DSP/BIOS UART Device Driver

Architecture/Design Document

Revision History

Document Version	Author(s)	Date	Comments
0.1	Madhvapathi Sriram	September 8 2008	Created the document
0.2	Vichu	October 6, 2008	Updated the document
0.3	Imtiaz SMA	January 21, 2009	Updated the sections 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

Table of Contents

List Of Figures

1 System Context

The purpose of this document is to explain the device driver design for UART peripheral using DSP/BIOS operating system running on DSP OMAPL138

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

1.1 Terms and Abbreviations

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

1.1 Disclaimer

This is a design document for the UART driver for the DSP/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 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 UART 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_C6747). Based on the include a chip specific header file (e.g soc_C6747) 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
2.	SPRUFM6	UART user guide (draft)

1.4 Hardware

The UART device driver design is in the context of DSP/BIOS running on DSP OMAPL138 /C64P core.

The UART module core used here has the following bocks:

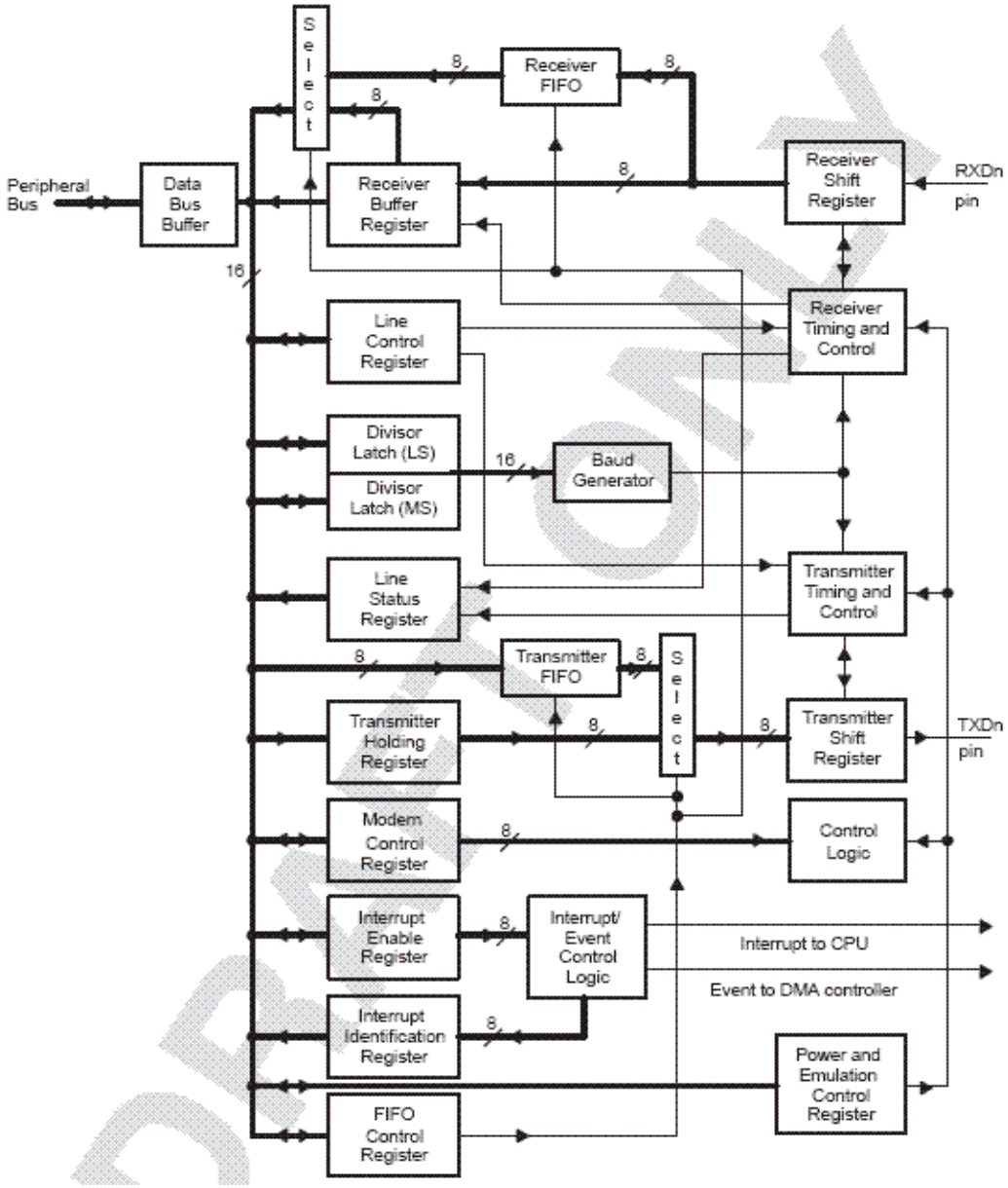


Figure 1 UART Block Diagram

1.5 Software

The UART driver discussed here is intended to run in DSP/BIOS™ V6.10 on the OMAPL138 DSP.

1.5.1 Operating Environment and dependencies

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

1.5.2 System Architechture

The block diagram below shows the overall system architecture.

