BIOSPSP

UART Driver Design Document

Rev No	Author(s)	Revision History	Date	Approval(s)
0.1	Raghavendra G M		6th Feb- 2012	

Table of Contents

1 System Context 4
1.1 Terms and Abbreviations 4
1.2 Disclaimer 4
1.3 Hardware 5
1.4 Software 6
1.4.1 Operating Environment and dependencies 6
1.4.2 System Architechture 6
1.5 Component Interfaces 7
1.5.1 IOM Driver Interface 7
1.5.2 CSLR Interface 8
1.6 Design Philosophy 9
1.6.1 The Module and Instance Concept 9
1.6.2 The Channel Concept 9
1.6.3 Design Constraints 10
2 UART Driver Software Architecture 10
2.1 Static View 10
2.1.1 Functional Decomposition 10
2.1.2 Data Structures 11
2.2 Dynamic View 17
2.2.1 The Execution Threads 17
2.2.2 Input / Output using UART driver 17
2.2.3 Functional Decomposition 18
2.3 Driver API's and Flowcharts 18
3 APPENDIX A – IOCTL commands 32

List Of Figures

Figure 1 UART Block Diagram 5
Figure 2 System Architecture 6
Figure 3 Instance Mapping 9
Figure 4 UART driver static view 11
Figure 5 Uart_init() flow diagram 19
Figure 6 uartMdBindDev() flow diagram 20
Figure 7 uartMdUnBindDev() flow diagram 21
Figure 8 uartMdCreateChan() flow diagram 23
Figure 9 uartMdDeleteChan() flow diagram 24
Figure 10 uartMdSubmitChan() flow diagram 25
Figure 11 uartMdControlChan() flow diagram 28
Figure 12 uartIsr() flow diagram 29
Figure 13 Uart_localIsrEdma() flow diagram 31

1 System Context

The purpose of this document is to explain the device driver design for UART 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 UART driver for the SYS/BIOS operating system. Although the current design document explain the UART driver in the context of the BIOS 6.x driver implementation, the driver design still holds good for the BIOS 5.x driver implementation driver. 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 UART driver 5.x.

1.3 Hardware

The UART device driver design is in the context of SYS/BIOS running on DM8148 on DSP C674x, ARM M3 and Cortex A8.

The UART module core used here has the following bocks:

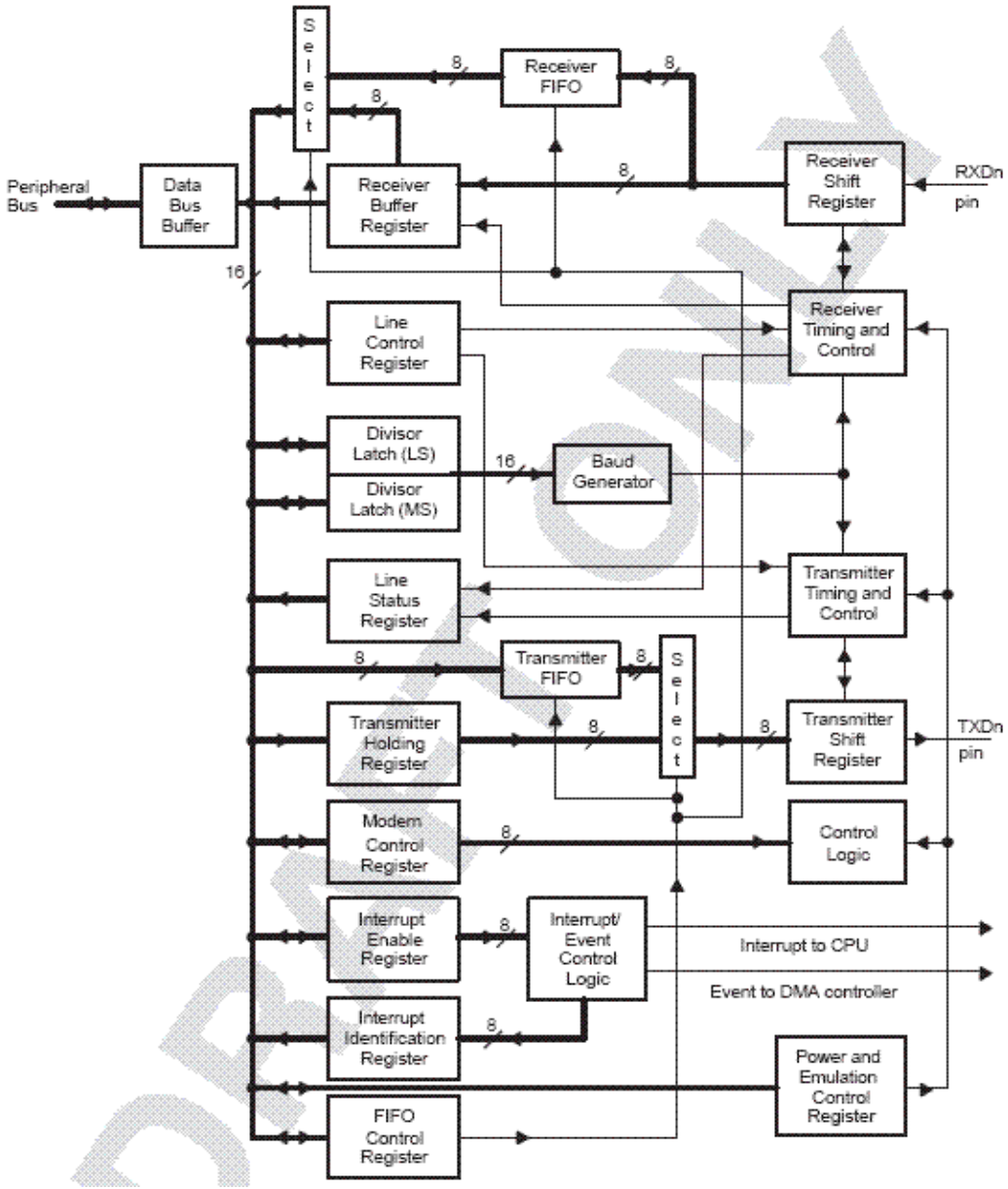


Figure 1 UART Block Diagram

1.4 Software

The UART driver discussed here is intended to run in SYS/BIOS™ on the DM814x

1.4.1 Operating Environment and dependencies

Details about the tools and SYS/BIOS versions that the driver is compatible with, can be found in the system Release Notes.

