20450 Century Boulevard
Germantown, MD 20874

MCBSP LLD

Software Design Specification (SDS)

Revision A

Revision Record	
Document Title: **Software Design Specification**	
Revision	Description of Change
A	1. Initial Release – Code drop 1.0.0.0

Note: Be sure the Revision of this document matches the Approval record Revision letter. The revision letter increments only upon approval via the Quality Record System.

TABLE OF CONTENTS

LIST OF FIGURES

1 Scope

This document describes the design of Multichannel Buffered Serial Port Low Level Driver (MCBSP LLD). Also, the data types, data structures and application programming interfaces (APIs) provided by the MCBSP driver are explained in this document.

2 References

The following references are related to the feature described in this document and shall be consulted as necessary.

No	Referenced Document	Control Number	Description
1	MCBSP User Guide	SPRUHH0	KeyStone Architecture MCBSP User Guide
2	MCBSP LLD Documentation		The MCBSP LLD APIs are generated by DOXYGEN and is located in the MCBSP package under the "`docs`" directory in CHM format.
3	EDMA User Guide	SPRUGS5A	Enhanced Direct Memory Access (EDMA3) Controller User Guide

Table 1. Referenced Materials

3 Definitions

Acronym	Description
API	Application Programming Interface
CSL	Chip Support Library
CPU	Central Processing Unit
DMA	Direct Memory Access
DSP	Digital Signal Processor
EDMA	Enhanced Direct Memory Access Controller
FIFO	First In First Out
IP	Intellectual Property
ISR	Interrupt Service Routine
LLD	Low Level Driver
MCBSP	Multichannel Buffered Serial Port

Acronym	Description
MMR	Memory Mapped Register
OSAL	Operating System Abstraction Layer
PARAM	Parameter RAM
SOC	System On Chip
SRGR	Sample Rate Generator

Table 2. Definitions

4 Overview

The multichannel buffered serial port (MCBSP) peripheral allows direct interface to other TI DSPs, codecs, and other devices in a system. The primary use for the MCBSP is for audio interface purposes. The following sub sections explain the hardware (MCBSP peripheral) and software context of the MCBSP LLD.

4.1 Hardware Overview

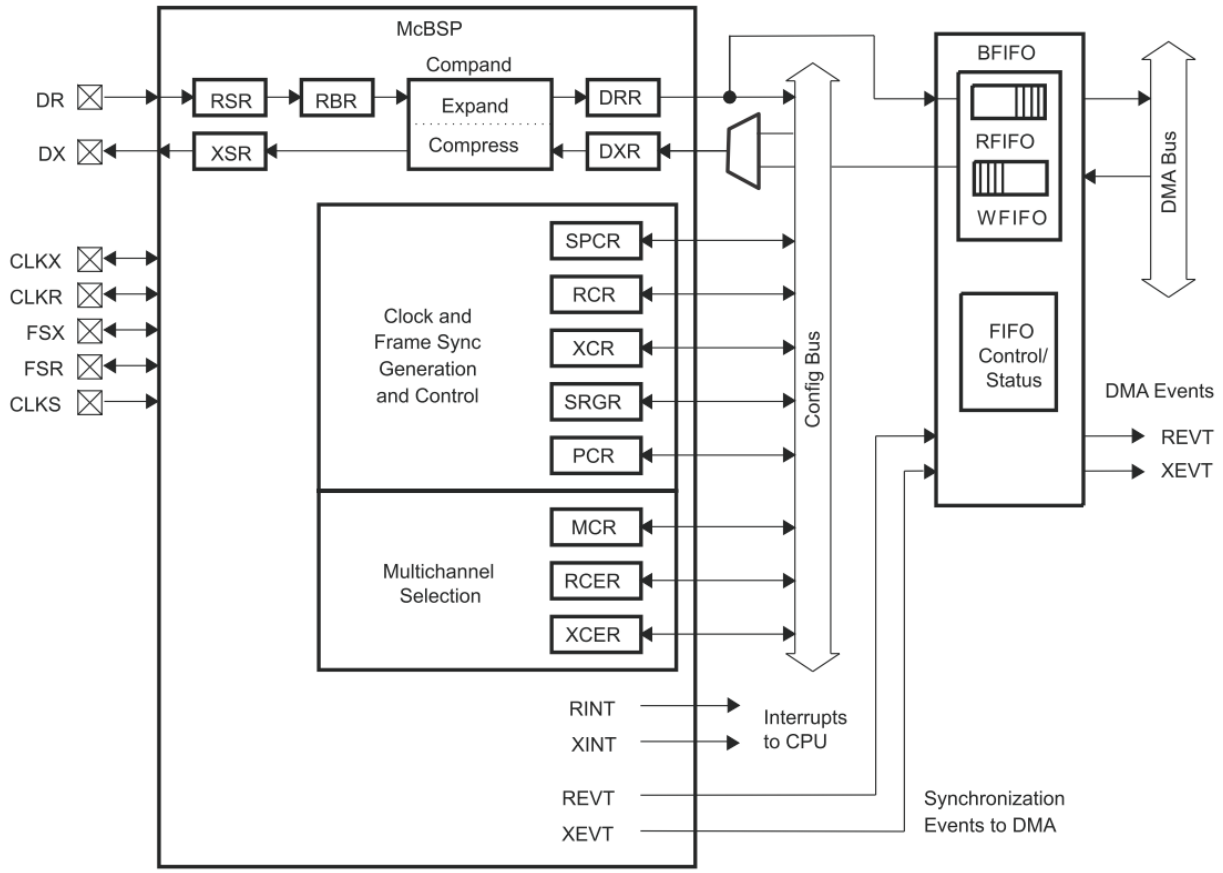

Figure 1: MCBSP Hardware Block Diagram

The Figure 1: MCBSP Hardware Block Diagram above shows the hardware overview of the MCBSP controller. The peripheral contains a main controller, a FIFO interface and also the EDMA controller interface. The MCBSP controller provides the hardware registers that allows the MCBSP to be configured for the serial data transfer. The MCBSP Buffer FIFO (BFIFO) provides additional data buffering for the MCBSP. The time it takes the CPU or DMA controller to respond to DMA requests from the MCBSP may vary. The additional buffering provided by the BFIFO allows greater tolerance to such variations.

The EDMA controller interface allows the EDMA to be programmed to move the serial data between the MCBSP and the DSP. There are dedicated EDMA channels available for the MCBSP to transfer and receive data. (The software also uses two additional spare EDMA PARAM sets for PING PONG operation for providing additional buffering required especially when transferring audio data as the tolerance to delays is very less during an audio data transfer).