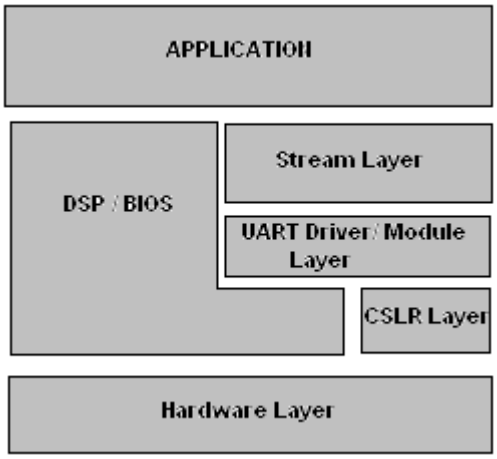


Figure 2 System Architecture

Driver module which this document discusses lies below the steam layer, which is an class driver layer provided by DSP BIOS™ (please note that the stream layer is designed to be OS independent with appropriate abstractions).The uart 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 uart driver module is object of IDriver class one may need to refer the IDriver documentation to access the uart 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 uart driver is intended to an XDC module and this module would inherit the IDriver interfaces. Thus the uart driver module becomes an object of IDriver class. Please note that the terms “Module” and “Driver or IDriver” would be used in this document interchangeably.

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

UART module specifications file (Uart.xdc)

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

These definitions would include

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

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

CONSTANTS: error ids and ioctls

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

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

The XDC framework translates this into the driver header file (Uart.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.

UART module script file (Uart.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 UART module/driver. The Uart.xs file contains two parts

1. Handling the module use references

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

2. Handling static instantiation of the UART instance

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

The UART IDriver module implements the following interfaces

S.No	IOM Interfaces	Description
1	Uart_Module_startup()	Register interrupts, configure hardware and initialization needed before opening a channel. Effectively makes the UART module ready for use. This function is called at least once for sure when the UART IDriver module is used. Hence these tasks are done here for all the instances that can be created.
2	Uart_Instance_init()	Handle dynamic calls to module instantiation. This shall be a duplication of the tasks done in instance static init in the module script file.

3	Uart_Instance_finalize()	Unregister interrupt, reset hardware and driver state and all deinitialization foes here. Effectively removes the usage of UART instance.
4	Uart_open ()	Creates a communication channel in specified mode to communicate data between the application and the UART module instance.
5	Uart_close ()	Frees a channel and all its associated resources.
6	Uart_control ()	Implements the IOCTLS for UART IDriver module. All control operations go through this interface.
7	Uart_submit ()	Submit an I/O packet to a channel for processing. Used for data transfer operations. Internally handles different mode of operation and asynchronous mode of communication.

1.6.2 CSLR Interface

The CSL register interface (CSLR) provides register level implementations. CSLR is used by the UART IDriver module to configure UART 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 UART hardware.

1.7.1 The Module and Instance Concept

The IDriver model, conforming to the XDC framework, provides the concept of the *module and instance* for the realization of the device and its communication path as a part of the driver implementation.

The module word usage (UART module) refers to the driver as one entity. Any detail, configuration parameter or setting which shall apply across the driver shall and thus shall configure the module behavior, shall be a module variable. However, there can also be module wide constants. For example, mode of operation (interrupt/polled/EDMA) setting is a module wide variable and can be set by the application.

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

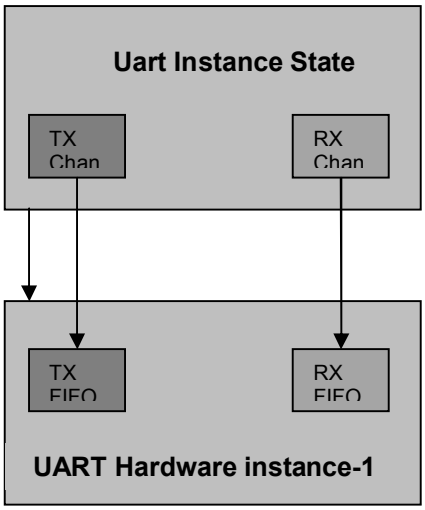


Figure 3 Instance Mapping

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

1.7.2 Design Constraints

UART IDriver 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

This section details the data structures used in the UART 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 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 UART IDriver 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.

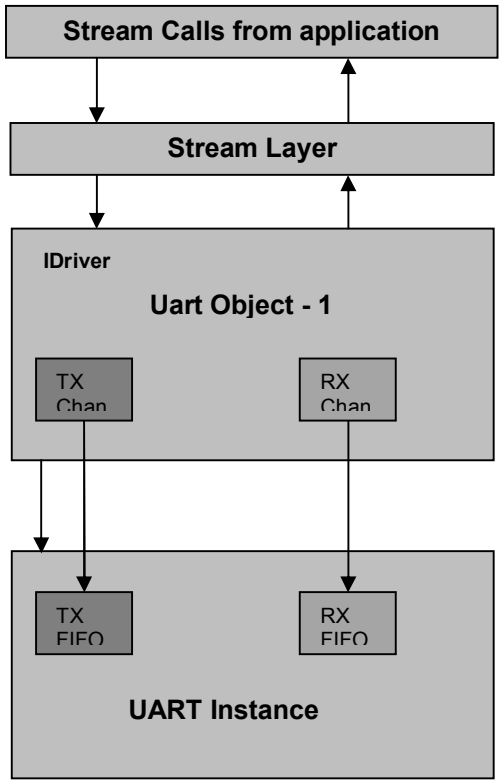


Figure 4 UART driver static view

2.1.2 Data Structures

The IDriver 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 IDriver 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 is described in the section Device Parameters.