1.4.2 System Architechture

The block diagram below shows the overall system architecture.

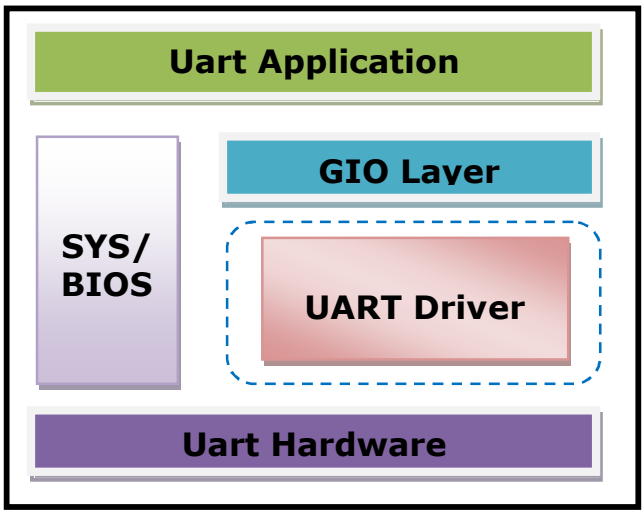


Figure 2 System Architecture

Driver module which this document discusses lies below the GIO layer, which is a class driver layer provided by SYS BIOS™. The uart 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 UART.

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 UART Device driver by TI.

1.5 Component Interfaces

In the following subsections, the interfaces implemented by each of the sub-component are specified. The uart driver module is an object of IOM Driver class one may need to refer the BIOS IOM Driver documentation to access the uart driver in raw mode.

1.5.1 IOM Driver Interface

The IOM Driver constitutes the Device Driver Manifest to application. This adapts the Driver to SYS/BIOS™. This uart driver is intended to inherit the IOM Driver interfaces. Thus the uart 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 UART 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 UART driver (because of the virtue of its IOM driver model compliance) exposes the following interfaces.

- Uart_init()
- Uart_IOMFXNS

The Uart_init() is a startup function that needs to be called by the user (application) to initialize the module state structure of the UART instance.

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

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

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

IOM Interfaces	Description
uartMdBindDev()	The uartMdBindDev() function is called by the SYS/BIOS™ after the bios initialization The uartMdBindDev() should typically perform the following actions. ❖ Acquire the device Handle for the specified instance of the UART on the SoC. ❖ Configure the UART device object with the specified parameters (or default parameters, if there is no external configuration. The default parameters shall be specified). ❖ Registers the UART driver ISR.
uartMdUnBindDev()	The uartMdUnBindDev() function is called to destroy an instance of the UART driver. ❖ It will unroll all the changes done during the bind operation and free all the resources allocated to the UART.
uartMdCreateChan()	The uartMdCreateChan() function is executed in response to the GIO_create() API call by the application. Application has to specify the mode in which the channel has to be created through the "mode" parameter. The UART driver supports only two modes of channel creation (input and output). ❖ It creates a communication channel in specified mode to communicate data between the application and UART device instance. ❖ It also configures UART channel and allocates required resources.
uartMdDeleteChan()	The uartMdDeleteChan() is invoked in response to the GIO_delete() API call by the application. ❖ It frees the channel and all the resources allocated during the creation of the channel.
uartMdSubmitChan()	Submit an I/O packet to a channel for processing. Used for data transfer operations like enqueue, dequeue and exchange operations.
uartMdControlChan()	Implements the IOCTLS for UART IOM mini driver. All control operations go through this interface.

- *UART driver does not have any default configuration support. Before using the driver, application should configure the driver with valid configurations during channel creation.*

1.5.2 CSLR Interface

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

1.6 Design Philosophy

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

1.6.1 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 UART device. The instance word usage refers to the driver as one entity. Any detail, configuration parameter or setting which shall apply across the driver shall be an instance variable. For example, loopback mode setting is an instance wide variable. Each hardware instance shall map to one UART instance. The "instance object" will be instantiated per physical instance of the peripheral. The lifetime of the instance is between its creation using uartMdBindDev() and its deletion using uartMdUnBindDev().

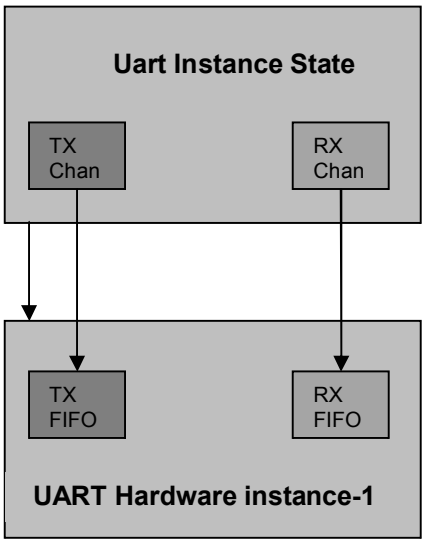


Figure 3 Instance Mapping

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

1.6.2 The Channel Concept

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

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

1.6.3 Design Constraints

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

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

2 UART Driver Software Architecture

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 UART IOM Driver module. The channel object is held inside the Instance State of the module. (This instance state is translated to the Uart_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 UART driver.

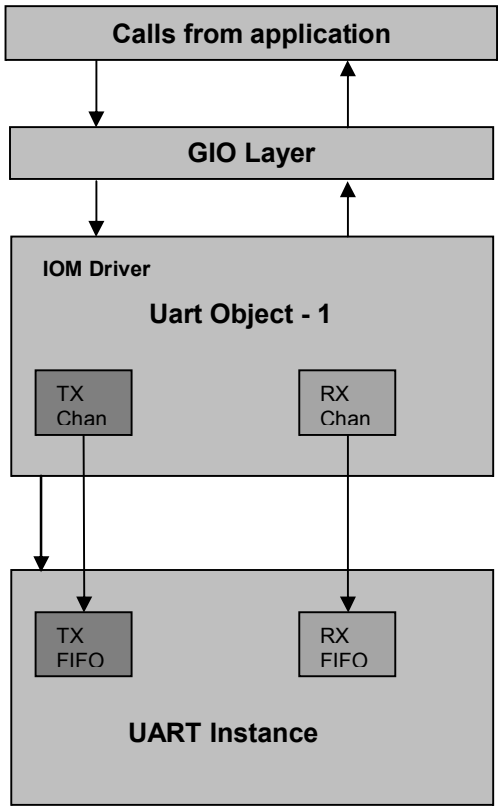


Figure 4 UART driver static view

2.1.2 Data Structures

The IOM Driver employs the Instance State (Uart_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. In non EDMA mode of operation this edmahandle can be NULL.