4.2 Software Overview

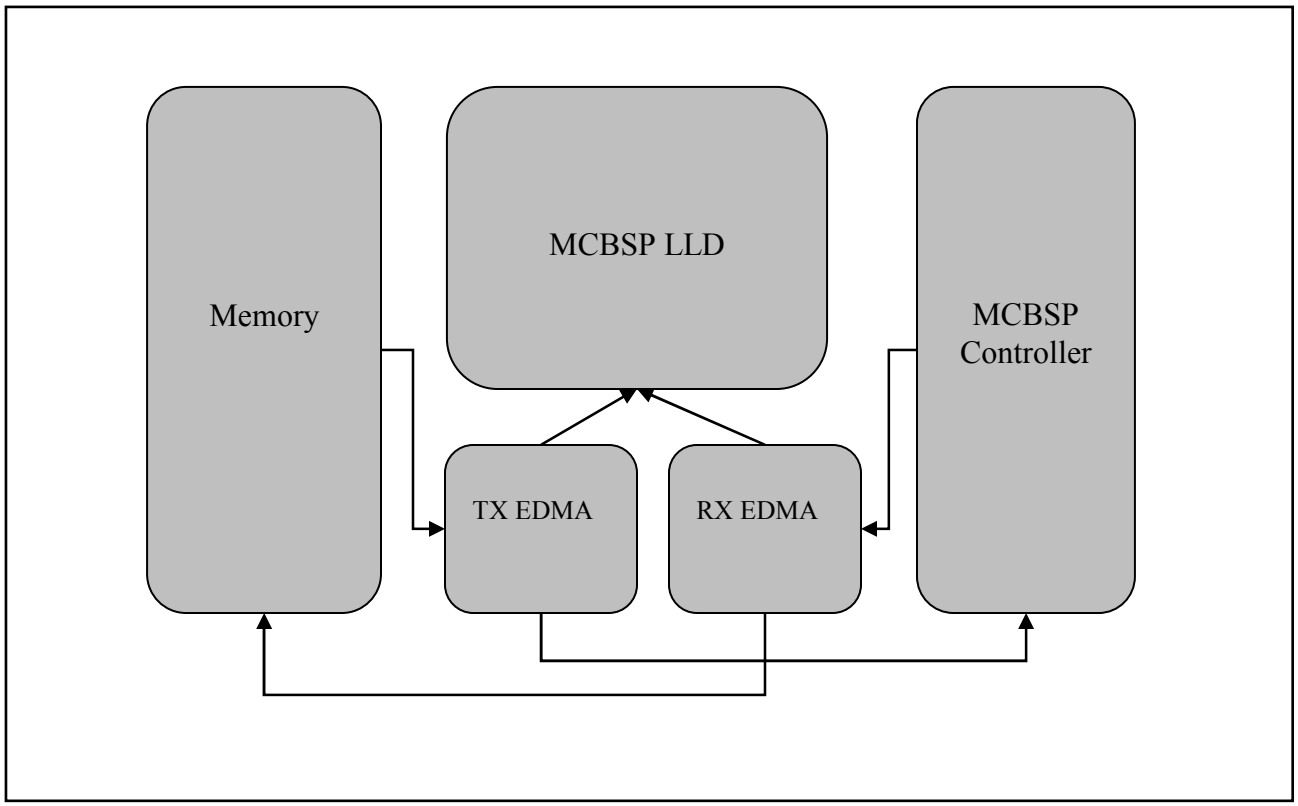

Figure 2: MCBSP LLD Software Overview

Figure 2: MCBSP LLD Software Overview depicts the various components involved in the transfer of data when the MCBSP driver runs on the DSP. Serial data is stored in the memory by DSP e.g. after decoding the audio data. The main function of MCBSP driver is to program the EDMA channels to move the data from memory to the MCBSP interface on every transfer event

from the MCBSP (TX path). Similarly, the driver can configure EDMA channels to move data received on MCBSP interface to the memory for DSP use (RX path).

The EDMA3 channel controller services MCBSP peripheral in the background of DSP operation, without requiring any DSP intervention. Through proper initialization of the EDMA3 channels, they can be configured to continuously service the peripheral throughout the device operation. Each event available to the EDMA3 has its own dedicated channel, and all channels operate simultaneously. The only requirements are to use the proper channel for a particular transfer and to enable the channel event in the event enable register (EER). When programming an EDMA3 channel to service MCBSP peripheral, it is necessary to know how data is to be presented to the DSP. Data is always provided with some kind of synchronization event as either one element per event (non-bursting) or multiple elements per event (bursting).

4.3 Key Features

Following are the key features of MCBSP LLD software:

- Multi-instance support and re-entrant driver
- Each instance can operate as a receiver and or transmitter
- Supports multiple data formats
- Can be configured to operate in multi-slot TDM, DSP (used in audio data transfer)
- Mechanisms to transmit desired data (such as NULL tone) when idle

5 Design

This section explains the overall architecture of MCBSP device driver, including the device driver functional partitioning as well as run-time considerations. The MCBSP LLD driver provides well-defined API layers which allow applications to use the MCBSP peripheral to send and receive data.

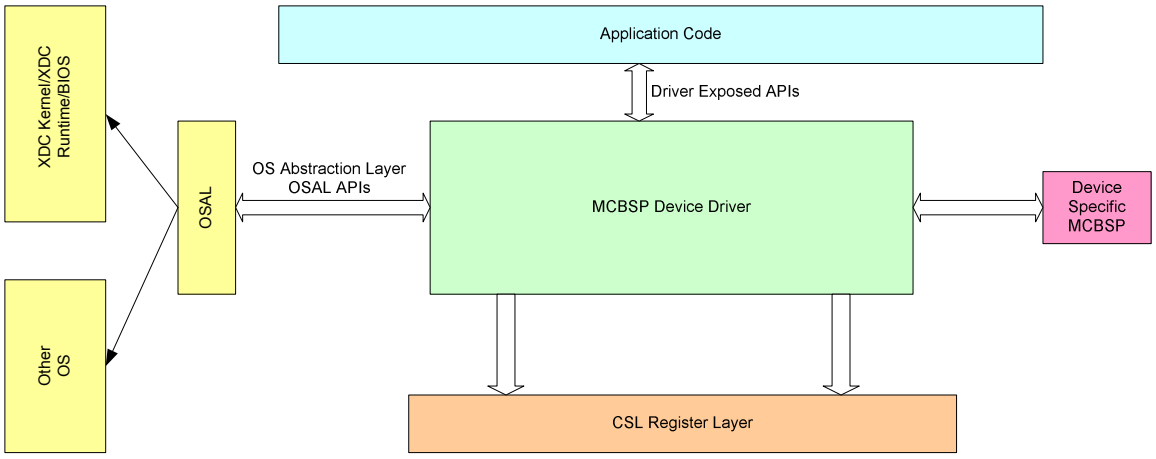

Figure 3: MCBSP LLD Driver Architecture

The Figure 3: MCBSP LLD Driver Architecture illustrates the following key components:

1.) **MCBSP Device Driver**
This is the core MCBSP device driver. The device driver exposes a set of well-defined APIs which are used by the application layer to send and receive data via the MCBSP peripheral. The driver also exposes a set of well-defined OS abstraction APIs which are used to ensure that the driver is OS independent and portable. The MCBSP driver uses the CSL MCBSP register layer for all MCBSP MMR access. The MCBSP driver also interfaces with the EDMA3 library to be able to transfer data to and from MCBSP peripheral and data memory.

2.) **Device Specific MCBSP Layer**
This layer implements a well defined interface which allows the core MCBSP driver to be ported on any device which has the same MCBSP IP block. This layer may change for every device.

3.) **Application Code**
This is the user of the driver and its interface with the driver is through the well-defined APIs set. Application users use the driver APIs to send and receive data via the MCBSP peripheral.

4.) **Operating System Abstraction Layer (OSAL)**
The MCBSP LLD is OS independent and exposes all the operating system callouts via this OSAL layer.

5.) **CSL Register Layer**
The CSL register layer is the IP block memory mapped registers which are generated by the IP owner. The MCBSP LLD driver directly accesses the MMR registers.

5.1 MCBSP Driver Initialization

The MCBSP Driver initialization API needs to be called only once and it initializes the internal driver data structures like device objects. Application developers need to ensure that they call the MCBSP Driver Init API before they call the MCBSP Device Initialization.