S.No	Structure Elements *(Uart_Object)*	Description
1	*instNum*	Preserve port or instance number of UART
2	*opMode*	Preserve the mode of operation of the driver (Polled/Interrupt/EDMA)
3	*devParams*	Preserve device related configuration parameters like baudRate, looback enable etc
4	*DevState*	Preserve the current state of the driver (Create/Deleted etc)
5	*xmt/rcvChanObj*	Transmit/Receive Channel objects for this instance, to

		hold channel status, io packets, queues, edmaHandle etc
6	*deviceInfo*	Device information for this instance like base register address, interrupt/event numbers etc
7	*stats*	Preserve the statistics like rx/tx error counts, rx/tx bytes etc
8	*Xmt tskLetHandle*	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
9	*Rcv tskLetHandle*	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
9	*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

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	Iscompleted	We had a critical issue in EDMA mode. EVENT happened immediately after the completion and it preempted the completion isr. To handle this condition we had this variable to use as flag

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. For further information of these methods please refer xdc documentation. These parameters are preserved in the DevParams structure and are explained below:

S.No	Structure Elements	Description
1	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	baudRate	The baud rate setting
3	stopBits	Number of stop bits that should be used embedded in the data transfer
4	charLen	The number of characters to be transmitted per transmit
5	parity	Odd or Even parity to be used for error checking
6	fc	If flow control should be enabled. If so s/w flow control or hardware flow control

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.

2.2 Dynamic View

2.2.1 The Execution Threads

The UART IDriver 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 Stream_read/write() calls (corresponding IDriver function is Uart_Submit()). 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 Uart_submit. 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 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(for more than one SOC), 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.

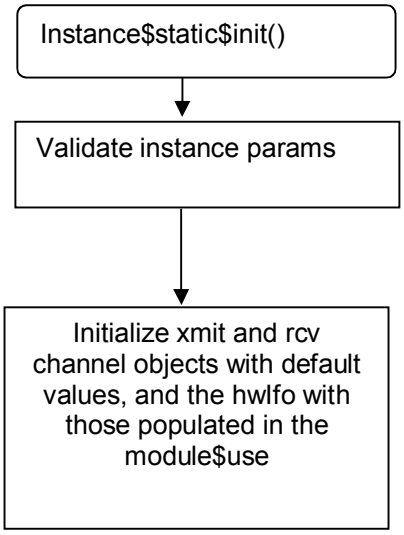


Figure 5 instance$static$init() flow diagram

2.2.3.3 Uart_Module_startup() of C file of driver Module

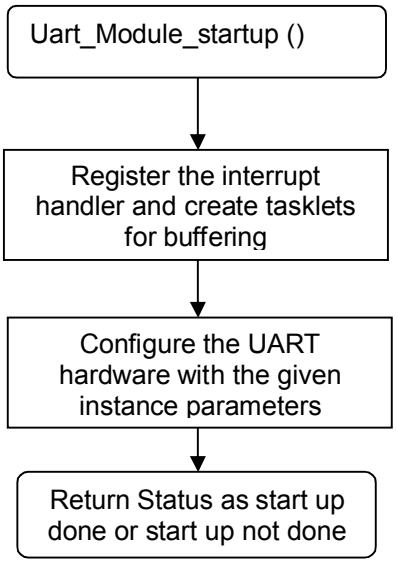


Figure 6 Uart_Module_startup() flow diagram

The module start up function is called as part of startup functions only once, during static instantiation, irrespective of the number of instances this module supports, during the module startup by the BIOS/RTSC framework. This function is provided to make ready the module to start execution. In this function the data structures/resources needed for all instances are prepared. Here, initialization that cannot be performed in the module script file, as part of static (compile-time for ex., interrupt registration) initialization, should be performed.

This is the place where the C functions for initialization should be called

For the UART module, for each instance:

1. Configure the UART hardware
    a. Bring module out of reset, disable interrupts, configure Tx and Rx parameters etc
2. Register interrupt handler, create the SWI interrupts required

In this function context the instance and the module variables shall not be available. In order to get these we shall use XDC APIs for getting these variables.

Hence the initialization for static instantiation consists of two parts, the script file (instance$static$init) and the C initialization (Uart_Module_startup).