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

2.1.2.1 The Instance Object (Uart_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 are described in the section Device Parameters.

S.No	Structure Elements *(Uart_Object)*	Description
1	*instNum*	Preserve port or instance number of UART
2	*chunkSize*	Chunk size specifies the number of bytes written/read to/from the FIFO at a time
3	*opMode*	Preserve the mode of operation of the driver (Polled/Interrupt/EDMA)
4	*devParams*	Preserve device related configuration parameters like baudRate, looback enable etc
5	*devState*	Preserve the current state of the driver (Create/Deleted etc)
6	*xmt/rcvChanObj*	Transmit/Receive Channel objects for this instance, to hold channel status, io packets, queues, edmaHandle etc
7	*deviceInfo*	Device information for this instance like base register address, interrupt/event numbers etc
8	*hwiNumber*	Hardware interrupt Number
9	*enableCache*	Cache enabled/disabled
10	*stats*	Preserve the statistics like rx/tx error counts, rx/tx bytes etc
11	*txTskletHandle*	This refers to the tasklet (SWI) which is used to transmit bytes. This is posted from ISR and in the event of TX fifo empty. As we use to poll the Shift register empty bit before we write data into FIFO, we preferred to have a

		SWI rather to poll in ISR itself to reduce the latency of the interrupt
12	*rxTskletHandle*	This refers to the tasklet (SWI) which is used to receive bytes. This is posted from ISR and in the event of RX buffer not empty (bytes in fifo have crossed threshold). We use to poll the FIFO not empty bit continuously and read the data bytes from FIFO, before we come out from ISR. Please note that in case of continous reception of data through UART, there is a possibility for UART to hog the ISR. Hence, we preferred to have a SWI rather to poll in ISR itself to reduce the latency of the interrupt
13	*polledModeTimeout*	In polled mode time for polling to expire should be provided to avoid indefinite looping in the wait state. This variable holds the given timeout value
14	*syncSem*	sync semaphore object(used in the polled mode transfer for sync between multiple tasks IO submit requests
15	*uartConfigured*	Boolean to ensures that Uart is configured only once
16	*pwrmInfo*	Structure for holding the PWRM related info – applicable only if PWRM supported.
17	*hwiHandle*	Handle to the HWI created (only used in case of ARM)
18	*prcmHandle*	Handle to the prcm instance to be used to switch on the instance. Not being used in this release
19	*prcmDevId*	prcm device ID, which is being used instead of prcm handle
20	*crossBarEvtParam*	Cross bar event params /* Valid only for M3 core */

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) for that particular IO direction. 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. Per instance we have two channels, each for transmit and receive.

S.No	Structure Elements (Uart_ChanObj)	Description
1	*status*	Preserves the state of the driver
2	*mode*	Channel mode of operation: Input or Output.
3	*cbFxn and cbArg*	In case the driver is in any interrupt mode of operation or EDMA mode of operation then the application would be notified of any IO completion by this callback registered by the application.
4	*queuePendingList*	While the driver is busy transmitting/receiving or doing some processing on the i/o packets, there could be additional requests from the application which need to be accepted and stored for later processing inside the driver. This is the list that contains such pending requests/packets
5	*activeIOP*	To store the current IOP and indicate if there is any IOP being processed at the moment
6	*activeBuffer*	The current buffer into which the data is actually being received or being transferred

7	bytesRemaining	Bytes to be transferred or received in the current I/O operation
8	chunkSize	While transmitting the data we must ensure that in one operation one does not exceed the maximum write size limit, which is usually the transmit FIFO size. This limit is preserved here
9	devHandle	This is used as a back pointer to the instance object and initialized during open. This pointer is needed to access the hardware, since only here data like base addresses, event numbers etc exist.
10	errors	Maintains count of how many errors encountered per IOP request
11	hEdma	Handle to DMA object
12	edmaTcc	Transfer completetion code number
13	edmaChId	DMA Channel number
14	edmaTC	EDMA TC to be used
15	ioCompleted	Flag to check if IO completed
16	optFlag	Flag used to get paramSet register value in EDMA mode
17	optValue	Used to store opt value in PaRAM register
18	gblErrCbkFxn	Function registered to notify of error conditions
19	gblErrCbkArg	Callback argument to be used by the driver for the callback
20	abortAllIo	Flag to indicate that the packet abort is in progress

2.1.2.3 The Device Parameters (also known as Instance 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 DevParams structure and are explained below:

S.No	Structure Elements	Description
1	fifoEnable/rxThreshold	This member provides whether the internal hardware FIFO of the UART should be enabled. In that case the the data is received in this FIFO, and the receive ready interrupt is generated only after the rxThreshold is reached
2	loopbackEnabled	Bool to check if loop back enabled/disabled
3	baudRate	The baud rate setting
4	stopBits	Number of stop bits that should be used embedded in the data transfer
5	charLen	The number of characters to be transmitted per transmit
6	rxThreshold	Receive trigger level value in case of FIFO enable
7	txThreshold	Transmit trigger level value in case of FIFO enable
8	parity	Odd or Even parity to be used for error checking
9	fc	If flow control should be enabled. If so s/w flow control or hardware flow control
10	softTxFifoThreshold	Software TX threshold
11	PwrmEnable	Psc enabled/disabled.

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

S.No	Structure Elements	Description
1	hEdma	This contains the parameter to the EDMA handle as passed by the user.
2	crossBarEvtParam	Cross bar event params /* Valid only for M3 core*/

2.2 Dynamic View

2.2.1 The Execution Threads

The UART IOM Driver module involves following execution threads:

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

In UART, the application can perform IO operation using GIO_read/write() calls (corresponding IOM Driver function is uartMdSubmitChan()). The handle to the channel, buffer for data transfer, size of data transfer and timeout for transfer should be provided.

