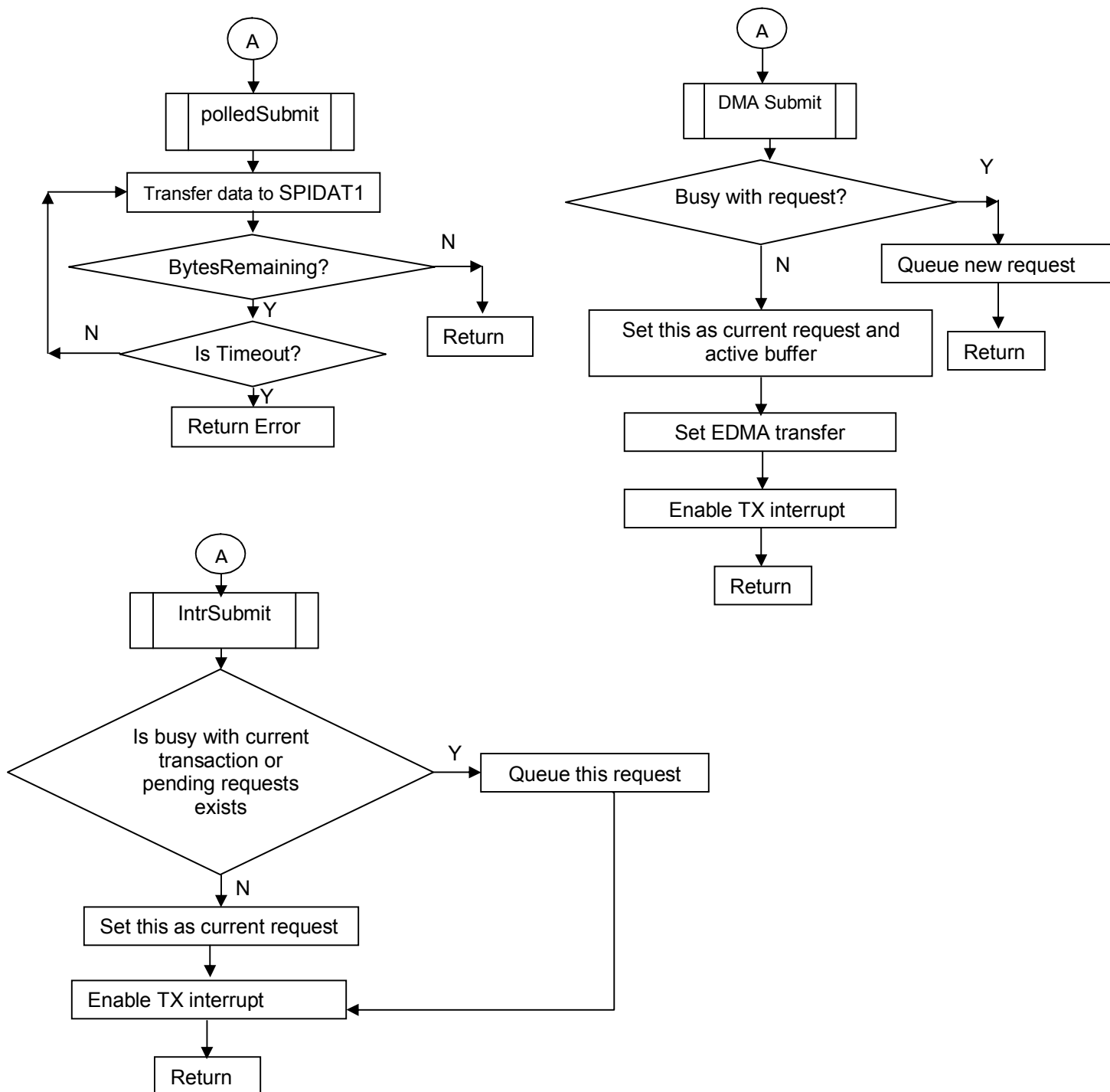