2.2.3.4 Uart_Instance_init()

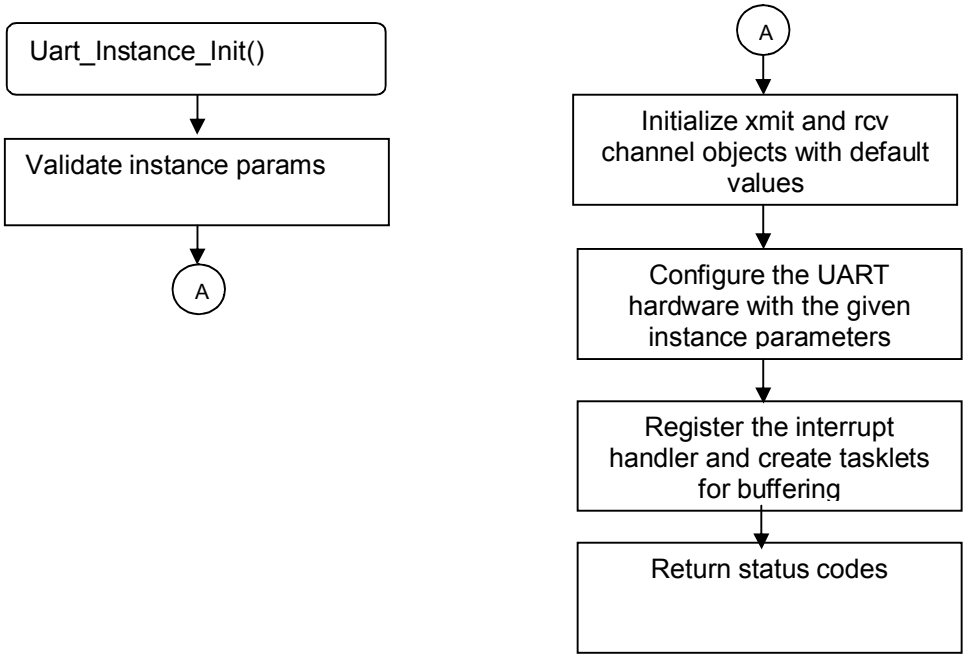


Figure 7 Uart_Instance_Init() flow diagram

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.

2.2.3.5 Uart_Instance_finalize()

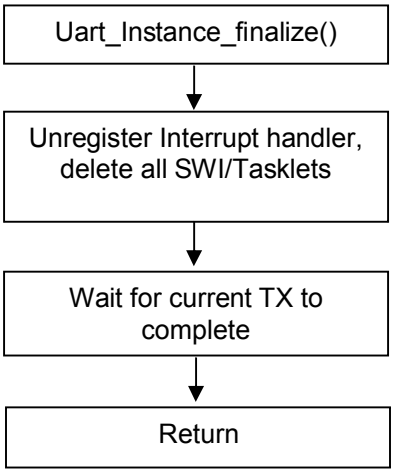


Figure 8 Uart_Instance_finalize () flow diagram

The Uart_Instance_init and Uart_Instance_finalize functions are called when by the XDC/BIOS framework.

The instance init function does a start up initialization for the driver. This function is called when the module is instantiated dynamically by the Uart_create call by the application. At this module instantiation, the interrupt registration, tasklets creation and all other tasks (including any allocation of resource needed later by the driver) 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, interrupt handlers shall be unregistered and all tasklets created shall be deleted. 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.

