SOFTWARE ARCHITECTURE TEMPLATE

BIOSPSP

McSPI Driver Design Document

Rev No	Author(s)	Revision History	Date	Approval(s)
0.1	Raghavendra G M		9th Mar - 2012	

Table of Contents

List Of Figures

1 System Context

The purpose of this document is to explain the device driver design for MCSPI peripheral using SYS/BIOS operating system running on DSP C674x, ARM M3 and Cortex A8.

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

1.2 Disclaimer

This is a design document for the MCSPI driver for the SYS/BIOS operating system.

1.3 Related Documents

1.	TBD	SYS/BIOS Driver Developer's Guide
2.		MCSPI Specs

1.4 Hardware

The MCSPI device driver design is in the context of SYS/BIOS running on DSP C674x core, ARM M3 and Cortex A8 core.

The MCSPI module core used here has the following bocks:

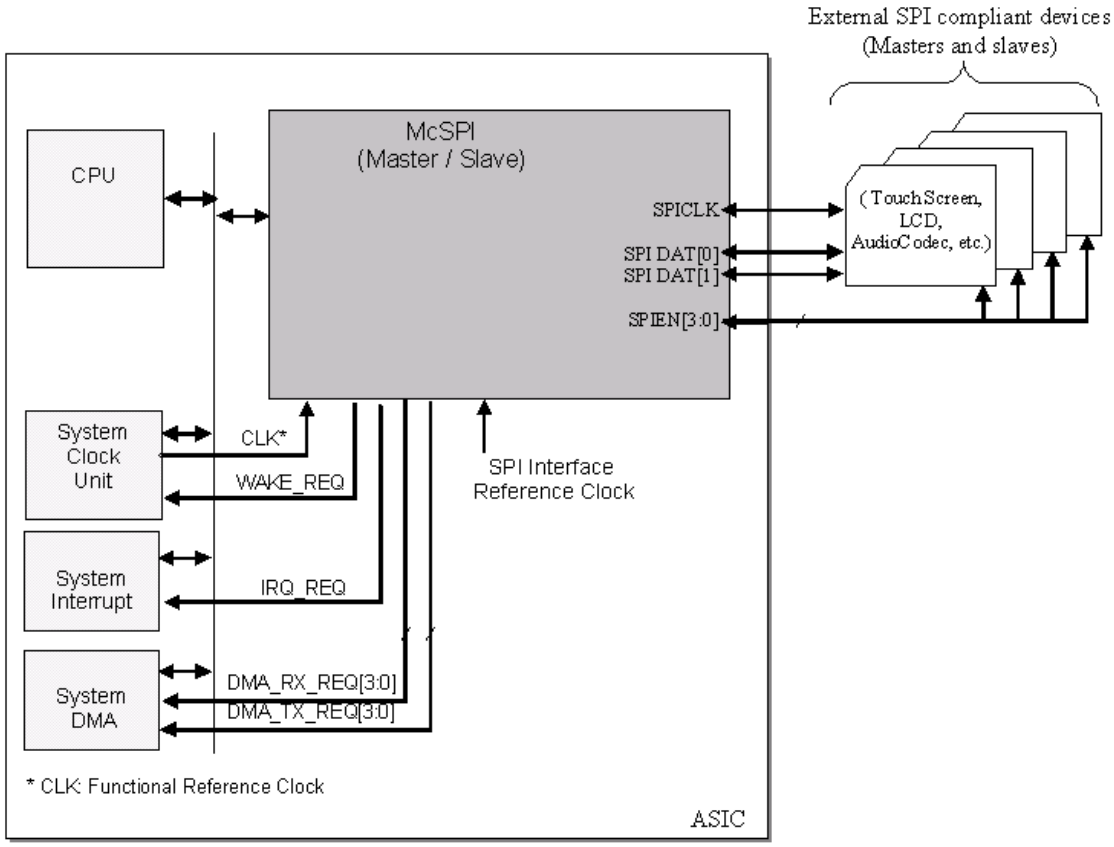


Figure 1 McSPI Block Diagram

1.5 Software

The MCSPI driver discussed here is running SYS/BIOS on the C674x DSP and ARM Cortex A8. Refer Userguide for limitation and the features.

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.

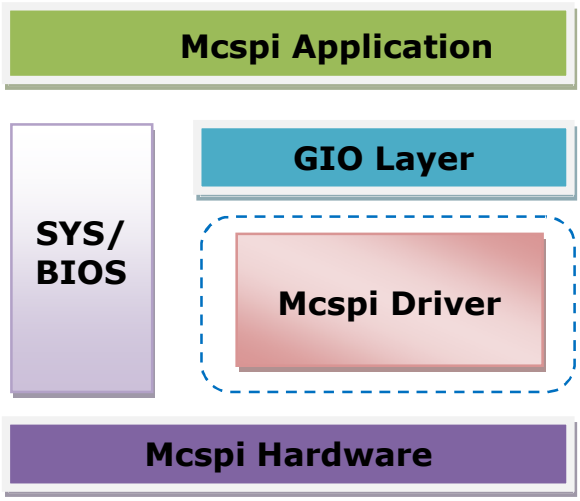


Figure 2 System Architecture

Driver module which this document discusses lies below the GIO layer, which is a class driver layer provided by SYS BIOS™. The McSPI driver would use the rCSL (register overlay) to access the Hardware and would use the SYS BIOS™ APIs for OS services.

The application would invoke the driver routines through the GIO API Calls. IOM is the component that performs the device specific operations. IOM Mini Driver directly controls MCSPI.

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

1.6 Component Interfaces

In the following subsections, the interfaces implemented by each of the sub-component are specified. The MCSPI driver module is object of IOM Driver class one may need to refer the IOM Driver documentation to access the McSPI driver in raw mode or could refer the GIO APIs to access the driver through GIO layer abstraction.

1.7 IOM Driver Interface

The IOM Driver constitutes the Device Driver Manifest to application. This adapts the Driver to SYS/BIOS™. This McSPI driver is intended to inherit the IOM Driver interfaces. Thus the McSPI driver module becomes an object of IOM Driver class. Please note that the terms “Module” and “Driver or IOM Driver” would be used in this document interchangeably.

The driver uses an internal data structure, called channel object, to maintain its state and parameters during execution. This channel is created whenever the application calls a GIO create call to the MCSPI IOM module. The channel object is held inside the Instance State of the module.

A driver which conforms to the IOM driver model exposes a well define set of interfaces -

- Driver initialization function.
- IOM Function pointer table.

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

- Mcspi_init()
- Mcspi_IOMFXNS

The Mcspi_init() is a startup function that needs to be called by the user (application) to initialize the module state and the Mcspi_deviceInstInfo structure of the McSPI instance.

  - The working of the Mcspi driver will be affected if this function is not called by the application prior to accessing the McSPI driver APIs.

The Model used by the device is identified by the function table. Hence, IOM_Fxns used for IOM model.

The McSPI driver exposes IOM function pointer table which contains various APIs provided by the McSPI driver. The IOM mini-driver implements the following API interfaces to the class driver**:**

Serial Num	Interfaces	Description
1	mcspiMdBindDev()	Allocates and configures the MCSPI port specified by devid.
2	mcspiMdUnBindDev()	Reset hardware and driver state and all deinitialization goes here. Effectively removes the usage of MCSPI instance.
3	mcspiMdCreateChan()	Creates a communication channel in specified mode to communicate data between the application and the MCSPI module instance.
4	mcspiMdDeleteChan()	Frees a channel and all its associated resources. Unregister interrupts.
5	mcspiMdControlChan()	Implements the IOCTLS for MCSPI IOM Driver module. All control operations go through this interface.
6	mcspiMdSubmitChan()	Submit an I/O packet to a channel for processing. Used for data transfer operations

1.7.1 CSLR Interface

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

1.8 Design Philosophy

This device driver is written in conformance to the SYS/BIOS IOM Driver model and handles communication to and from the MCSPI hardware.

1.8.1 The Module and 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 MCSPI device. The module word usage (MCSPI 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 (MCSPI 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 MCSPI instance. This is represented by the Instance_State in the MCSPI module configuration file.