The following API is used to initialize the MCBSP Driver.

```
int32_t mcbspInit (void)
```

The function returns `MCBSP_STATUS_COMPLETED` on success indicating that the MCBSP driver internal data structures have been initialized correctly.

5.2 MCBSP Peripheral Configuration

The MCBSP driver provides a sample implementation sequence which initializes the MCBSP IP block. The MCBSP Device initialization API is implemented as a sample prototype:

```
void McbspDevice_init (void)
```

The function initializes all the instance specific information like base address of instance CFG registers, FIFO address for the instance, TX and RX CPU event numbers, TX and RX EDMA event numbers etc. The function also sets the inUse field of MCBSP instance module object to FALSE so that the instance can be used by an application which will create it. In loop job enabled mode the LOOP Job buffers and the mute buffers are initialized. The non-loop job mode doesn't have any LOOP Job buffers so only mute buffers are initialized. Please refer to the figure below for the typical control flow during the device initialization.

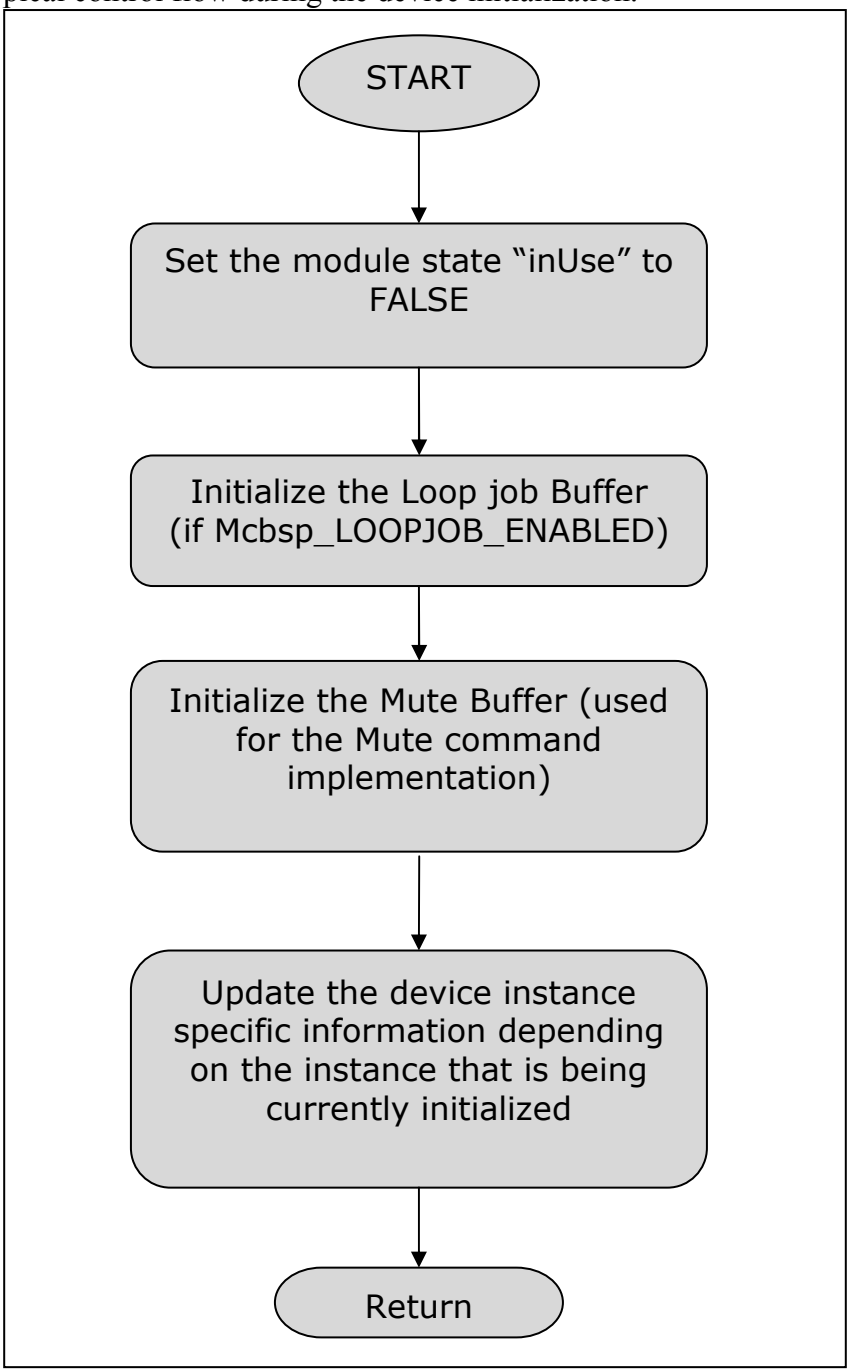

Figure 4: Device Initialization Sequence

The Figure 4: Device Initialization Sequence depicts the typical control flow during the initialization of the MCBSP device.

This implementation is **sample only** and application developers are recommended to modify it as deemed necessary. The initialization sequence is **not** a part of the MCBSP driver library. This was done because the MCBSP Device Initialization sequence has to be modified and customized by application developers. If the initialization sequence was a part of the MCBSP driver then it would require the driver to be rebuilt. Moving this API outside the driver realm solves this issue. The MCBSP Device Initialization API should only be called after calling the MCBSP Device Init API. Failure to do so will result in unpredictable behaviors.

5.3 MCBSP Driver External Interface (Public APIs)

The following table outlines the basic interfaces provided by MCBSP LLD.

Function	Description
`mcbspBindDev`	The `mcbspBindDev` function is called by the application after MCBSP device initialization. The `mdBindDev` performs following actions: ❖ Acquire the device handle for the specified instance of MCBSP on the SOC. ❖ Configure the MCBSP device instance with the specified parameters (or default parameters, if there is no external configuration).
`mcbspUnBindDev`	The `mcbspUnBindDev` function is called to delete an instance of the Mcbsp driver. It will unroll all the changes done during the bind operation and free all the resources allocated to the MCBSP.
`mcbspCreateChan`	The `mcbspCreateChan` function creates a TX or RX channel on the specified MCBSP instance. Application has to specify the mode in which the channel has to be created through the "`mode`" parameter. The MCBSP driver supports only two modes of channel creation (input and output mode) for every device instance. It performs following actions: ❖ The required EDMA channel and spare PARAM sets are acquired and configured. ❖ The required TX or RX sections (clocks, SRGR, frame sync etc.) are setup.
`mcbspDeleteChan`	The `mcbspDeleteChan` deletes a channel created on a MCBSP instance. It frees all the resources allocated during the creation of the channel.
`mcbspSubmitChan`	The `mcbspSubmitChan` is invoked with the appropriate channel handle and IOBuf (aka frame) containing the operation to be performed and required parameters needed for programming the EDMA channels.

`mcbspGblXmtIsr`	This function is the interrupt service routine for the MCBSP TX event.
`mcbspGblRcvIsr`	This function is the interrupt service routine for the MCBSP RX event.
`mcbspControlChan`	The `mcbspControlChan` function is used to issue a control command to the MCBSP driver. Please refer to the list of control commands supported by the MCBSP driver. ❖ Typical commands supported are PAUSE, RESUME, STOP, START etc.