The UART module receives this information via uartMdSubmitChan(). Here some sanity checks on the driver shall be done like valid buffer pointers, mode of operation etc and actual read or write operation on the hardware shall be performed.

2.2.3 Functional Decomposition

The UART 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 design concept with flowcharts is explained in the sections to follow.

2.3 Driver API's and Flowcharts

2.3.1.1 Driver Creation (Driver Initialization and Binding)

The UART IOM driver needs the global data used by the UART driver to be initialized before the driver can be used. The initialization function for the UART driver is not included in the IOM_Fxns table, which is exported by the UART 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
- LPSC number for the module in the Power sleep controller.

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

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

.

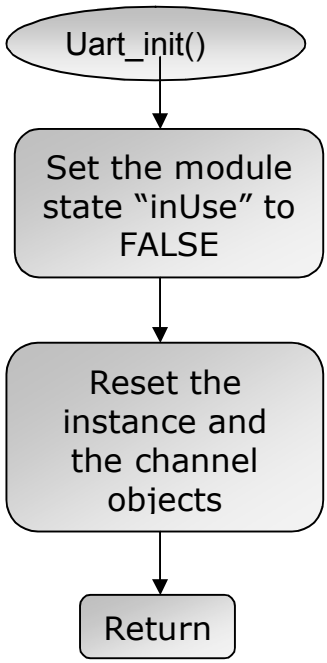


Figure 5 Uart_init() flow diagram

2.3.1.1.1 uartMdBindDev()

The binding function (uartMdBindDev) of the UART 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 UART. 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 UART device instance in the PSC module.
- Reset the UART device and disable the device.
- Configure the UART for the DLB mode if applicable and also to set the emulation mode settings.
- 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 “Uart_Params”. Refer to Uart_Params section for more details.

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

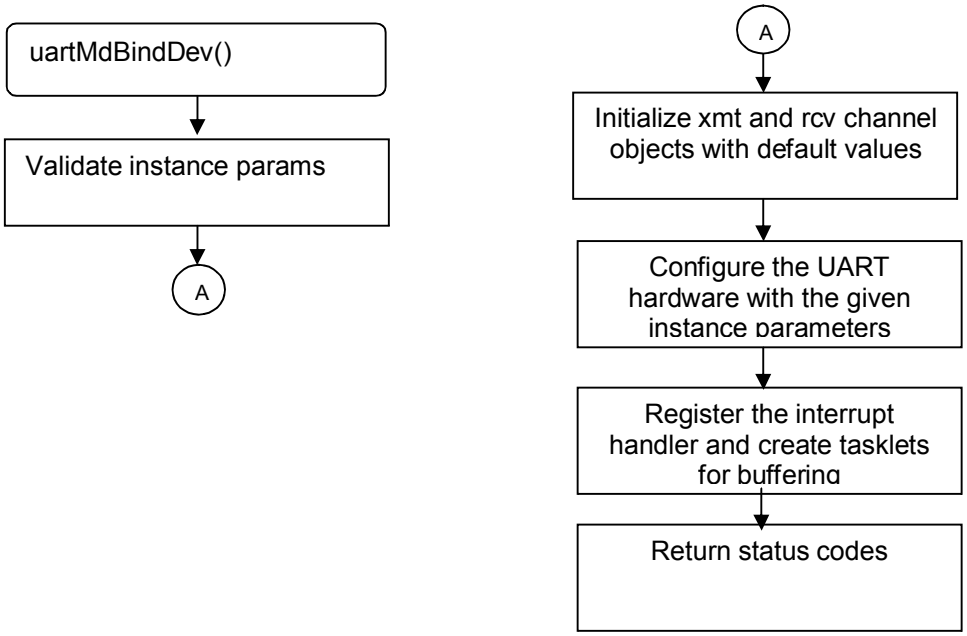


Figure 6 uartMdBindDev() flow diagram

2.3.1.2 Driver Deletion (Driver DeInitialization and UnBinding)

The instance MdUnBindDev function does a final clean up before the driver could be relinquished of any use. Here, all the resources which were allocated during instance MdBindDev shall be unallocated. 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 PSC.
- Unregister the notification registered with the BIOS power management module(only if the driver supports power management)

2.3.1.2.1 uartMdUnBindDev()

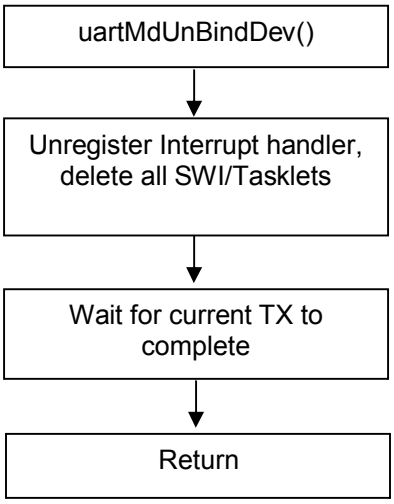


Figure 7 uartMdUnBindDev() flow diagram

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 UART. 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 UART IO mini driver's uartMdCreateChan() function. The application shall call GIO_create() with the appropriate "mode" (IOM_INPUT or IOM_OUTPUT) parameter for the type of the channel to be created (Reception or transmission).

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

The uartMdCreateChan() 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 UART 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 uartMdCreateChan()

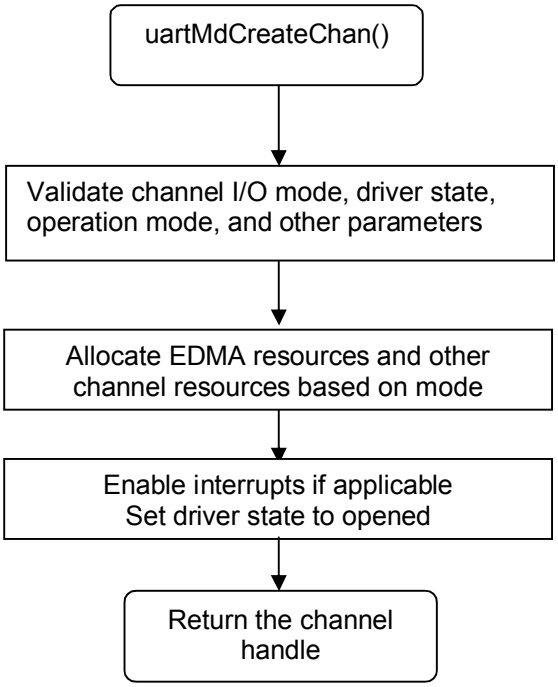


Figure 8 uartMdCreateChan() flow diagram

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 Uart driver provides the “uartMdDeleteChan” API to delete a previously created UART 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 uartMdDeleteChan()