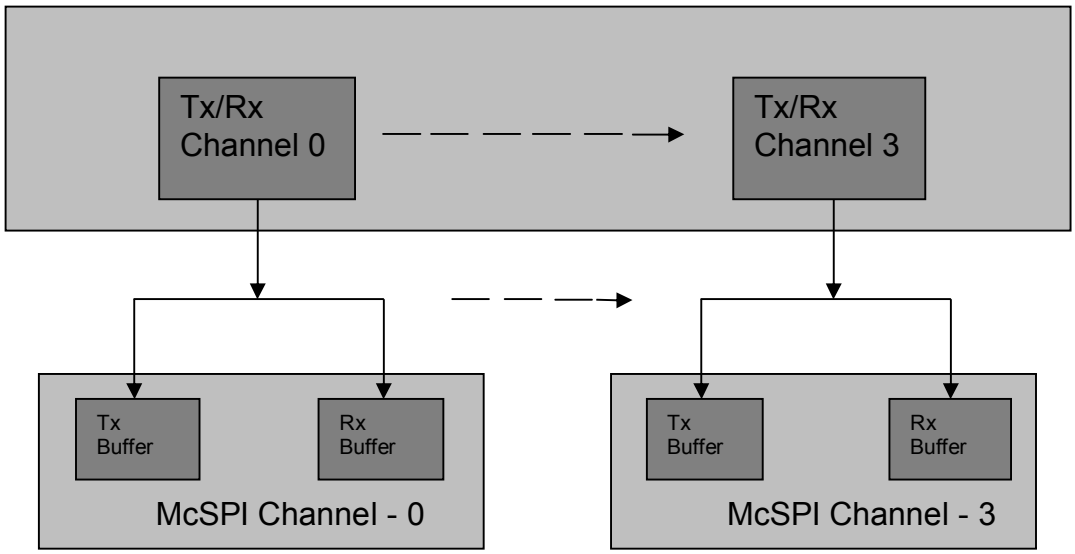


Figure 3 Instance Mapping

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

1.8.2 Design Constraints

MCSPI IOM Driver module imposes the following constraint(s).

- MCSPI driver shall not support dynamically changing modes between Interrupt, Polled and DMA modes of operation.
- The Driver shall not support the dynamic switching between physical channels in a single channel Master mode.
- There shall be only one channel for slave mode of operation.
- An instance can be configured as a slave or master.
- Driver shall not support the multiple channel open with FIFO enabled.
- The data word length can be configured only as 8 bits, 16-bits or 32-bits.
- Only logical channel 0 of a McSPI physical channel is supported and validated. In other words, multiple logical channels for a physical channel are not supported.

2 MCSPI Driver Software Architecture

This section details the data structures used in the MCSPI IOM driver module and the interface it presents. A diagrammatic representation of the IOM Driver 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 GIO create call to the MCSPI IOM Driver module. The channel object is held inside the Instance State of the module. The data structures used to maintain the state are explained in greater detail in the following *Data Structures* sub-section. The Mcspi hardware supports multiple channels called as physical channels. Each channel corresponds to a chip select and each channels has its own set of register set (like Rx register, Tx register etc).The following figure shows the static view of McSPI driver.

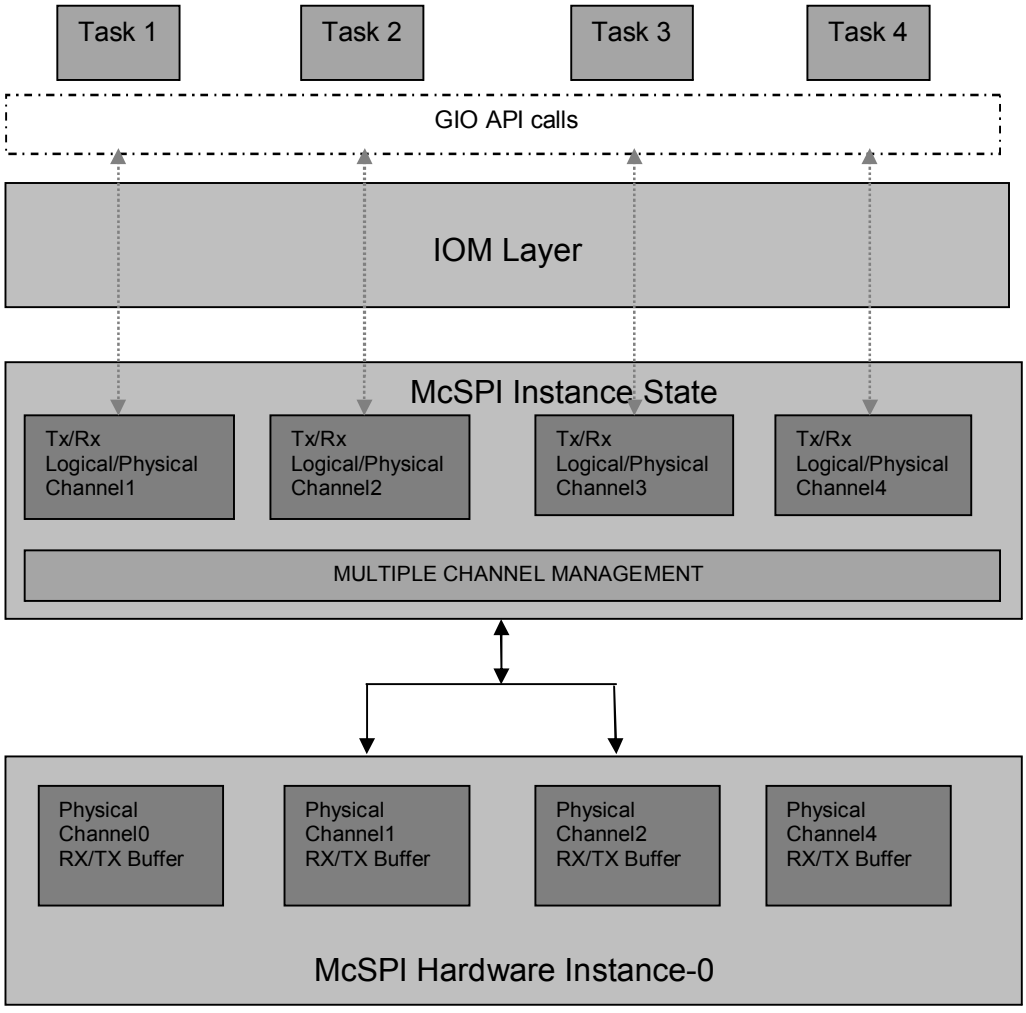


Figure 4 Mcspi driver static view

2.1.2 Data Structures

The IOM Driver employs the Instance State (Mcspi_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 IOM Driver module and the instance.

2.1.2.1 The Instance Object (Mcspi_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 (Mcspi_*Object)*	Description
1.	*instNum*	Preserve port or instance number of McSPI
2.	*devState*	State of the McSPI Either created or deleted
3.	*opMode*	Mode of operation
4.	deviceInfo	Instance specific information
5.	chanObj[]	Channel objects for the McSPI
6.	numOpens	The number of channels opened on this instance
7.	edma3EventQueue	Edma event Q to be requested, while requesting an EDMA3 channel
8.	isSingleChMasterMode	To indicate that the single channel master mode is

		enabled or not.
9.	isFifoEnabled	To indicate that the FIFO is enabled or not.
10	numOfPhyChannels	The number of physical channels available in a perticular Mcspi instance.
11	hwiNumber	Hardware interrupt number
12	enableCache	Submitted buffers are in cacheable memory.
13	spiHWconfig	McSPI Hardware configurations
14	dmaChanAllocated	Flag to inidicate EDMA channels allocation status
15	edmaCbCheck	Use to check occurance of EDMA callback
16	polledModeTimeout	Variable to hold Timeout to for the io operation
17	currentActiveChannel	The channel for which the current IO operation is in progress
18	isSlaveChannelOpened	This boolean track for having only one slave channel
19	syncSem	Semaphore to sync multiple tasks during the polled mode operation
20	prcmPwrmEnable	Option to enable or disable the PRCM control in the driver
21	pwrmInfo	structure to hold the PWMR related information
22	prcmHandle	Handle to the prcm instance to be used to switch on the instance
23	hwiHandle	Handle to the HWI created (only used in case of ARM)