5.3.1 Driver Instance Binding

The binding function (`mcbspBindDev`) of the MCBSP driver is called to allocate and configure a MCBSP instance as specified by `devid`. Each driver instance corresponds to one hardware instance of the MCBSP. The function performs following actions:

- Check if the instance being created is already in use by checking "`inUse`".
- Update the instance object with the user supplied parameters.
- Initialize all the channel objects (TX and RX) with default parameters.
- Initialize queues to hold the pending frames and currently executing frames (floating queue).
- Configure the MCBSP to receive the Frame Sync and bit clocks either externally or internally for both receiver and transmitter depending on the user supplied parameters.
- Return the device handle.

The driver binding operation expects the following parameters:

1. Pointer to hold the function returned device handle.
2. Instance number of the MCBSP instance being created.
3. Pointer to the user provided device parameter structure required for the creation of device instance. The user provided device parameter structure will be of type "`Mcbsp_Params`".

Please refer the Figure 5: Driver Instance Binding below for the control flow of driver Bind operation.

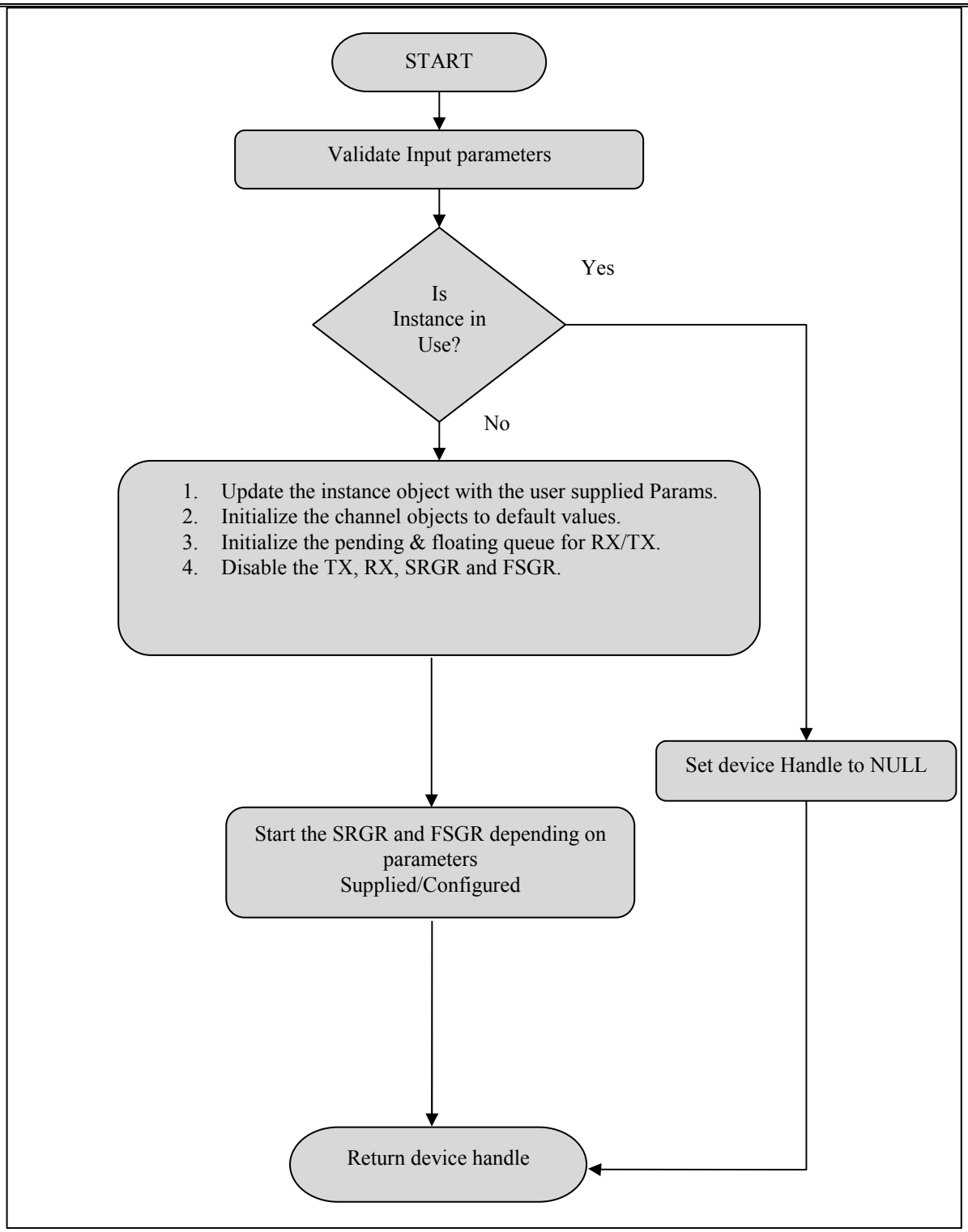

Figure 5: Driver Instance Binding

5.3.2 Channel Creation

Once the application has created a device instance, it 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. The driver allows at most two channels per MCBSP instance to be created which are a transmit channel (TX path e.g. audio playback or data transmission) and a receive channel (RX path e.g. audio recording or data reception). The application can create a communication channel by calling `mcbspCreateChan` function. The application should call `mcbspCreateChan` with the appropriate "`mode`" (`MCBSP_MODE_OUTPUT` or `MCBSP_MODE_INPUT`) parameter for the type of the channel to be created.

The application needs to supply the parameters which will characterize the features of the channel e.g. number of slots, slot width etc. The application can use the "`Mcbsp_ChanParams`" structure to specify the parameters to configure the channel.

The `mcbspCreateChan` function performs the following actions:

- Validates the input parameters given by the application.
- Checks if the requested channel is already opened or not. If it is already opened then the driver will flag an error to the application else the requested channel will be allocated.
- Updates the appropriate channel object with the user supplied parameters.
- MCBSP is configured with the appropriate word width.
- EDMA parameters for the requested channel are setup.
- If the global error callback function registration is enabled, the appropriate user supplied function is registered to be called in case of an error.
- If the LOOPJOB configuration is enabled then the respective section (TX or RX) is enabled and the EDMA transfer is enabled.
- If the channel creation fails then it will perform a cleanup and also free all the resource allocated by it till now.
- If the complete process of channel creation is successful, then it will return a unique channel handle to the application. 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.