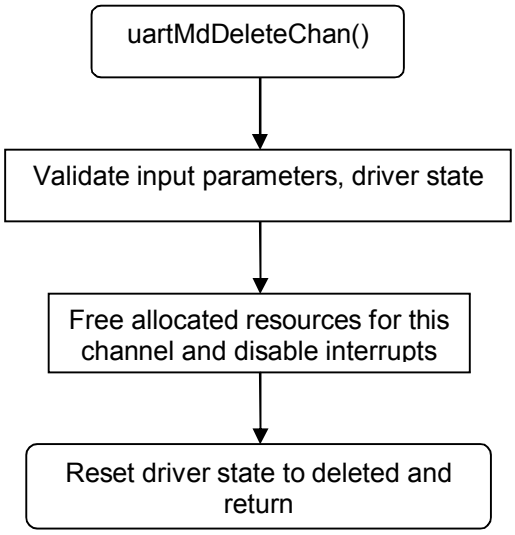


Figure 9 uartMdDeleteChan() flow diagram

2.3.1.5 IO submit

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

- IOM_READ. Drivers that support input channel must implement IOM_READ.
- IOM_WRITE. Drivers that support output channel must implement IOM_WRITE.

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 mdSubmitChan function of the UART driver typically performs the following activities:

- The input packet is validated.

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

2.3.1.5.1 uartMdSubmitChan()

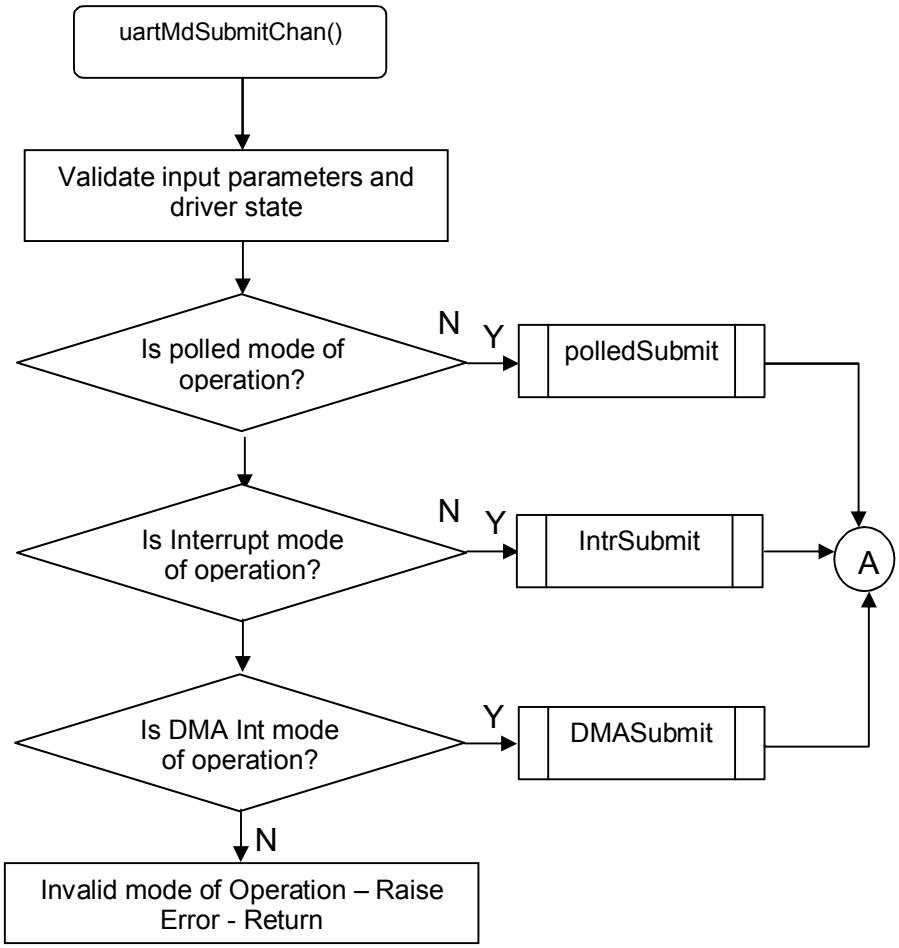
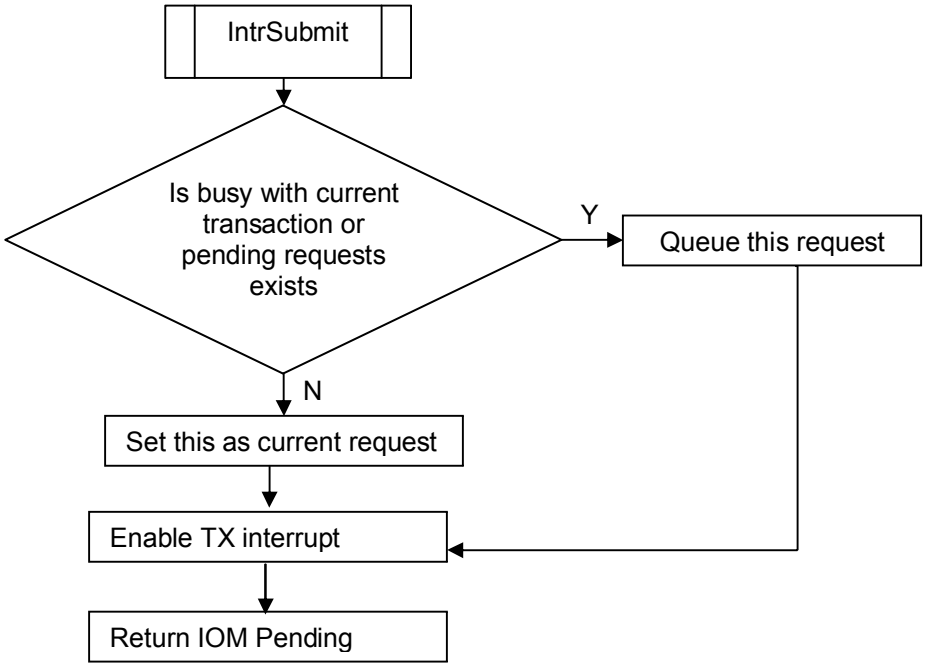
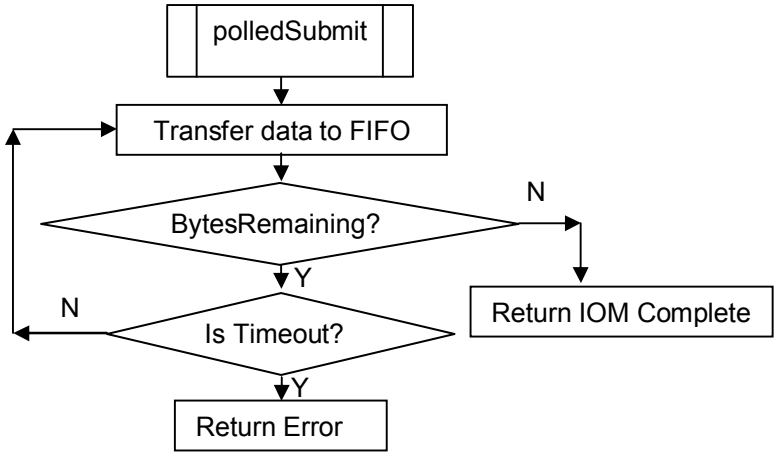
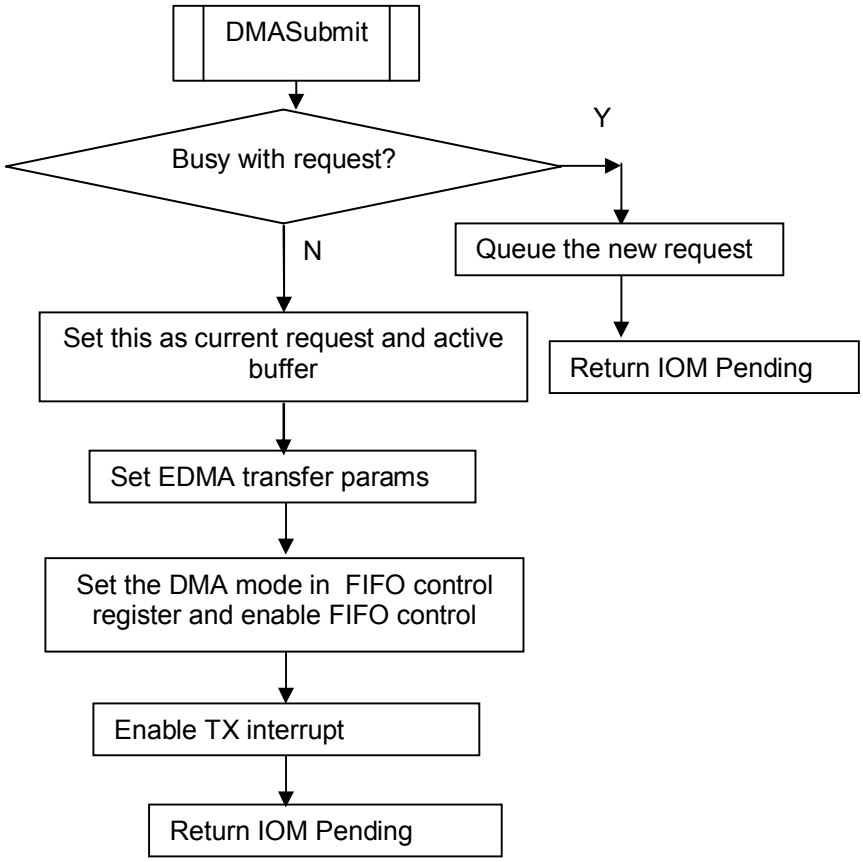