24	stats	This is to collect the statistics info
25	crossBarEvtParam	Cross bar event params

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 McSPI 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 (Mcspi_*ChanObj)*	Description
1.	*mode*	Channel mode of operation: Input or Output
2.	*chanNum*	The physical channel number being used.
3.	*channelState*	state of the McSPI Either created or deleted
4.	*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
5.	*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
6.	*instHandle*	Spi Handle to access the spi channel params

7.	*hEdma*	Edma handle
8.	*wordCntInFifoMode*	To keep track of the number of word transfered.
9.	*rxDmaEventNumber*	The edma event for a Rx specific channel.
10	*txDmaEventNumber*	The edma event for a Tx specific channel
11	*dmaEnable*	Indicate to the Dma.
12	*fifoEnable*	Indicate to use the FIFO
13	*spiChipSelectHold*	To hold the SPIEN during its transaction the user can select this as a channel parameter
14	*activeIOP*	The pointer to the current active packet (being processed)
15	*currentActiveChannel*	This specifies the current active channel
16	*dataParam*	The parameters for the data transfer
17	*pendingState*	Shows whether io is in pending state or not
18	*abortAllIo*	Shows whether all IO should be aborted
19	*currError*	current error flag
20	*txBuffer*	buffer for TX operation of transceive operation
21	*rxBuffer*	buffer for RX operation of transceive operation
22	*queuePendingList*	pending Iop List head
23	*taskPriority*	this will hold the priority of the task that created this channel
24	*txBufferLen*	Length of the TX buffer
25	*rxBufferLen*	Length of the RX buffer
26	*txBufFlag*	Flag to indicate if the TX buffer is suppiled by user or is NULL

27	*rxBufFlag*	Flag to indicate if the RX buffer is suppiled by user or is NULL
28	*txTransBuf[128]*	Buffer to be used when the user supplied buffer is NULL
29	*rxTransBuf[128]*	Buffer to be used when the user supplied buffer is NULL
30	*configChfmt*	Data Format Configuration values
31	*chipSelTimeControl*	Number of interface clock cycles introduced between CS toggling and first or last edge of SPI clock
32	*queueFloatingList*	list to manage floating packets in DMA
33	*tempPacket*	Temp IOP holder
34	*isTempPacketValid*	Valid packet flag in EDMA callback
35	*submitCount*	Number of submit calls pending
36	*pramTblRx[2u]*	Logical channel numbers of EDMA, which are used for linking
37	*pramTblAddrRx[2u]*	Physical address of logical channel numbers of EDMA, which are used for linking
38	*pramTblTx[2u]*	Logical channel numbers of EDMA, which are used for linking
39	*pramTblAddrTx[2u]*	Physical address of logical channel numbers of EDMA, which are used for linking
40	*nextLinkParamSetToBeUpdated*	Used to store the next index of link param to be updated
41	*currentPacketErrorStatus*	This member will hold the error status -normally updated from cpu interrupt thread and is used in EDMA completion thread
42	*edmaCbCheck*	se to check occurance of EDMA callback
43	*enableErrIntr*	This boolean enables/disables error Interrupts in DMA mode of operation

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

S.No	Structure Elements	Description
1	*instNum*	Instance Number
2	opMode	Mode of operation
3	hwiNumber	Hardware interrupt number
4	enableCache	To enable/disable the cache
5	edma3EventQueue	To select the event queue number.
6	spiHWCfgData	McSPI Hardware Data Configuartion
7	prcmPwrmEnable	PRCM control enable disable
8	pllDomain	Pll domain where the device is configured
9	polledModeTimeout	Variable to hold Timeout to for the io operation.

10	prcmDevId	prcm device ID
11	enableErrIntr	Flag to Enable/Disable Error Interrupts in DMA mode

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 McSPI 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	chipSelTimeControl	Number of interface clock cycles introduced between CS toggling and first or last edge of SPI clock
3	fifoEnable	To have the fifo or not in DMA mode
4	spiChipSelectHold	Sets the FORCE bit in the Conf register, if the slave requires to hold the SPIEN during its transaction the user can select this as a channel parameter
5	chanNum	To indicate which physical channel needs to be opened
6	crossBarEvtParam	cross bar event params /* Valid only for M3 core*/

2.1.2.1 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 McSPI 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.

McSPI Hardware Configuration Structure

S.No	**Structure Elements** spiHWconfig	**Description**
1	masterOrSlave	This variable is used to configure SPI device either in Master or in Slave mode
2	singleOrMultiChEnable	Indicates the McSPI is in multi channel mode or in single channel mode
3	pinOpModes	McSpi Operation Modes
4	fifoRxTrigLvl	The receive FIFO trigger level.
5	fifoTxTrigLvl	The receive FIFO trigger level.
6	configChfmt[]	Physical Channel specific Configuration values

2.2 Dynamic View

2.2.1 The Execution Threads

The McSPI IOM Driver module involves following execution threads:

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

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

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

2.2.3 Functional Decomposition

The McSPI driver,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.

2.3 Driver API's and Flowcharts

2.3.1.1 Driver Creation (Driver Initialization and Binding)

The McSPI IOM driver needs the global data used by the McSPI driver to be initialized before the driver can be used. The initialization function for the McSPI driver is not included in the IOM_Fxns table, which is exported by the McSPI driver; instead a separate extern is created for use by the SYS/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
- PRCM Id for the module.

The function also sets the "inUse" field of the McSPI instance module object to FALSE so that the instance can be used by an application which will create it.

Please refer to the figure below for the typical control flow during the initialization of the driver.

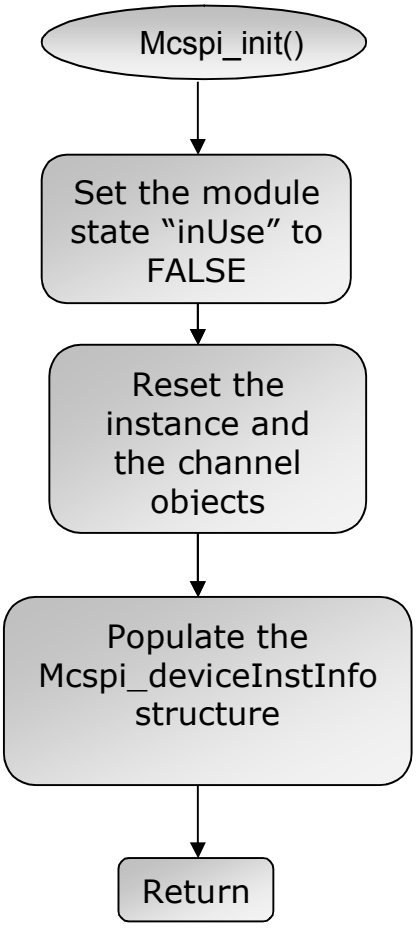


Figure 5 Mcspi_init() flow diagram

2.3.1.1.1 mcspiMdBindDev()

The binding function (mcspiMdBindDev) of the McSPI IOM mini-driver is called in case of a static or dynamic creation of the driver. In case of dynamic creation application will call GIO_addDevice() API to create the device instance. Otherwise, the instance could *be created statically* through a configuration, *.cfg file by calling GIO.addDeviceMeta().

Each driver instance corresponds to one hardware instance of the McSPI. 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 McSPI device instance in the PRCM module.
- Reset the McSPI device and disable the device.
- Return the device handle to the GIO 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 “Mcspi_Params”. Refer to Mcspi_Params section for more details.