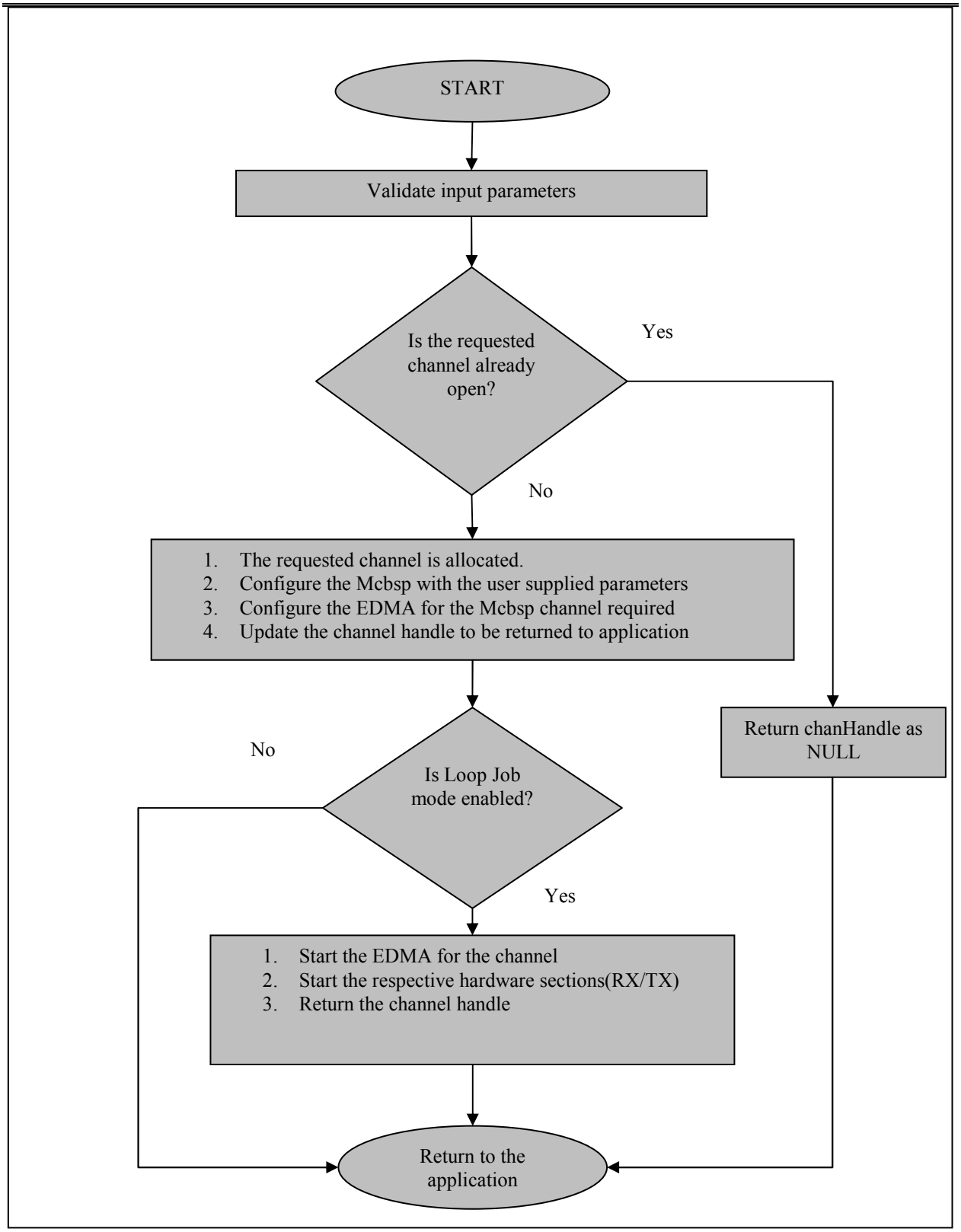

Figure 6: Create Channel Flow Diagram

5.3.3 I/O Frame Processing

MCBSP driver provides `mcbspSubmitChan` interface to submit `ioBufs` (frames) for the I/O transactions to be performed. Application invokes this API for data transfer using MCBSP. This API submits a `Mcbsp_IOBuf` frame containing all the transfer parameters needed by the driver to program the underlying hardware for data transfer. The `mcbspSubmitChan` function handles the command code passed to it as part of the `Mcbsp_IOBuf` structure.

The command codes supported by the MCBSP driver are: Mcbsp_IOBuf_Cmd_READ, Mcbsp_IOBuf_Cmd_WRITE, Mcbsp_IOBuf_Cmd_ABORT and Mcbsp_IOBuf_Cmd_FLUSH.

- **Mcbsp_IOBuf_Cmd_READ**: Read data from MCBSP interface and store it in memory (input channel – RX path).
- **Mcbsp_IOBuf_Cmd_WRITE**: Write data from memory to MCBSP interface (output channel – TX path).
- **Mcbsp_IOBuf_Cmd_ABORT and Mcbsp_IOBuf_Cmd_FLUSH.** To abort or flush I/O requests already submitted, all I/O requests pending in the driver must be completed and returned to the device independent layer. The `mcbspSubmitChan` function will de-queue each of the I/O requests from the driver's channel queue. It will then set the size and status fields in the `Mcbsp_IOBuf`. Finally, it will call the callback function registered for the channel. **Note:** The behavior of the driver will be same for both the ABORT and FLUSH commands i.e. all the frames will be aborted and returned back to the application.

The `mcbspSubmitChan` function performs the following actions:

- The input `Mcbsp_IOBuf` frame is validated.
- If the driver has sufficient frames then the current frame is loaded in to the pending queue.
- Otherwise the frame is programmed into the link PARAMs of the EDMA.
- In NON LOOP JOB mode, the first frame is always loaded in to the main transfer channel. The subsequent two frames are loaded into the spare PARAM sets of the EDMA. Also if this is the first frame for the driver then the clocks are started as per the configuration of the channel. Any other frames after this are loaded into the pending queue. These frames will be loaded by the EDMA callback into the appropriate PARAM set of the EDMA.

5.3.3.1 Asynchronous I/O Mechanism

The MCBSP driver supports asynchronous I/O mechanism. In this mechanism, multiple I/O requests can be submitted by the application without causing it to block while waiting for the previous I/O requests to complete. Application can submit multiple I/O requests using `mcbspSubmitChan` API. The application callback function registered during the transfer request submission will be called upon transfer completion by the driver. The driver internally will queue the I/O frames submitted to support the asynchronous I/O functionality.

5.3.4 Control Commands

MCBSP driver implements device specific control functionality which may be useful for any application, which uses the MCBSP driver. Application may invoke the control functionality through a call to `mcbspControlChan`. MCBSP driver supports the following control functionality.

The below table lists the control commands supported by the MCBSP driver

Command	Command Argument	Explanation
`Mcbsp_IOCTL_START`	`NULL`	Starts the requested (TX or RX) section.
`Mcbsp_IOCTL_STOP`	`NULL`	Stops the requested (TX or RX) section.
`Mcbsp_IOCTL_MUTE_ON`[1]	`NULL`	Mutes the TX channel
`Mcbsp_IOCTL_MUTE_OFF`[2]	`NULL`	Un-Mutes the TX channel
`Mcbsp_IOCTL_PAUSE`	`NULL`	Pauses the selected section (channel)
`Mcbsp_IOCTL_RESUME`	`NULL`	Resumes a previously paused channel.
`Mcbsp_IOCTL_CHAN_RESET`	`NULL`	Resets the requested channel.
`Mcbsp_IOCTL_DEVICE_RESET`	`NULL`	Resets the entire device by resetting both the channels.
`Mcbsp_IOCTL_SRGR_START`	`NULL`	starts the sample rate generator
`Mcbsp_IOCTL_SRGR_STOP`	`NULL`	stops the sample rate generator
`Mcbsp_IOCTL_FSGR_START`	`NULL`	starts the frame sync generator
`Mcbsp_IOCTL_FSGR_STOP`	`NULL`	Stops the frame sync generator.

[1] This command is applicable only for the TX section
[2] This command is applicable only for the TX section

Mcbsp_IOCTL_SET_CLKMODE,	Mcbsp_TxRxClkMode *	Configure the bit clock mode.
Mcbsp_IOCTL_SET_FRMSYNCMODE,	Mcbsp_FsClkMode *	Configure the frame sync mode
Mcbsp_IOCTL_CONFIG_SRGR,	Mcbsp_srgConfig *	Configure the sample rate generator
Mcbsp_IOCTL_SET_BCLK_POL	Mcbsp_ClkPol *	Set the Bit clock polarity
Mcbsp_IOCTL_SET_FRMSYNC_POL	Mcbsp_FsPol *	Set the frame sync polarity
Mcbsp_IOCTL_MODIFY_LOOPJOB	Mcbsp_ChanParams *	Configure the user supplied loop job buffer.
Mcbsp_IOCTL_RECEIVE_SYNCERR_INT_ENABLE	NULL	Enable the SYNCERR for RX section
Mcbsp_IOCTL_XMIT_SYNCERR_INT_ENABLE	NULL	Enable the SYNCERR for TX section
Mcbsp_IOCTL_LOOPBACK	Mcbsp_Loopback *	Enable/disable the loopback mode
Mcbsp_IOCTL_CHAN_RESET	NULL	Resets the required channel
Mcbsp_IOCTL_DEVICE_RESET	NULL	Resets both the TX and RX channels

The typical control flow for the MCBSP 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 basic control flow for the handling of the control commands for the driver is shown in Figure 7: Control Command Flow. Please note that the individual command handling is not detailed here.