2.2.3.6 Uart_open()

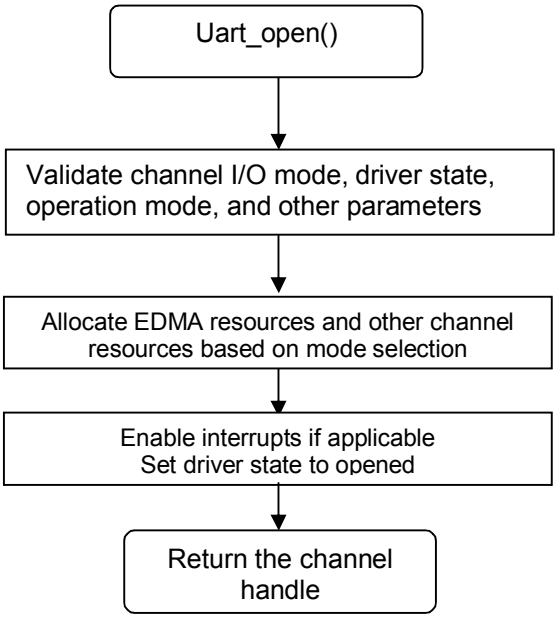


Figure 9 Uart_open () flow diagram

2.2.3.7 Uart_close()

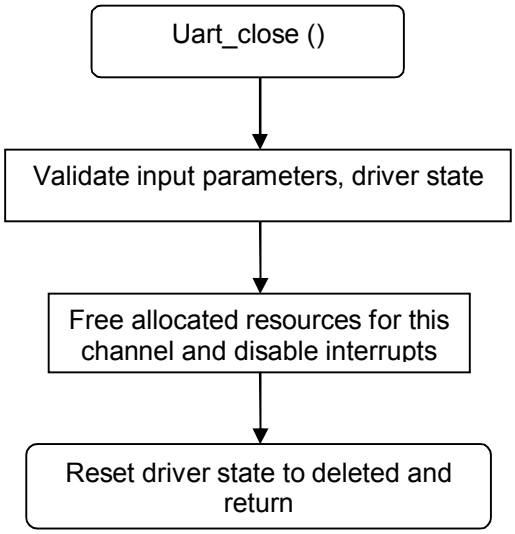


Figure 10 Uart_close() flow diagram

2.2.3.8 Uart_control()

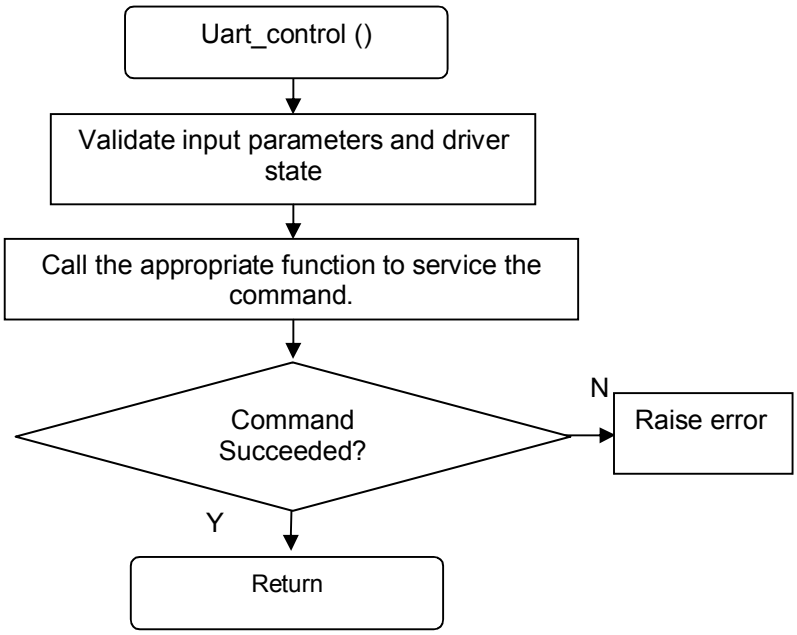


Figure 11 Uart_control () flow diagram

2.2.3.9 Uart_submit()