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

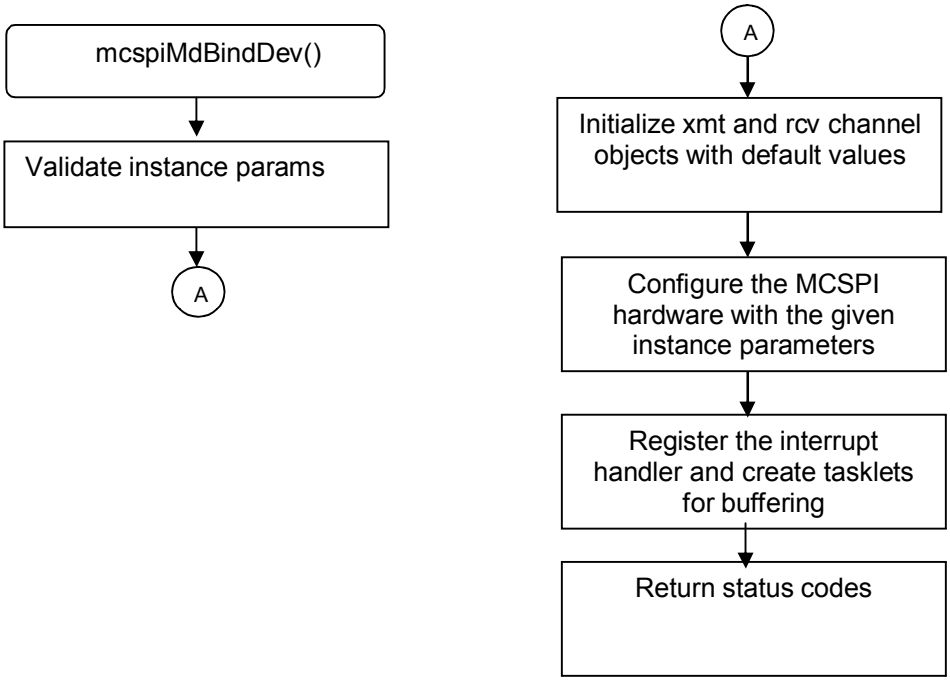


Figure 6 mcspiMdBindDev() flow diagram

2.3.1.2 Driver Deletion (Driver DeInitialization and UnBinding)

The McSPI driver provides the interfaces for deleting the driver instance. The mcspiUnBindDev 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 all channels 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 PRCM.
- Unregister the notification registered with the BIOS power management module(only if the driver supports power management)

2.3.1.2.1 *mcspiMdUnBindDev()*

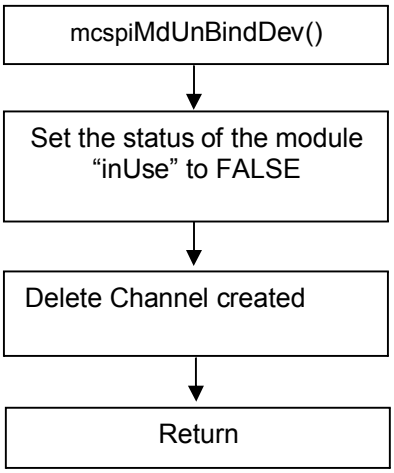


Figure 7 mcspiMdUnBindDev() flow diagram

2.3.1.3 *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 than a channel will correspond to a physical Channel on the McSPI. Hence an application can create at most 2 channels.

The application can create a communication channel by calling GIO_create() API which in turn calls McSPI IO mini driver's mcspiMdCreateChan() function. The application shall call GIO_create() with the appropriate "mode" (GIO_INOUT) parameter for the type of the channel to be created.

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 "Mcspi_ChanParams" structure to specify the parameters to configure the channel.

The mcspiMdCreateChan() 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 McSPI is configured with the appropriate parameters for the channel.
- The clock divider settings for the channel are configured.

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.

2.3.1.3.1 *mcspiMdCreateChan()*

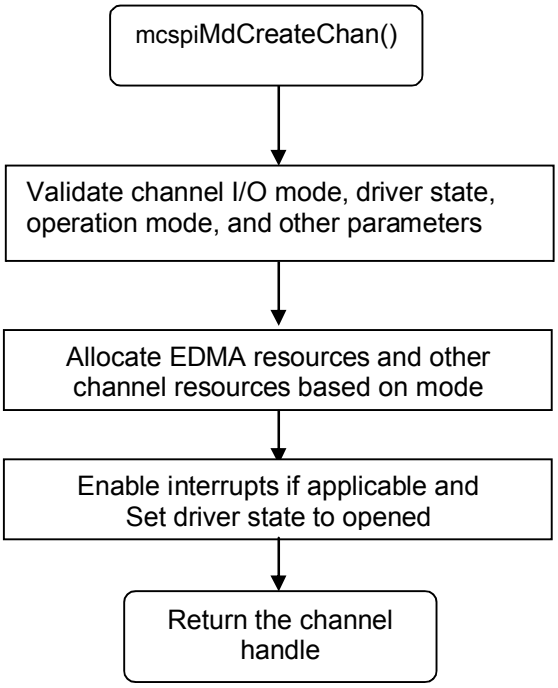


Figure 8 mcspiMdCreateChan() flow diagram

2.3.1.4 *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 Mcspi driver provides the "mcspiMdDeleteChan" API to delete a previously created McSPI channel.

The typical activities performed during the channel deletion are as follows:

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

2.3.1.4.1 mcspiMdDeleteChan()

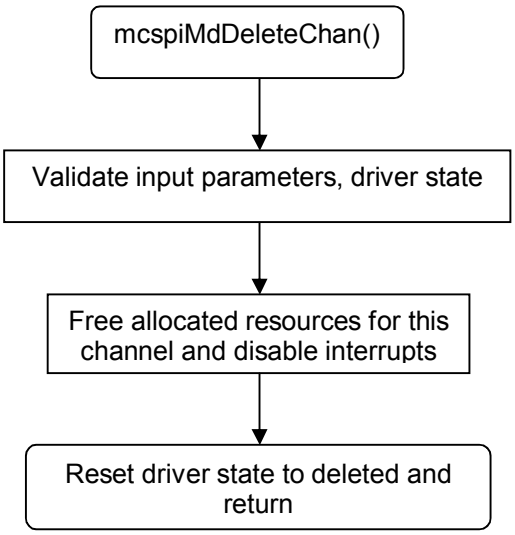


Figure 9 mcspiMdDeleteChan() flow diagram

2.3.1.5 IO submit

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

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 driver by inheritance of the IOM Driver 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 in progress and shall return IOM_PENDING status to the GIO layer. The GIO layer takes care of the pending status. The interrupt handler or the EDMA callback functions call the application callback (essentially the GIO 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 GIO. 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.

2.3.1.5.1 mcspiMdSubmitChan()