Figure 10 uartMdSubmitChan() flow diagram


IntrSubmit
Is busy with current transaction or pending requests exists
Y
Queue this request
N
Set this as current request
Enable TX interrupt
Return IOM Pending


polledSubmit
Transfer data to FIFO
BytesRemaining?
N
Y
Return IOM Complete
N
Is Timeout?
Y
Return Error



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 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).

2.3.1.6 Control Commands

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

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

- Validate the command sent by the application.
- Check if the appropriate arguments are provided by the application for the execution of the command.
- Process the command and return the status back to the application.
- The APPENDIX A table lists the control commands supported by the UART driver

2.3.1.6.1 uartMdControlChan()

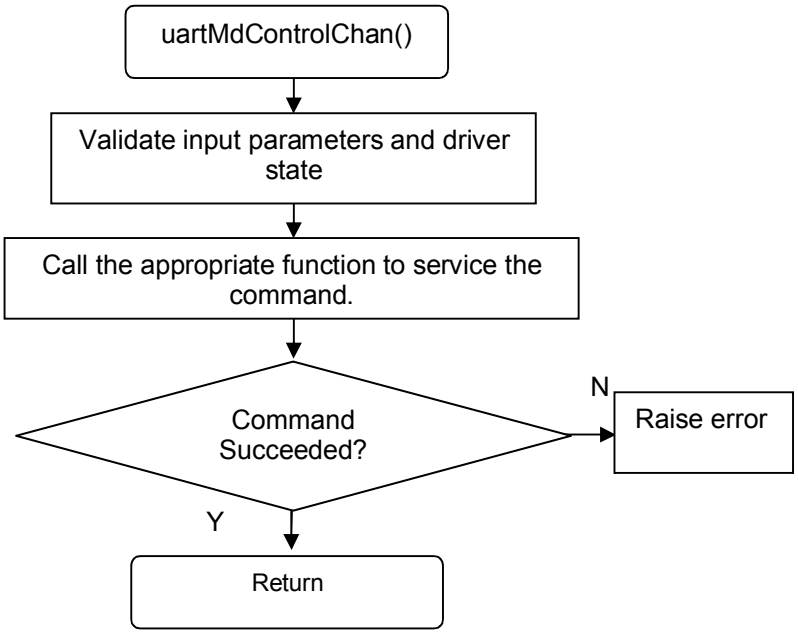


Figure 11 uartMdControlChan() flow diagram

2.3.1.7 uartIsr()

Apart from the line errors handled, the transmit and receive operations are handled. However, for transmit and receive operations, bottom halves (tasklet/software interrupts) is posted from the ISR to reduce the latency of the interrupt handler.