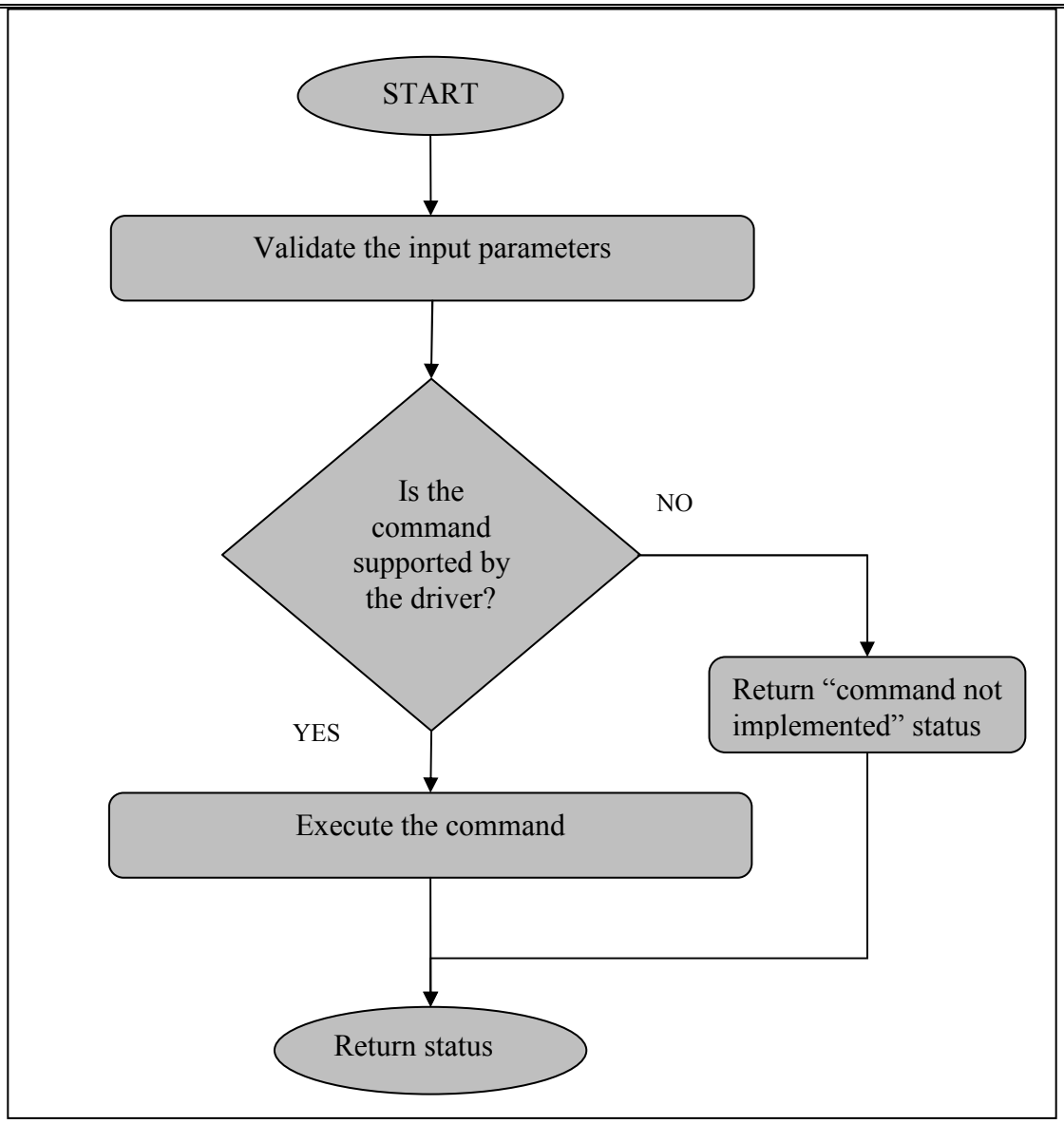

Figure 7: Control Command Flow

5.3.5 Channel Deletion

Once a channel has completed all the transactions it can be closed so that all the resources allocated to the channel can be freed. The driver provides `mcbspDeleteChan` API to delete a previously created MCBSP channel for an instance. The actions performed during the channel deletion are as follows:

- The channel to be deleted is reset.
- The reset operation aborts all the packets in the pending queue and also the packets in the current active queue.
- The EDMA transfer for this channel is disabled.
- The MCBSP state machines are stopped.
- The interrupt handlers are unregistered.

- All the spare PARAM sets of the EDMA are freed.
- The status of the channel is updated to DELETED.

5.3.6 Driver Instance Unbinding/Deletion

The MCBSP driver provides `mcbspUnBindDev` interface to delete a driver instance. The function de-allocates all the resources allocated to the instance object during the driver binding operation. The operations performed by the unbind operation are as listed below:

- Check if both the TX and the RX channels are closed.
- Update the instance object.
- Set the status of the driver instance to "DELETED".
- Set the status of the instance "`inUse`" to FALSE (so that instance can be used again).

5.4 Data Structures

5.4.1 Constants and Enumerations

5.4.1.1 Mcbsp_TXEVENTQUE

This constant defines the EDMA3 event queue to be used in case of Transmit channel operation.

Definition

```
#define Mcbsp_TXEVENTQUE (1u)
```

Comments

None

Constraints

Please check the available event queues in the EDMA3 before changing/modifying this.

See Also

None

5.4.1.2 Mcbsp_RXEVENTQUE

This constant defines the EDMA3 event queue to be used in case of Receive channel operation.

Definition

```
#define Mcbsp_RXEVENTQUE (2u)
```

Comments

None

Constraints

Please check the available event queues in the EDMA3 before changing/modifying this.

See Also

None

5.4.1.3 Mcbsp_OpMode

This enumeration defines the operating mode of the MCBSP driver.

Definition

```
typedef enum Mcbsp_OpMode_t
{
    Mcbsp_OpMode_POLLED = 0,
    Mcbsp_OpMode_INTERRUPT,
    Mcbsp_OpMode_DMAINTERRUPT
} Mcbsp_OpMode;
```

Comments

None

Constraints

Only EDMA mode of operation is supported by the MCBSP driver.

See Also

None

5.4.1.4 Mcbsp_DevMode

This enumeration is used to define the operational mode of the MCBSP device like normal MCBSP device or SPI device (master/slave) mode.

Definition

```
typedef enum Mcbsp_DevMode_t
{
    Mcbsp_DevMode_McBSP,
} Mcbsp_DevMode;
```

Comments

None

Constraints

The SPI mode of operation is not supported as the underlying hardware doesn’t support the same.

See Also

None

5.4.1.5 Mcbsp_BufferFormat

This enumeration is used to specify the different types of buffer formats supported by the MCBSP driver.