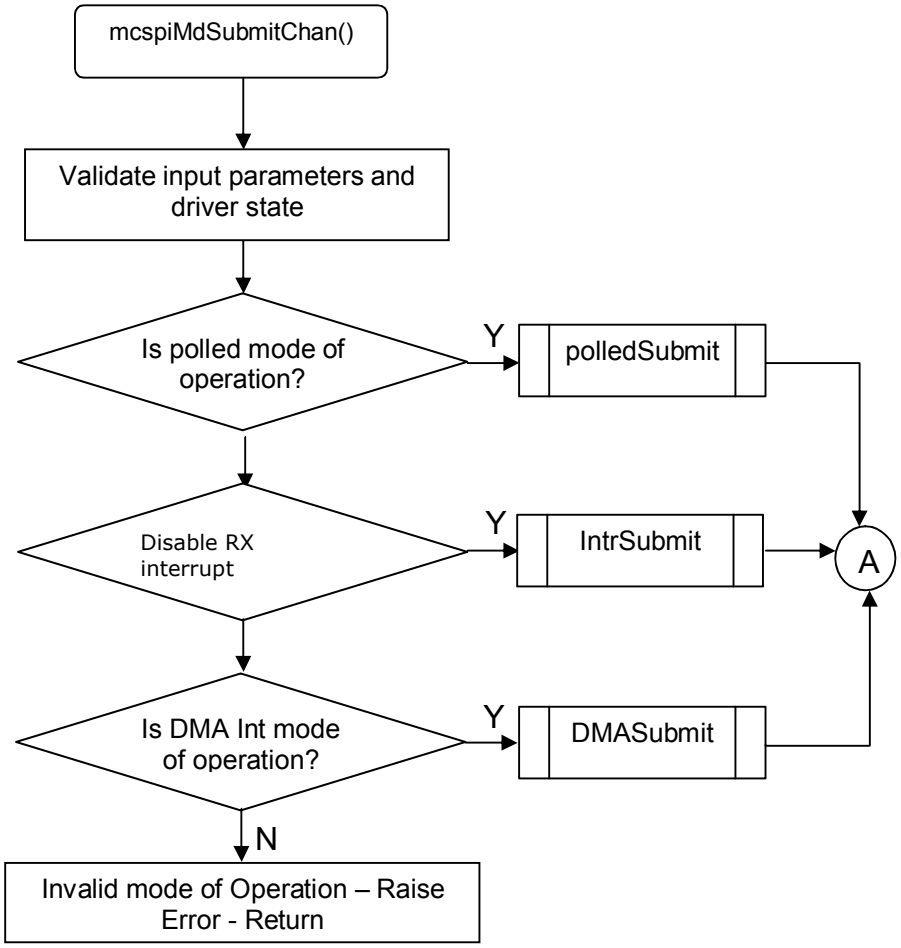
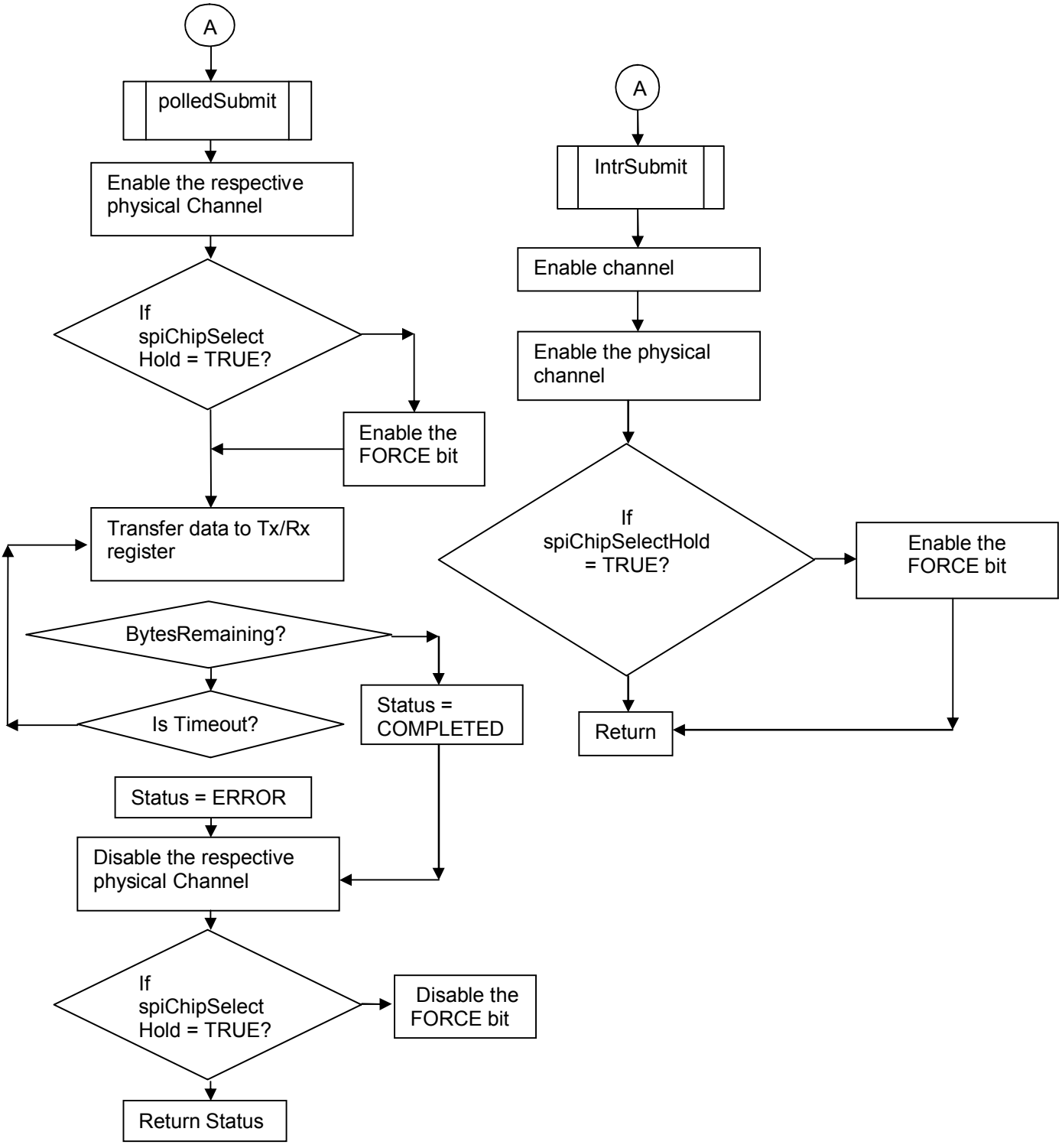


Figure 10 mcspiMdSubmitChan() flow diagram


A
polledSubmit
Enable the respective physical Channel
If spiChipSelect Hold = TRUE?
Enable the FORCE bit
Transfer data to Tx/Rx register
BytesRemaining?
Is Timeout?
Status = COMPLETED
Status = ERROR
Disable the respective physical Channel
If spiChipSelect Hold = TRUE?
Disable the FORCE bit
Return Status
A
IntrSubmit
Enable channel
Enable the physical channel
If spiChipSelectHold = TRUE?
Enable the FORCE bit
Return

2.3.1.5.2 EDMA interaction with Mcspi IOM Driver channel

Each operation – transmit/receive – of the McSPI will have EDMA channel allocated to it to service its data requirements. The two operations of the McSPI are provided with individual EDMA event each to trigger transfers. The IOM Driver also registers the appropriate callback function to service interrupts raised by the individual channels.

The control flow of the callback function is discussed in the next section.

As a reference case, let us take up EDMA channel servicing the transmit section of the McSPI. The receive operation would be similar in its treatment.

Depending on the channel configuration, two cases exist in channel configuration:

While creating the transmit channel, the IOM Driver requests for a single EDMA channel to service this. The McSPI transmit event is registered with this channel as a trigger for data transfer.

The EDMA interacts with the transmit section of the McSPI using its data address. During McSPI channel create, the EDMA channel to service this is programmed with the destination address, elements, and block sizes to generate an interrupt after a block is transferred. At this time two extra EDMA PaRAM tables are also requested. For a transmit channel the source of transmit is taken as the application supplied buffer or default buffer. The two EDMA PaRAM tables are used for linking. The EDMA channel is then turned on and waits for the McSPI to trigger it for data transfer. Each data transfer will happen at every event.

When the McSPI channel is enabled, it triggers EDMA event. The EDMA channel transfers one element for every event triggered, which is done every time the McSPI consumes the previous element transferred.

As per the programming of the EDMA, when the last element of the block is transferred, a BLOCKIE interrupt is raised by this channel, which is serviced by the “Callback” function.

This function checks the channels pending queue for further data packets.

If the channel’s pending queue holds data packets to be processed, the “Callback” picks the next packet from the queue and programs the EDMA channel with its source address. This ensures a constant data flow to the McSPI transmitter.

While creating the transmit channel, the IOM Driver requests for a single EDMA channel to service all the requests. The McSPI transmit event is registered with this channel as a trigger for data transfer.

Each data packet would contain data to service one block. The “Callback” would be called on to service the BLOCKIE interrupt generated by the channel. This would then program the source address with the next data packet contents, as outlined in the earlier case.

Also for receive data similar to the transmit data we request for two PaRAM tables used for linking. Once in EDMA callback ISR, if there are no other packets to be linked, it will complete the entire transfer and call the application callback.

2.3.1.5.3 EDMA Params and Linking

McSPI Data requires that it should be handled with no or very little latencies. Hence it is required that the McSPI handles the next pending requests almost immediately as soon as the current request is completed.