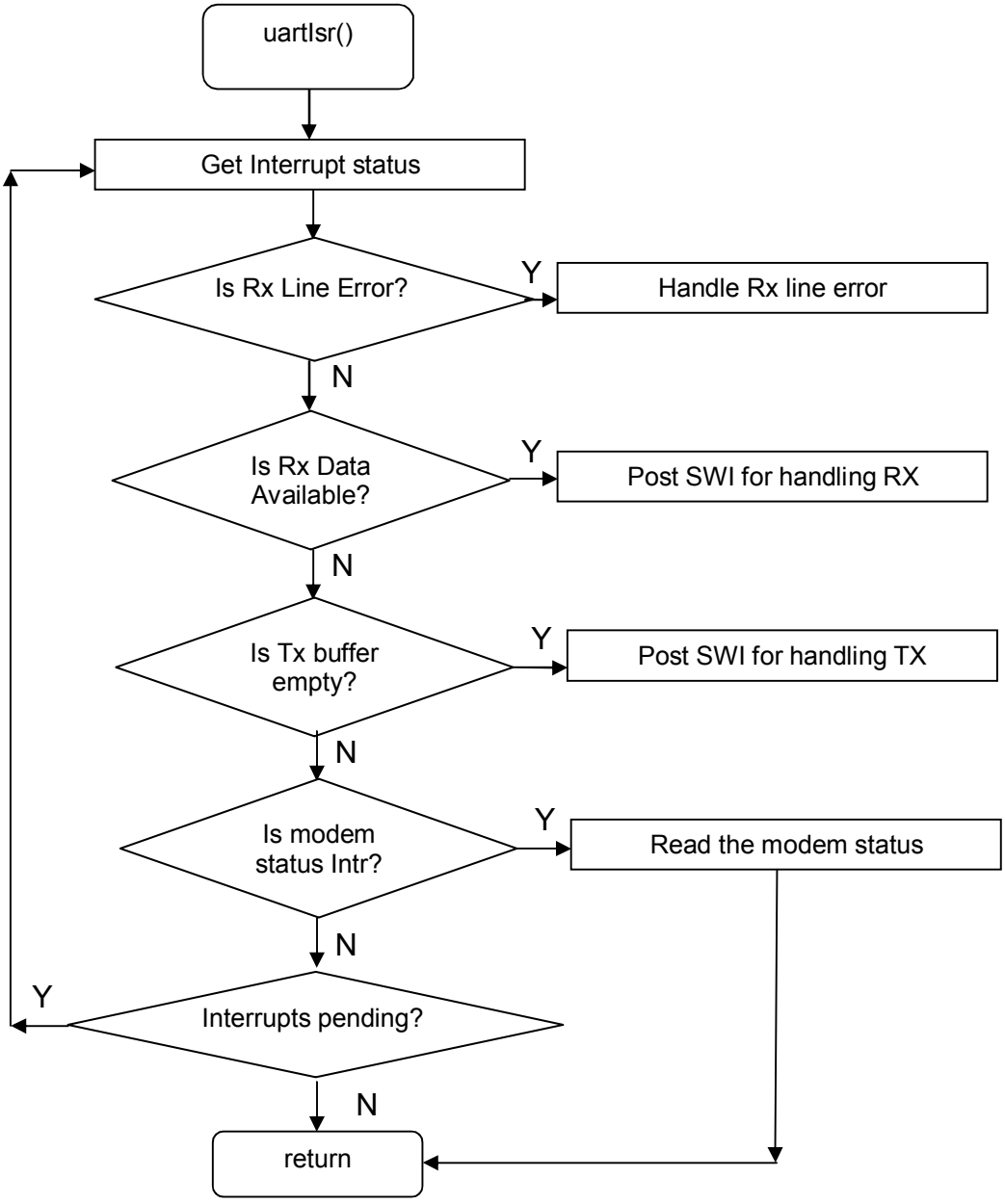
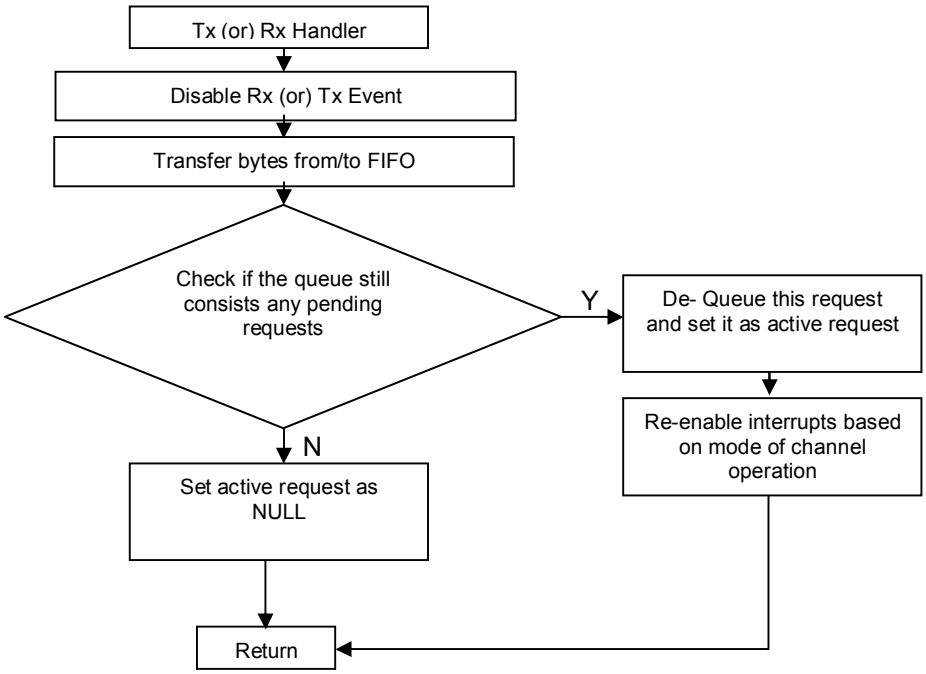


Figure 12 uartIsr() flow diagram

The ISR apart from processing the errors, also handles the transfer complete (Tx Buffer empty) interrupt and the receive data interrupt (Rx buffer full). These operations consume many CPU cycles, and hence should not be done in the ISR context, in which case disables all lower interrupts

and hence increases interrupt latency. This is more apparent when there is heavy receive. Thus the SWIs are posted, which are actually bottom halves to process the data to be transferred or received. These are called when there are no hardware interrupts to be services but definitely before running any pending tasks. These SWIs are created in during instantiation sequence as described Module_startup and stored in the channel objects.