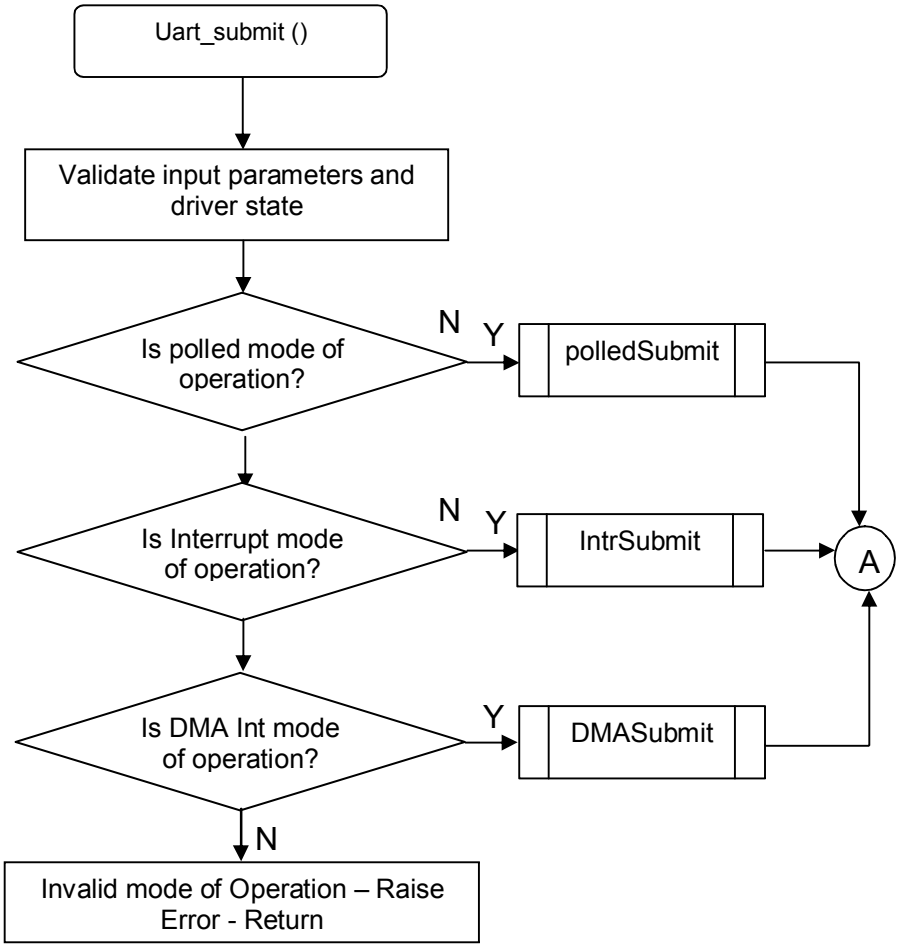
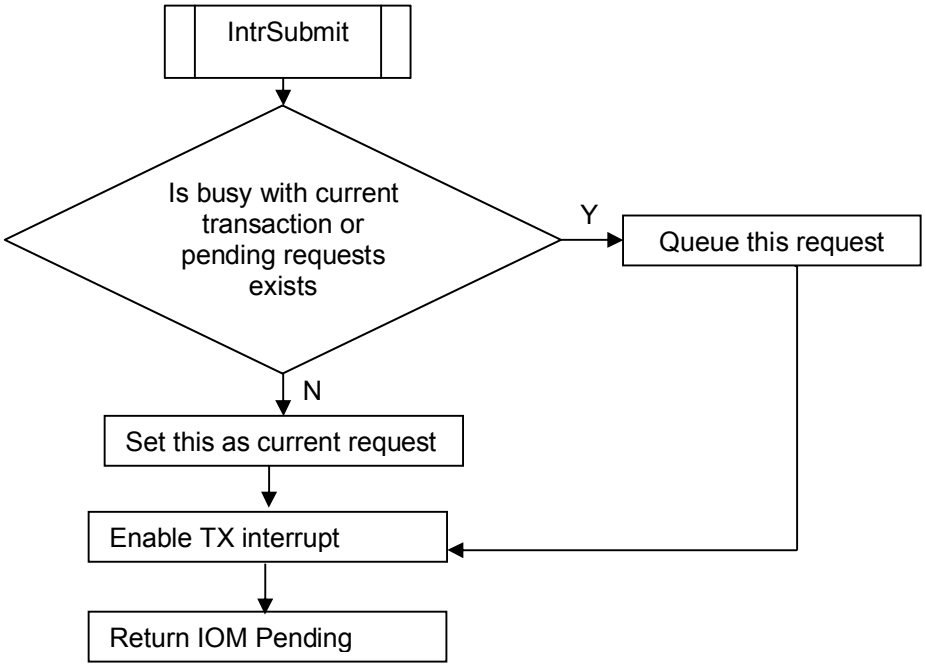
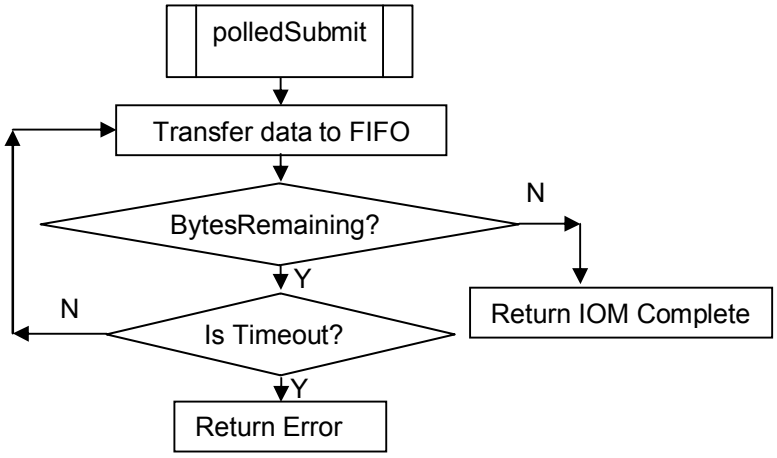
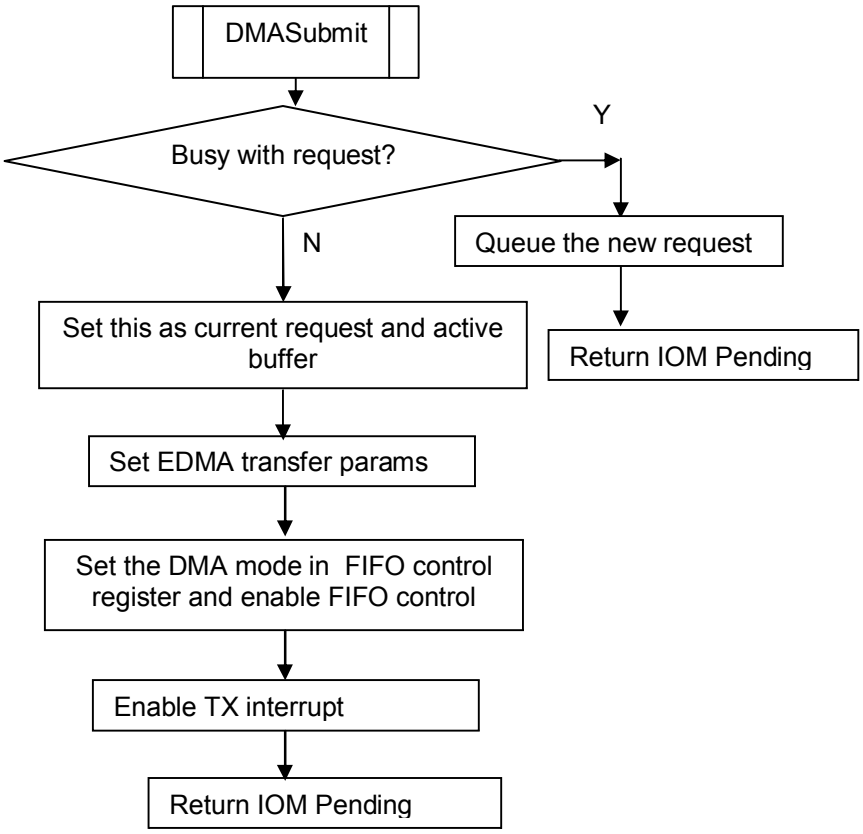


Figure 12 Uart_submit() flow diagram


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 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).