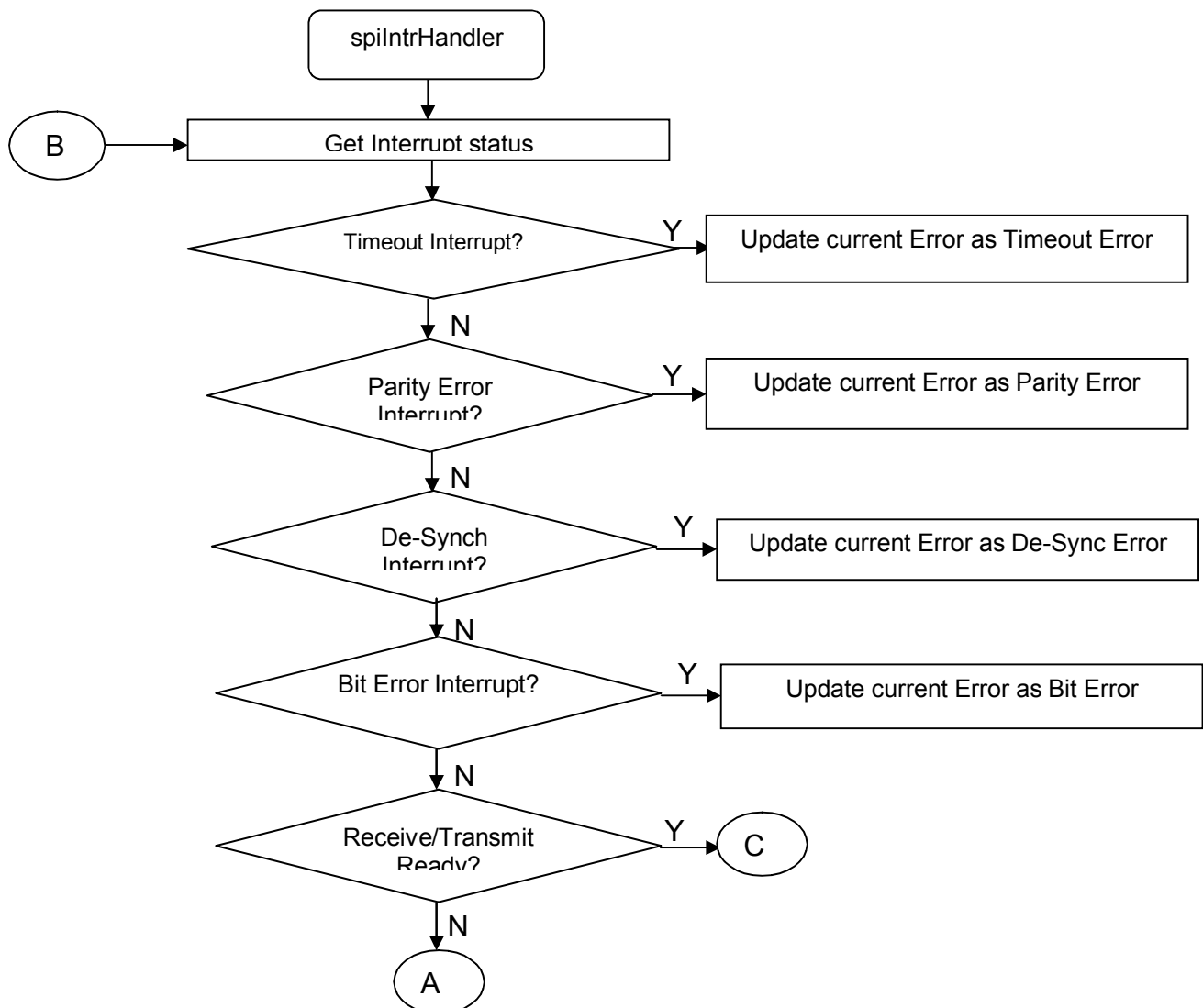