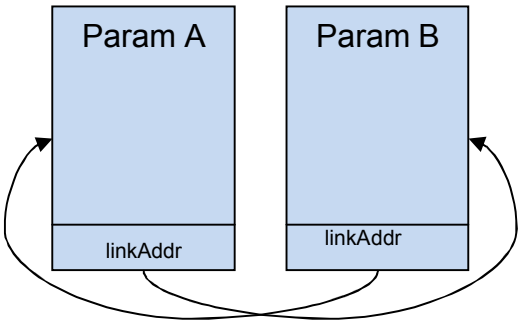
To handle these stringent timing constraints the McSPI is programmed in the EDMA mode with the facility to link the next IO request in the queue immediately with the usage of EDMA channel linking facility.

The explanation of the linking scenario is as explained below.

Initially during the opening of the channel, the McSPI IOM Driver will request for two spare EDMA PaRAM entries for the linking purpose.

The IOM Driver maintains two lists per channel for the handling of the requests. One is a queueFloatingList and the other is queuePendingList.

The “queueFloatingList” will contain a maximum of two packets and these are the packets that are currently loaded in to the spare EDMA PaRAM sets. The remaining requests (other than the outstanding 2) will be queued up in the “queuePendingList”.MdSubmitChan()



2.3.1.5.4 DMA Submit

The application invokes GIO_submit, GIO_write, or GIO_issue API's for data transfer using McSPI. These API's in turn submits a IOM_packet containing all the transfer parameters needed by the IOM Driver to program the underlying hardware for data transfer.

The McSPI IOM Driver provides the "mcspiMdSubmitChan" API to handle all the IO requests for the McSPI driver from the application. The IOM_Packet has the command code embedded inside it.

The McSPI_submit handles the following command

1. IOM_INOUT

The McSPI driver works essentially in the asynchronous IO mode.

Asynchronous IO Mechanism

A driver is said to be in asynchronous mode when multiple IO requests can be submitted to the driver by a thread without causing the thread to block till the completion of the IO request. Usually after the completion of the IO a callback function registered by the application will be called to notify the thread of the IO completion.

The McSPI IOM Driver's async mode working can be broadly divided in to the following sections.

1. IOM_Packet Queuing
2. EDMA callback and subsequent packet loading.

2.3.1.5.5 IOM_Packet Queuing

The most important aspect of the McSPI IOM Driver for the async mode is the queuing of the IO requests while a packet is currently under processing.

When the IOM Driver submit function is called to submit a new IO request the following is the sequence of events that take place in the driver.

1. The command embedded within the IO packet is verified for validity.
2. The buffer address supplied by the application is validated.
3. Since, essentially the McSPI driver works in EDMA mode the cache is flushed.

The McSPI driver essentially uses two lists to maintain the IO packets

1. **queueFloatingList** is the list that contains the IO requests which are currently loaded in to the spare param sets of the EDMA. This should hold a maximum of two IO requests i.e. the one currently being processed and the other being the packet to be loaded next.
2. **queuePendingList** is the lists that holds the pending IO requests after the floating list is FULL. Can hold any number of packets.

Now, there are two possible scenarios.

1. The “queueFloatingList” is empty i.e. there are no current requests under processing.
2. The “queueFloatingList” is full i.e. there are currently two request also queued in the driver.

queueFloatingList NOT FULL

When the queueFloatingReqList is not full, It means that the driver is currently having either no request or one outstanding request only. In this case the IO request is directly queued in to the “queueFloatingList”.

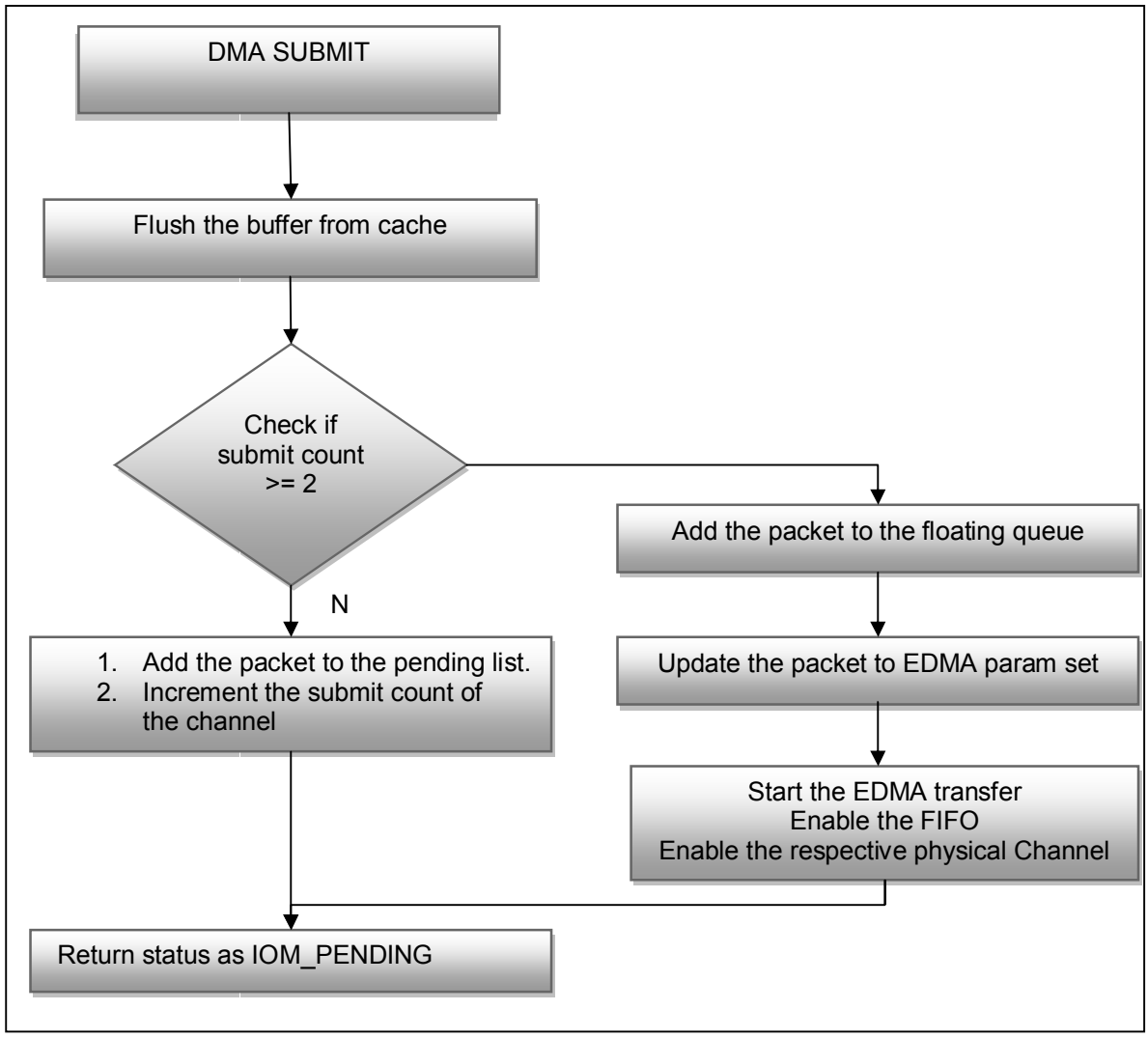
Now the new EDMA parameters pertaining to the IO packet are loaded in to the spare EDMA param sets (requested while opening the channel).

In case that this is the first packet to be loaded in to the “queueFloatingList” Then the EDMA transfer needs to be enabled.

queueFloatingList FULL

When the queueFloatingReqList is full, it means that there are currently two outstanding requests in the EDMA driver and that no further requests can be submitted to the EDMA. Hence the IO packet is added to the pending IO list i.e. “queuePendingList”.