Definition

```
typedef enum Mcbsp_BufferFormat_t
{
    Mcbsp_BufferFormat_1SLOT,
    Mcbsp_BufferFormat_MULTISLOT_NON_INTERLEAVED,
    Mcbsp_BufferFormat_MULTISLOT_INTERLEAVED
} Mcbsp_BufferFormat;
```

Comments

None

Constraints

None

See Also

None

5.4.2 Internal Data Structures

5.4.2.1 Driver Instance Object

This structure is the MCBSP driver's internal data structure. This data structure is used by the driver to hold the information specific to the MCBSP instance. There will be one unique instance object for every instance of the MCBSP controller supported by the driver.

Definition

```
typedef struct Mcbsp_Object_t
{
    int32_t                     instNum;
    Mcbsp_DriverState           devState;
    Mcbsp_OperatingMode         mode;
    Mcbsp_OpMode                opMode;
    Bool                        enablecache;
    Mcbsp_HwInfo                hwInfo;
    Bool                        stopSmFsXmt;
```

```
    Bool                    stopSmFsRcv;
    Mcbsp_ChannelObj        xmtObj;
    Mcbsp_ChannelObj        rcvObj;
    Mcbsp_srgConfig         srgrConfig;
    Bool                    txSrgEnable;
    Bool                    rxSrgEnable;
    Bool                    srgConfigured;
    volatile Bool           srgEnabled;
    Bool                    txFsgEnable;
    Bool                    rxFsgEnable;
    Bool                    fsgConfigured;
    volatile Bool           fsgEnabled;
    Uint32                  retryCount;
    Bool                    loopJobMode;
} Mcbsp_Object;
```

Fields

instNum	Instance number of the MCBSP.
devState	Current state of the driver (Created/Deleted).
Mode	Operating mode of the MCBSP (Mcbsp, SPI master Mode, SPI slave mode).
opMode	Mode of operation of the driver(POLLED/INTERRUPT/DMA)
enableCache	Whether the driver should take care of cache cleaning operations for the buffers submitted by the application
hwInfo	Structure holding the hardware information related to the instance (e.g. interrupt numbers, base address etc).
stopSmFsXmt	State of transmit state machine. (TRUE = stopped, FALSE = running).
stopSmFsRcv	State of receive state machine. (TRUE = stopped, FALSE = running).
xmtObj	Transmit channel object
rcvObj	Receive channel object
srgrConfig	Sample rate generator configurations supplied by the user.
txSrgEnable	Variable to indicate if the sample rate generator is required by the TX section.

rxSrgEnable	Variable to indicate if the sample rate generator is required by the RX section.
srgConfigured	Variable to indicate if the sample rate generator is configured or not.
srgEnabled	Variable to indicate if the sample rate generator is running.
txFsgEnable	Variable to indicate if the frame sync generator is required by the TX section.
rxSrgEnable	Variable to indicate if the frame sync generator is required by the RX section.
fsgEnabled	Variable to indicate if the frame sync generator is running.
retryCount	Retry count to be used by the driver when waiting in indefinite loops. (e.g. waiting for the TX to get empty etc).
loopJobMode	check if the loop job mode is enabled or not
pscPwrmEnable	Option to enable or disable the PSC control

Comments

1. The MCBSP Driver works only in the EDMA mode of operation.
2. One instance object represents one instance of the driver.

Constraints

None

See Also

Mcbsp_ChannelObj

5.4.2.2 Channel Object

This structure is the MCBSP driver's internal data structure. This data structure is used by the driver to hold the information specific to a channel. There will be at most two channels supported per MCBSP instance (one for TX and one for RX). It is used to maintain the information pertaining to the channel like the current channel state, callback function etc. This structure is initialized by `mcbspCreateChan` and a pointer to this is passed down to all other channel related functions. Lifetime of the data structure is from its creation by `mcbspCreateChan` till it is invalidated (deleted) by `mcbspDeleteChan`.

Definition

```
typedef struct Mcbsp_ChannelObj_t
{
    uint16_t                    mode;
    Mcbsp_DriverState           chanState;
```

```
    Void*                   devHandle;
    Mcbsp_CallbackFxn       cbFxn;
    void*                   cbArg;
    void*                   edmaHandle;
    uint32_t                edmaEventQue;
    EDMA3_RM_TccCallback    edmaCallback;
    uint32_t                xferChan;
    uint32_t                tcc;
    uint32_t                pramTbl[Mcbsp_MAXLINKCNT];
    uint32_t                pramTblAddr[Mcbsp_MAXLINKCNT];
    void*                   ptrQPendList;
    void*                   ptrQFloatList;
    Mcbsp_IOBuf             *tempIOBuf;
    Mcbsp_IOBuf             *dataIOBuf;
    uint32_t                submitCount;
    Mcbsp_BufferFormat      dataFormat;
    volatile Bool           nextFlag;
    volatile Bool           bMuteON;
    volatile Bool           paused;
    volatile Bool           flush;
    volatile Bool           isTempIOBufValid;
    Bool                    enableHwFifo;
    Mcbsp_GblErrCallback    gblErrCbk;
    uint32_t                userDataBufferSize;
    void*                   loopJobBuffer;
    uint16_t                loopJobLength;
    uint32_t                userLoopJobLength;
    uint32_t                nextLinkParamSetToBeUpdated;
    volatile Bool           loopjobUpdatedinParamset;
    uint16_t                roundedWordWidth;
    uint16_t                currentDataSize;
    Mcbsp_DataConfig        chanConfig;
    Mcbsp_ClkSetup          clkSetup;
    Mcbsp_McrSetup          multiChanCtrl;
    uint32_t                chanEnableMask[4];
    Bool                    userLoopJob;
    int32_t                 currentError;
}Mcbsp_ChannelObj;
```

Fields

mode	Current operating mode of the channel (INPUT/OUTPUT).
chanState	Current state of the channel (opened/closed).
devHandle	Pointer to the instance object.
cbFxn	Callback function pointer
cbArg	Callback function argument
edmaHandle	Pointer to the EDMA handle given by the application.
edmaEventQue	EDMA event queue to be used by this channel.
edmaCallback	EDMA callback function pointer.
xferChan	The EDMA transfer channel to be used.
tcc	Transfer completion code to be used in case of EDMA mode.
pramTbl	Value of the two spare PARAM sets issued by the EDMA driver.
pramTblAddr	Address of the two spare paramsets.
ptrQPendList	Pointer to queue for holding the pending packets.
ptrQFloatList	Pointer to queue for holding currently executing packets.
tempIOBuf	Temporary place holder for the currently completed frame.
dataIOBuf	Pointer to hold the `Mcbsp_IOBuf` frame
submitCount	Total number of packets held in the driver for this channel
dataFormat	The format in which the MCBSP data is arranged in the buffer.
nextFlag	Flag used in stopping the MCBSP state machines.
bMuteON	Flag to indicate if the mute is ON.
paused	Flag to indicate if the channel is paused.
flush	Flag to indicate if the flush command is issued to the driver.
isTempIOBufValid	Flag to indicate if the "`tempIOBuf`" is holding a valid frame.
enableHwFifo	Flag to indicate if the hardware FIFO is to be enabled for this channel (RX/TX).