Figure 11 `Spi_submit` flow diagram()

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).

2.2.3.9 *spilntrHandler ()*



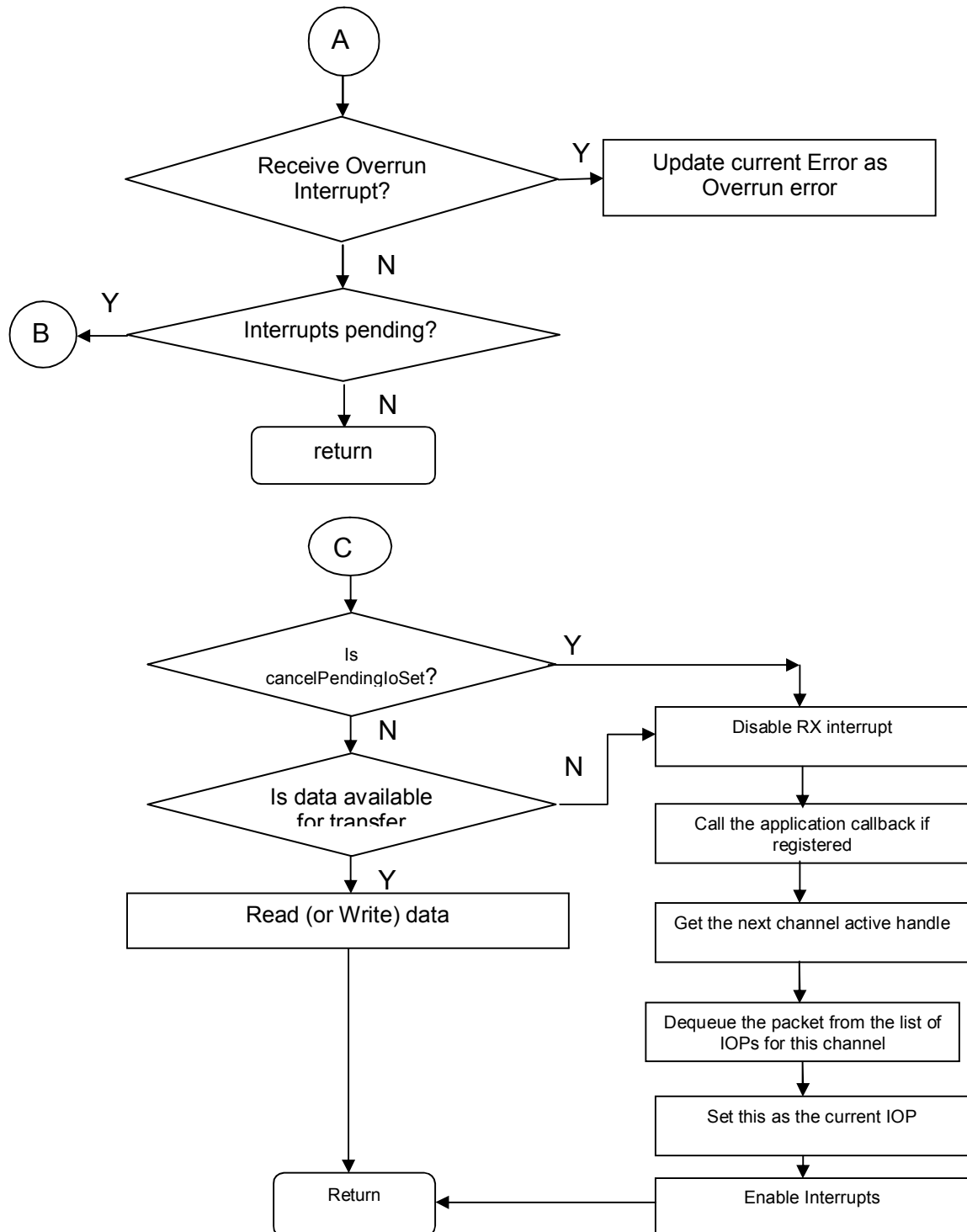


Figure 12 `spiIntrHandler ()`

2.2.3.10 *Spi_localCallbackTransmit/Receive*

When Spi is operating in the DMA interrupt mode for data transfer, the Spi driver during the channel open requests EDMA channels and registers a callback (*Spi_localCallbackTransmit/Receive ()*) for notification of the transfer completion status. Before starting any transfer, Spi 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 *Spi_localCallbackTransmit/Receive ()* as shown in the figure below.