2.2.3.10 uartIsr()

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

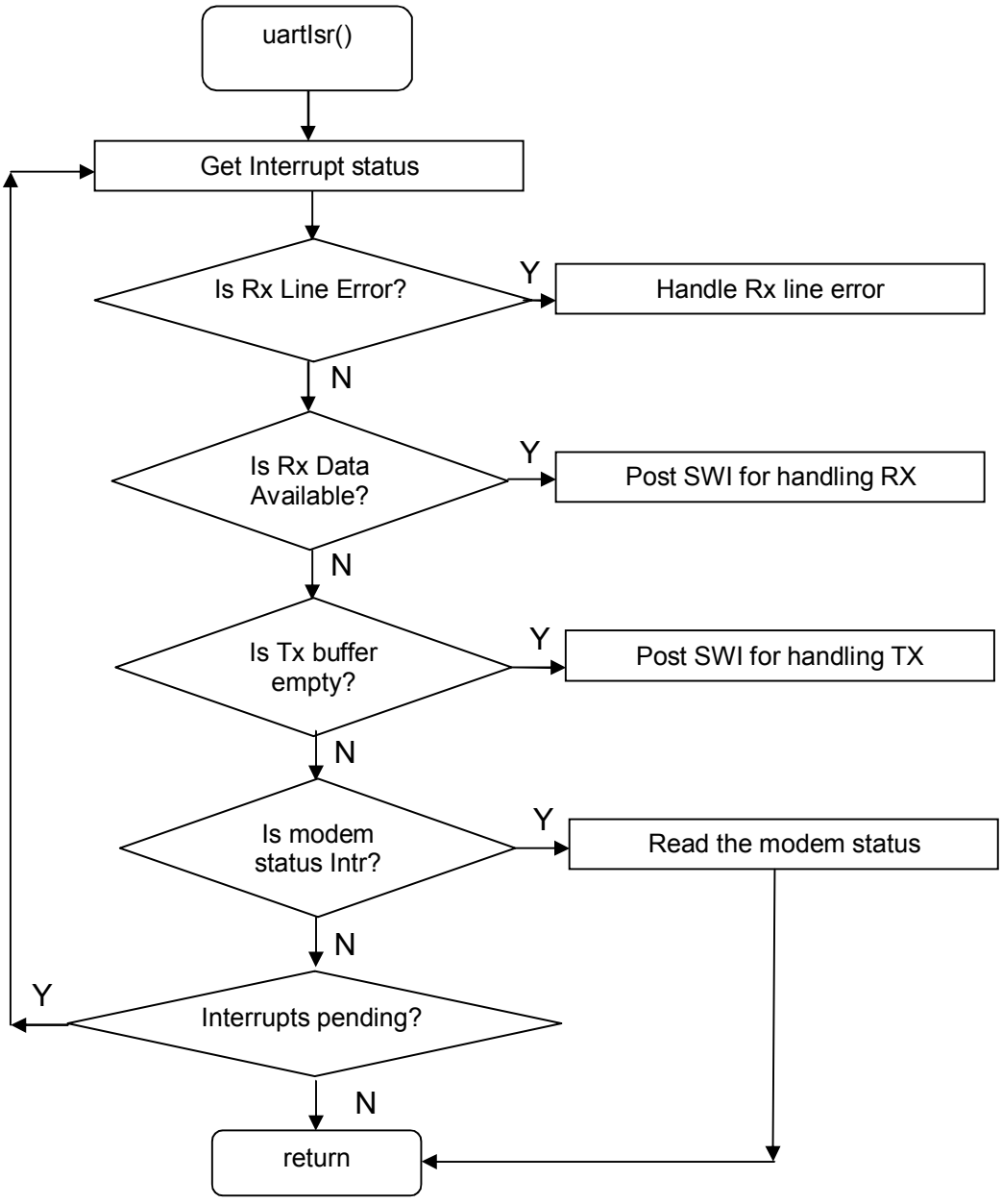
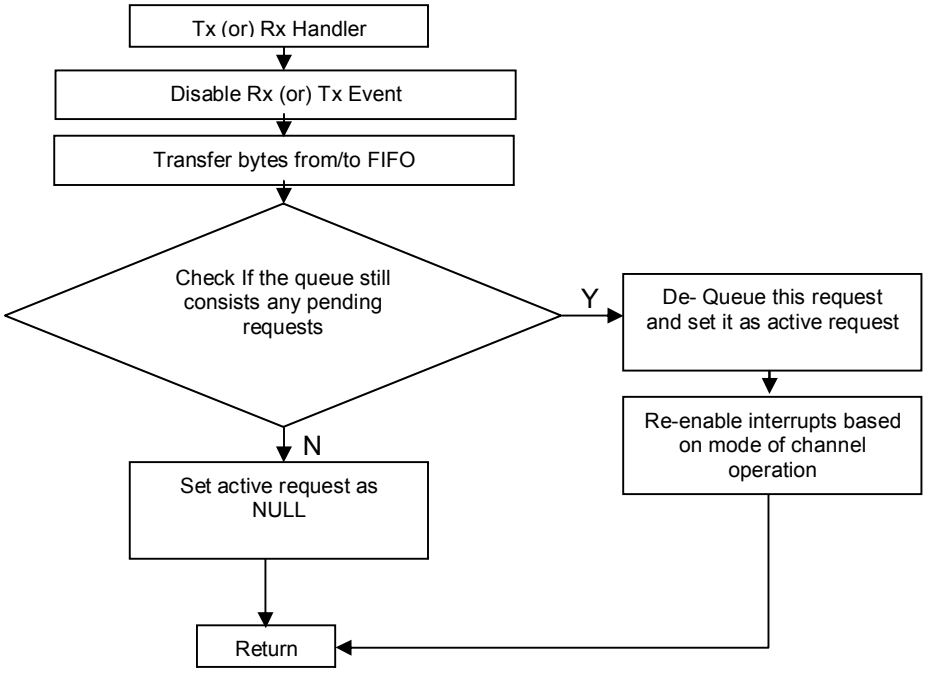


Figure 13 uartIsr flow diagram

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.