The below figure shows the flow diagram in the mcspiMdSubmitChan() in DMA context


DMA SUBMIT
Flush the buffer from cache
Check if submit count >= 2
N
Add the packet to the floating queue
Update the packet to EDMA param set
Start the EDMA transfer
Enable the FIFO
Enable the respective physical Channel
1. Add the packet to the pending list.
2. Increment the submit count of the channel
Return status as IOM_PENDING

2.3.1.5.6 mcspiMdControlChan()

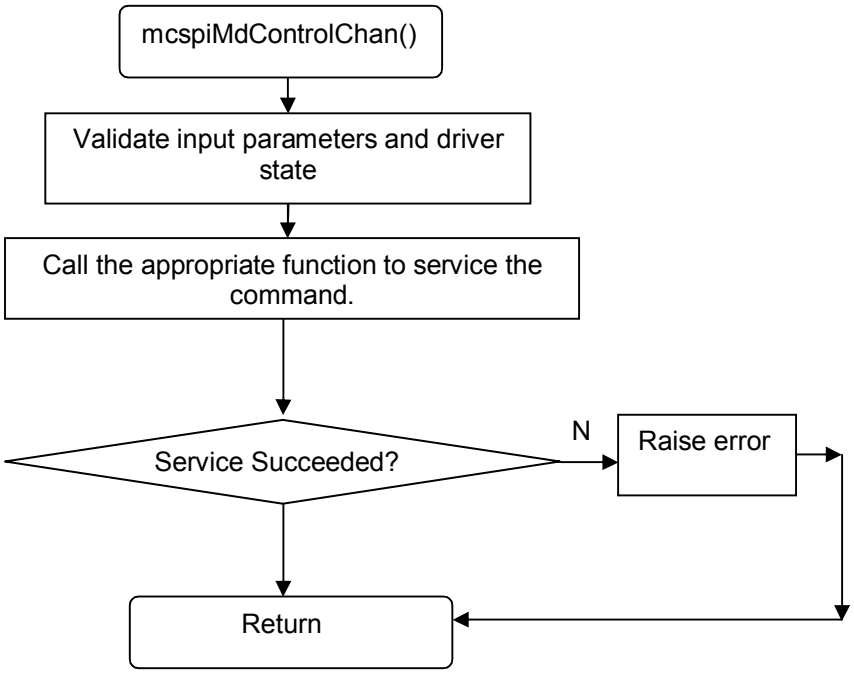
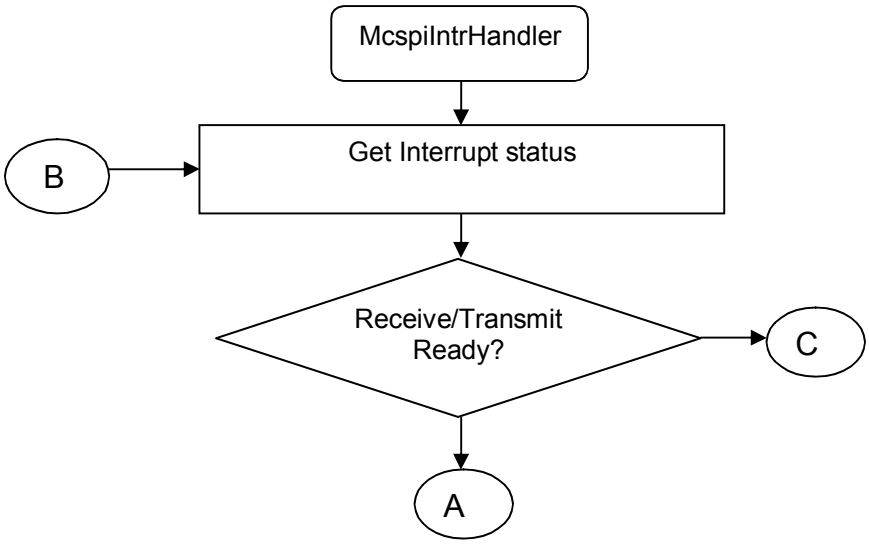
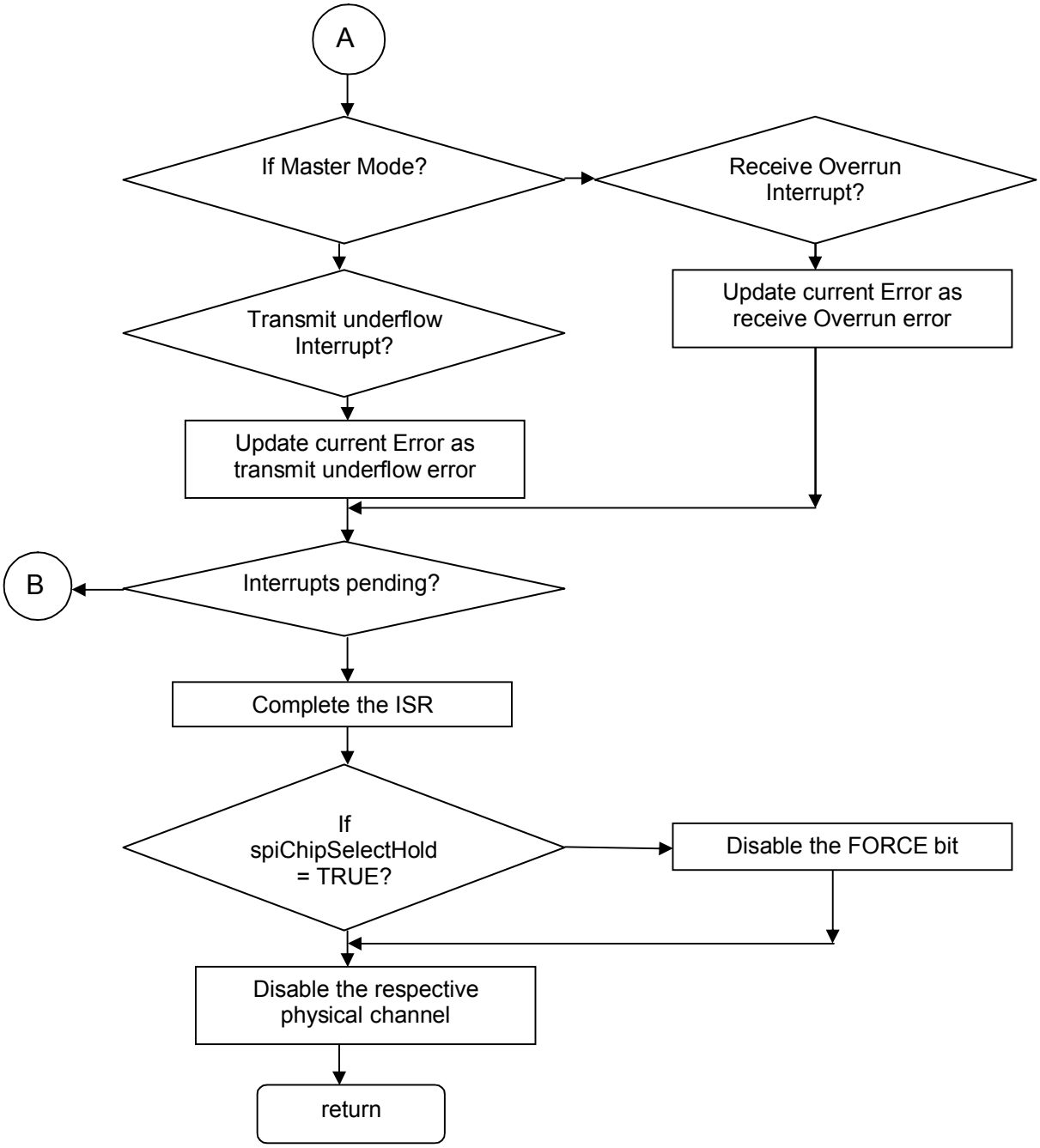


Figure 11 mcspiMdControlChan() flow diagram

2.3.1.6 McspiIntrHandler()




A
If Master Mode?
Receive Overrun Interrupt?
Update current Error as receive Overrun error
Transmit underflow Interrupt?
Update current Error as transmit underflow error
B
Interrupts pending?
Complete the ISR
If spiChipSelectHold = TRUE?
Disable the FORCE bit
Disable the respective physical channel
return