2.3.1.8 Uart_localIsrEdma()

When UART is operating in the DMA interrupt mode for data transfer, the UART driver during the channel open requests EDMA channels and registers a callback (Uart_localIsrEdma())for notification of the transfer completion status. Before starting any transfer, UART 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 UART driver to take act upon the status. This is done in the Uart_localIsrEdma() as shown below.

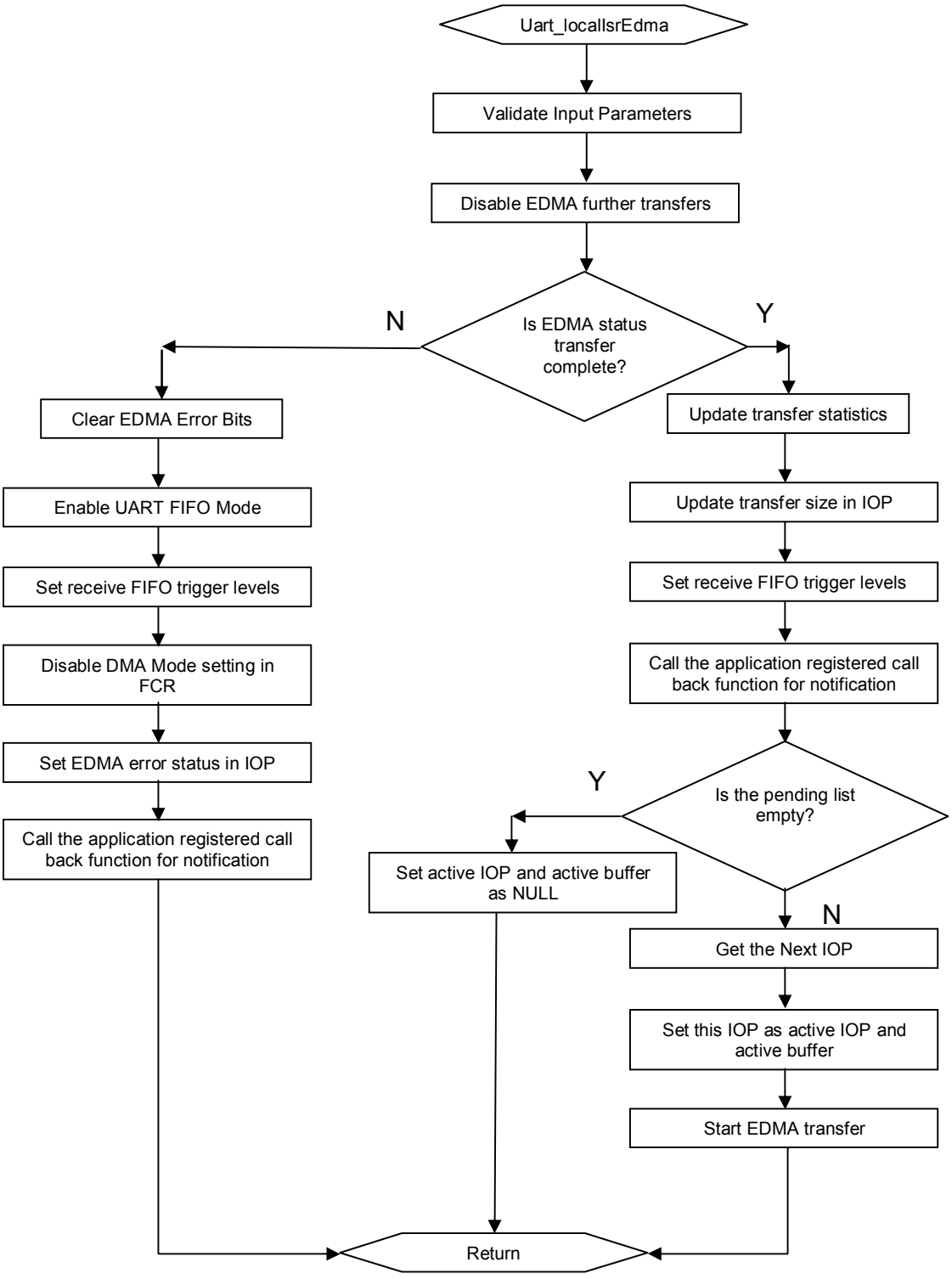


Figure 13 Uart_localIsrEdma() flow diagram

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	Uart_IOCTL_SET_BAUD	Set the Baud rate.
2	UART_IOCTL_SET_STOPBITS	Set number of stop bits.
3	UART_IOCTL_SET_DATABITS	Set number of data bits.
4	UART_IOCTL_SET_PARITY	Set parity Odd/Even
5	UART_IOCTL_SET_FLOWCONTROL	Set flow control HW or SW
6	UART_IOCTL_SET_TRIGGER_LEVEL	Set trigger level for FIFO
7	UART_IOCTL_RESET_RX_FIFO	Clear RXFIFO
8	UART_IOCTL_RESET_TX_FIFO	Clear TXFIFO
9	UART_IOCTL_CANCEL_CURRENT_IO	Cancel the current IO
10	UART_IOCTL_GET_STATS	Get statistics information
11	UART_IOCTL_CLEAR_STATS	Clear statistics information
12	Uart_IOCTL_FLUSH_ALL_REQUEST	Cancel the current and all the pending requests
13	Uart_IOCTL_SET_POLLEDMODETIMEOUT	Set the i/o operation timeout value for polled mode