2.2.3.11 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.

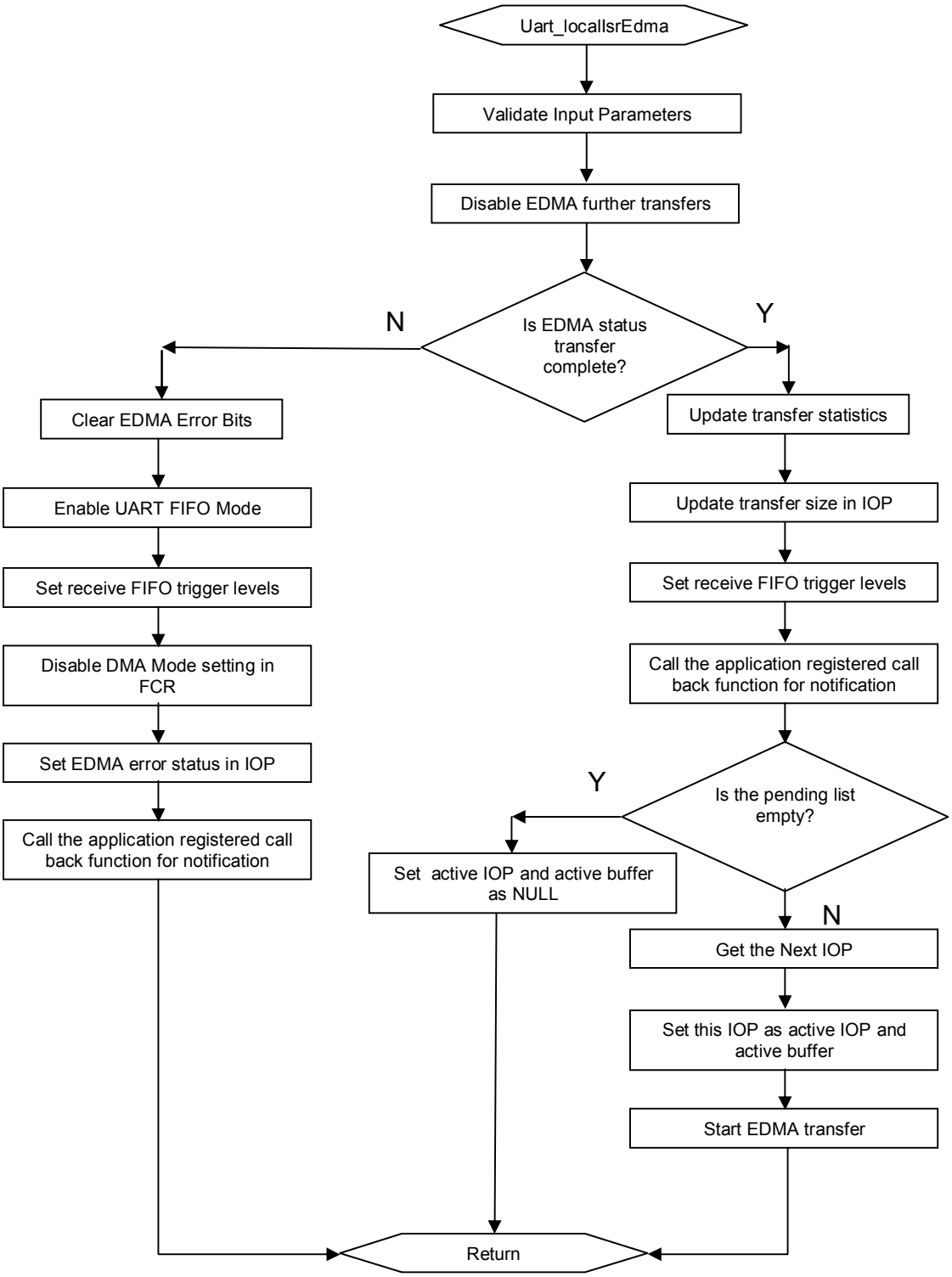

Uart_localIsrEdma
Validate Input Parameters
Disable EDMA further transfers
Is EDMA status transfer complete?
N
Y
Clear EDMA Error Bits
Enable UART FIFO Mode
Set receive FIFO trigger levels
Disable DMA Mode setting in FCR
Set EDMA error status in IOP
Call the application registered call back function for notification
Update transfer statistics
Update transfer size in IOP
Set receive FIFO trigger levels
Call the application registered call back function for notification
Is the pending list empty?
Y
N
Set active IOP and active buffer as NULL
Get the Next IOP
Set this IOP as active IOP and active buffer
Start EDMA transfer
Return

Figure 14 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