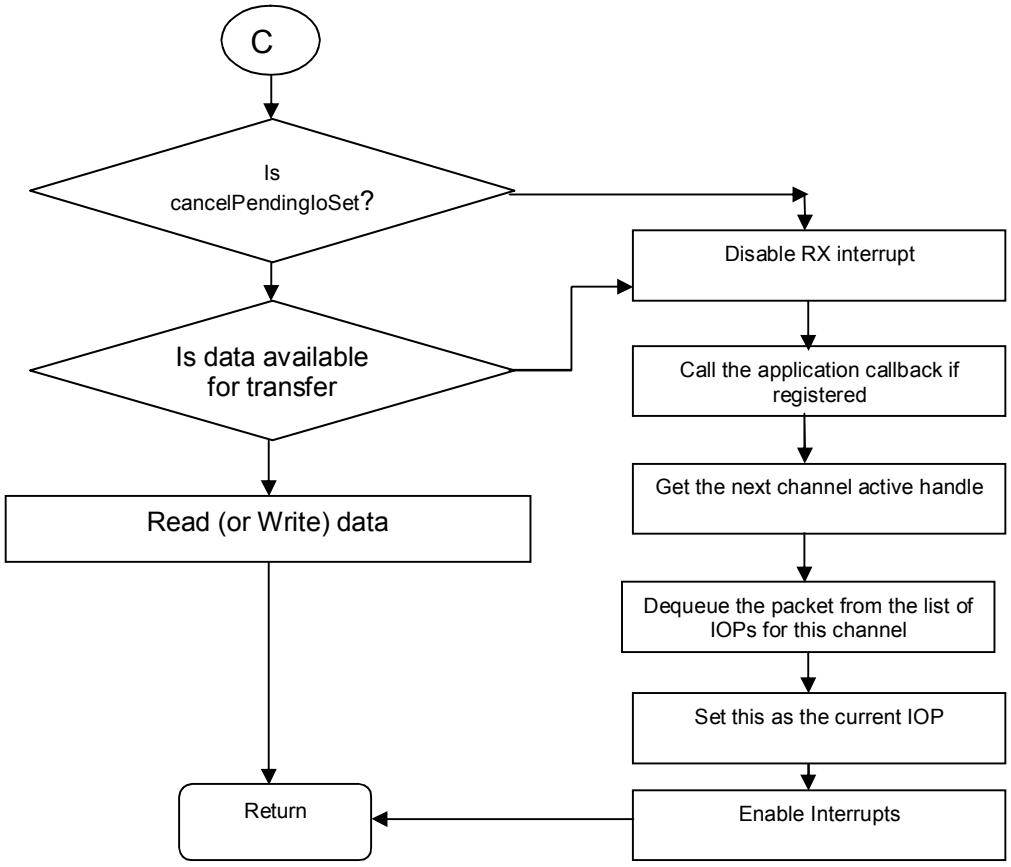


Figure 12 mcspiIntrHandler

2.3.1.7 *Mcspi_localCallbackTransmit/Receive*

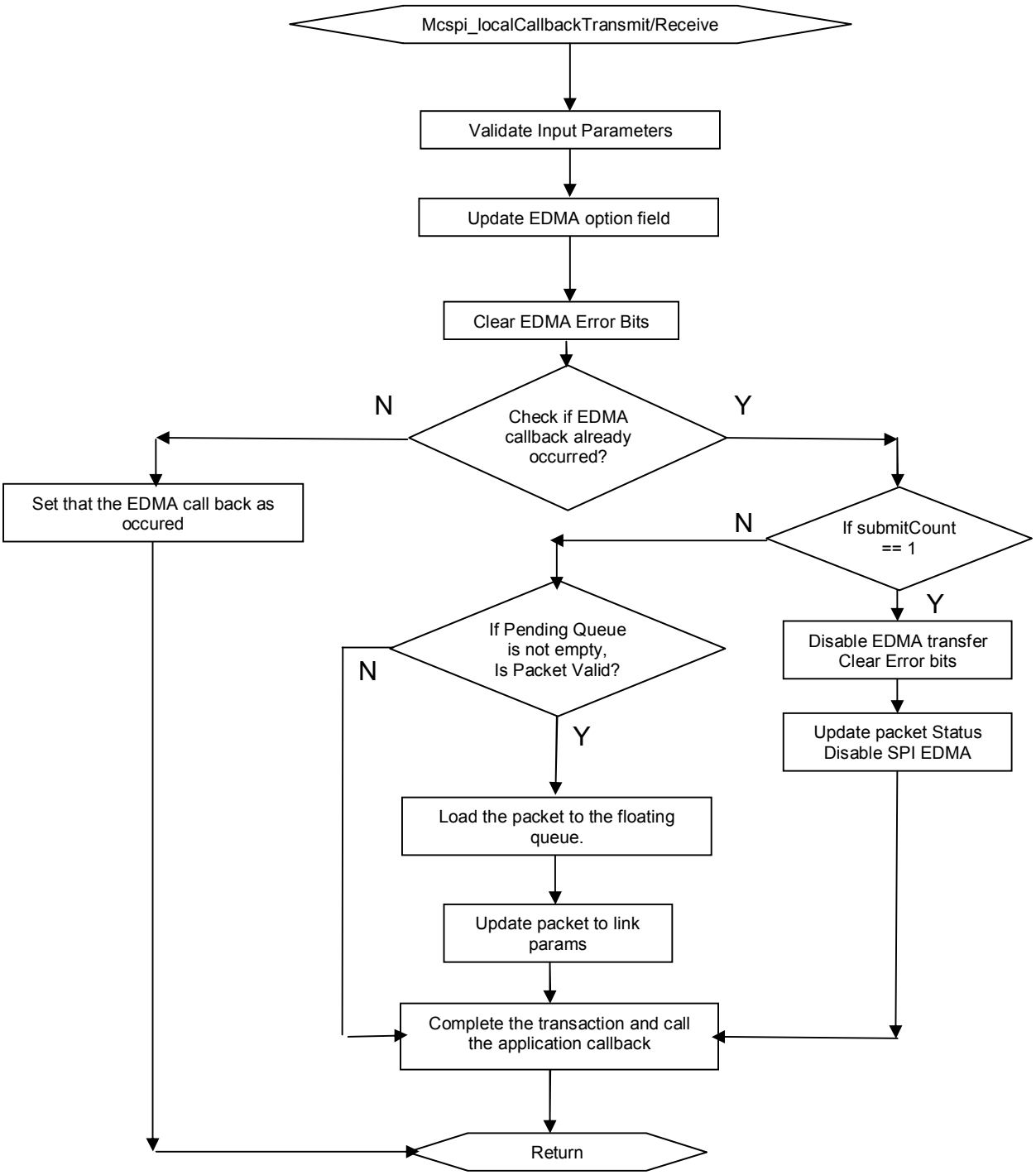


Figure 13 mcspi_localCallbackTransmit/Receive()

2.3.1.7.1 Mcspi EDMA Callback

Before starting any transfer, Mcspi 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. The McSPI driver has callback functions registered for both receive and transmit EDMA events. The callback function is invoked after the completion of the transfer. The EDMA completion status is updated in to the IOM_Packet error field as the request completion status. In case of Error the status is updated as “IOM_EABORT” else “IOM_COMPLETED” is updated. Once the status of the packet is completed, the “queuePendingList” is checked to find if there are any packets pending.

queuePendingList is EMPTY

In case that the”queuePendingList" list is empty, it means that there are no outstanding requests in the McSPI driver.

queuePendingList is NOT EMPTY

In case that the “queuePendingList” is not empty, it means that there are pending requests in the McSPI driver. The pending packet is taken from the “queuePendingList” and is programmed in to the empty EDMA param set and the currently executing packet’s param set is linked with the newly loaded param set.

2.4 Slave mode of operation

The McSPI 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 other than physical channel ‘0’ cannot be used since, the multichannel slave is not supported. The clock configurations are also invalid for 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, DMA 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_SPIEN_POLARITY	Ioctl to set the chip select polarity
3	IOCTL_SET_POLLEDMODETIMEOUT	Ioctl to set the polled mode timeout
4	IOCTL_SET_TRIGGER_LVL	Ioctl to set the trigger level for FIFO