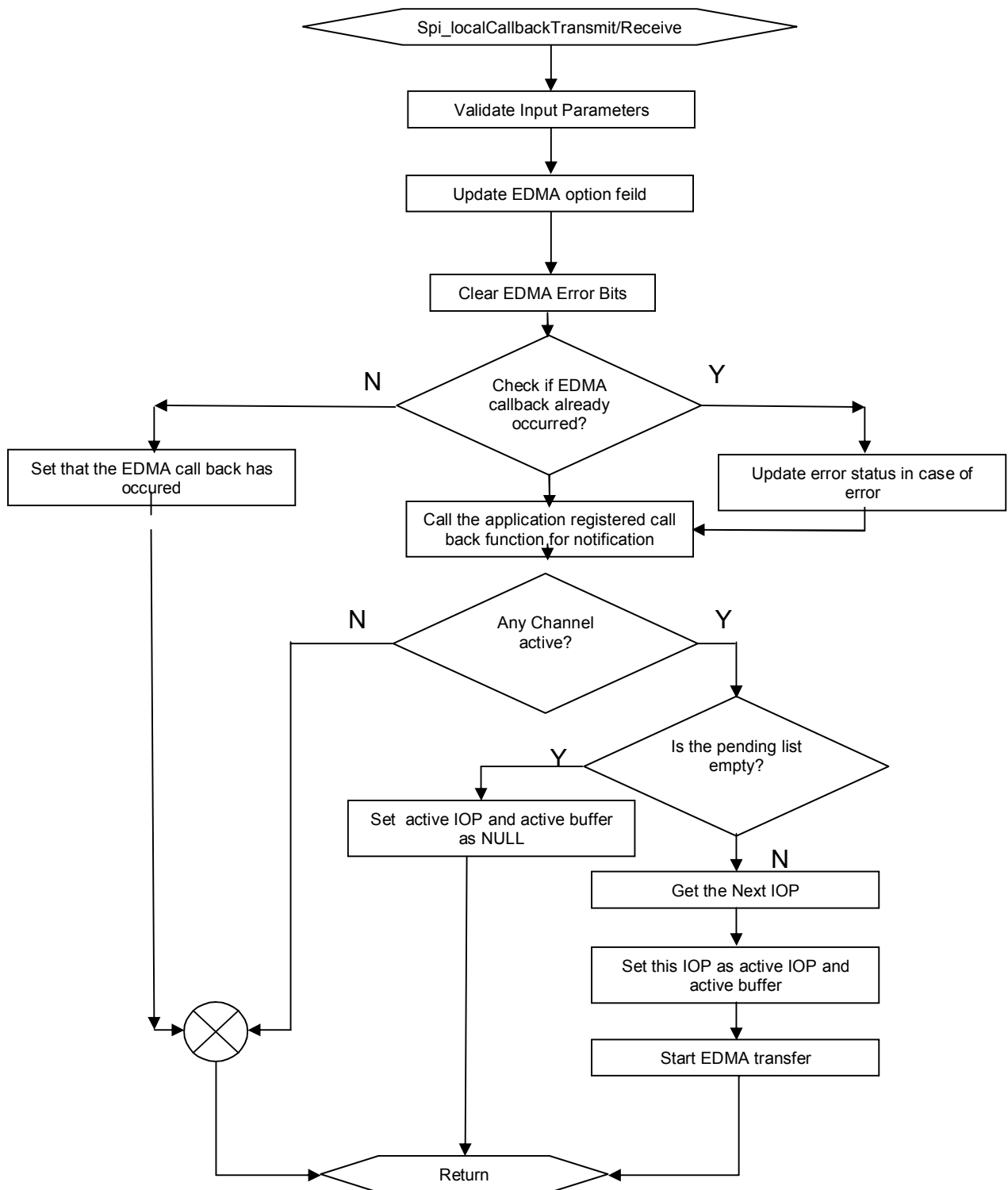


Figure 13 Spi_localCallbankTransmit/Receive()

2.3 Slave mode of operation

The SPI 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 prescaler are not valid since, the clock is received by this slave generated from the master. Delay configurations like CS active to transmit active delay also are not applicable to the slave mode of operation.

The option of slave mode (or master mode) of operation, should be supplied along with the HWConfig (device parameter) structure (masterOrSlave field) in device parameters, while instantiation of the module. This is because the mode of operation is fixed for one instance and cannot be changed dynamically or per-channel per instance.

Also note that in slave mode of the device only one channel can be opened.

Note:

For polled mode of operation the driver does not implement the task sleeping 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_CANCEL_PENDING_IO	ioctl to cancel the current pending IO packet
2	IOCTL_SET_CS_POLARITY	ioctl to set the chip select polarity
3	IOCTL_SET_POLLEDMODETIMEOUT	ioctl to set the polled mode timeout