gblErrCbk	Application registered callback function to be called in case of an error.
userDataBufferSize	Size of the user supplied buffer.
loopJobBuffer	Loop job buffer to be used when the driver does not have any more packets for the I/O
loopJobLength	Length of the loop job buffer.
userLoopJobLength	User specified loop job's length.
nextLinkParamSetToBeUpdated	Variable to indicate which of the spare paramset is to be updated next.
loopjobUpdatedinParamset	Variable to indicate if the loop job is loaded in to the paramset.
roundedWordWidth	The actual word width to be transferred per sync event.
currentDataSize	The size of the current data packet
chanConfig	Channel configuration required for the configuring of the channel.
clkSetup	Clock setup to be used for this channel.
multiChanCtrl	Multiple channel selection settings.
chanEnableMask	Mask for the channels to be enabled
userLoopJob	Variable to indicate if the user loop job is used or internal driver loop job buffer.
currentError	Current packet error status

Comments

1. Only 2 channels are supported per instance

Constraints

None

See Also

Mcbsp_Object

5.4.3 External Data Structures

5.4.3.1 Mcbsp_Params

This structure is used to supply user parameters during the creation of the driver instance. The structure is as defined below:

Definition

```
typedef struct Mcbsp_Params_t
```

```
{
    Mcbsp_DevMode           mode;
    Mcbsp_OpMode            opMode;
    Bool                    enablecache;
    Mcbsp_Loopback          dlbMode;
    Mcbsp_srgConfig        *srgSetup;
} Mcbsp_Params;
```

Fields

mode	Operating mode of the Mcbsp (Mcbsp, SPI master Mode, SPI slave mode). ***Default mode is MCBSP mode****.*
opMode	Mode of operation of the controller. ***Default is EDMA mode.*** ***Note:*** *Only EDMA mode is supported for the MCBSP mode of operation*
enableCache	Whether the driver should take care of cache cleaning operations for the buffers submitted by the application.
dlbMode	Digital loop back mode selection.
srgSetup	Sample rate generator setup.

Comments

1. The Mcbsp Driver works only in the EDMA mode of operation.

Constraints

None

See Also

Mcbsp_srgConfig

5.4.3.2 Mcbsp_ChanParams

This structure is used to supply user parameters during the creation of the channel instance. During the creation of the channel, user needs to supply the above structure with the appropriate parameters as per the required mode of operation. The structure is defined as below:

Definition

```
typedef struct Mcbsp_ChanParams_t
{
    uint32_t                wordWidth;
    void*                   userLoopJobBuffer;
    uint16_t                userLoopJobLength;
    Mcbsp_GblErrCallback    gblCbk;
```

```
    void*                   edmaHandle;
    uint32_t                edmaEventQue;
    uint32_t                hwiNumber;
    Mcbsp_BufferFormat      dataFormat;
    Bool                    enableHwFifo;
    Mcbsp_DataConfig       *chanConfig;
    Mcbsp_ClkSetup         *clkSetup;
    Mcbsp_McrSetup         *multiChanCtrl;
    uint32_t                chanEnableMask[4];
}Mcbsp_ChanParams;
```

Fields

wordWidth	Word width per slot
userLoopJobBuffer	User supplied loop job buffer
userLoopJobLength	User supplied buffer length
gblCbk	Pointer to the function to handle the Error conditions.
edmaHandle	Handle to the EDMA driver.
edmaEventQue	Event queue of the EDMA to be used by this channel.
hwiNumber	HWI number for the ECM group in which the event is configured
dataFormat	Buffer format to be used by the application
enableHwFifo	Flag to indicate whether hardware FIFO's are to be enabled.
chanConfig	Channel configuration settings.
clkSetup	Clock configuration settings.
multiChanCtrl	Multi channel control settings.
chanEnableMask	Multiple channel selection mask

Comments

1. The user can provide the Loop Job buffer, if required. Otherwise the "userLoopJobBuffer" and "userLoopJobLength" should be set to NULL and 0 respectively. In case, the user has not provided the buffer then the driver will use its internal buffer.

   Note: This is applicable only if the driver is in loop job mode.

2. "gblCbk" function will be called in the ISR context hence appropriate care should be taken that the function confirms to the ISR coding guidelines.

3. "hwiNumber" needs to specified according to the ECM event group that the channel being configured falls into.

Constraints

See above.

See Also

Mcbsp_DataConfig

5.4.3.3 Mcbsp_srgConfig

This is the MCBSP sample rate generator configuration structure. The application needs to configure the sample rate generator to generate the BCLK and Frame Sync signals at the specified rate in MCBSP master mode.

Definition

```
typedef struct Mcbsp_srgConfig_t
{
    Bool            gSync;
    Mcbsp_ClkSPol   clksPolarity;
    Mcbsp_SrgClk    srgInputClkMode;
    uint32_t        srgrInputFreq;
    uint32_t        srgFrmPulseWidth;
}Mcbsp_srgConfig;
```

Fields

gSync	Sample rate generator synchronization bit
clksPolarity	CLKS polarity used to drive the CLKG and FSG clocks.
srgInputClkMode	Source for the sample rate generator.
srgrInputFreq	Input frequency for the Sample rate generator
srgFrmPulseWidth	Frame sync width

Comments

1. This structure will be required to specify the sample rate generator settings if sample rate generator is required.
2. The driver will decide internally if the sample rate generator need to be enabled or not depending on the TX or RX channel clock requirements

Constraints

None

See Also

Mcbsp_Params

5.4.3.4 Mcbsp_DataConfig

This specifies the configuration for the MCBSP data stream including whether it is single phase or dual phase, number of frames, the word length in each phase and data delay etc.

Definition

```
typedef struct Mcbsp_DataConfig_t
{
    Mcbsp_Phase              phaseNum;
    Mcbsp_WordLength         wrdLen1;
    Mcbsp_WordLength         wrdLen2;
    uint32_t                 frmLen1;
    uint32_t                 frmLen2;
    Mcbsp_FrmSync            frmSyncIgn;
    Mcbsp_DataDelay          dataDelay;
    Mcbsp_Compand            compandSel;
    Mcbsp_BitReversal        bitReversal;
    Mcbsp_IntMode            intMode;
    Mcbsp_Rjust              rjust;
    Mcbsp_DxEna              dxState;
}Mcbsp_DataConfig;
```

Fields

phaseNum	Option to choose single phase or dual phase frame.
wrdLen1	Word length for the first frame.
wrdLen2	Word length for the second frame. *Will be used only in case of a dual phase frame.*
frmLen1	Length of the first frame.
frmLen2	Length of the second frame. *To be specified only in case of dual frame.*
frmSyncIgn	Option to select the action to be taken in case if an unexpected frame sync.
dataDelay	Data delay from the frame sync
comapandSel	Companding (a-law, mu-law etc.) selection
bitReversal	Option to select the bit reversal of data (MSB first or LSB first).

intMode	Event which should generate an CPU interrupt
rjust	Receive data justification settings
dxState	DX pin high impedance state enable disable option.

Comments

1. The frmLen2 and wrdLen2 options should be used only in case of an dual phase frame.
2. dxState option is applicable only while creating a channel for transmission.
3. rjust option is applicable only in case of creating a channel for reception.

Constraints

None

See Also

Mcbsp_Params

5.4.3.5 Mcbsp_ClkSetup

This structure is used to configure the clock settings for the Mcbsp channel.

Definition

```
typedef struct Mcbsp_ClkSetup_t
{
    Mcbsp_FsClkMode         frmSyncMode;
    uint32_t                samplingRate;
    Mcbsp_TxRxClkMode       clkMode;
    Mcbsp_FsPol             frmSyncPolarity;
    Mcbsp_ClkPol            clkPolarity;
}Mcbsp_ClkSetup;
```

Fields

frmSyncMode	Frame sync generator mode (Internal/external).
samplingRate	Frame sync frequency.
clkMode	Bit clock mode (internal/external)
frmSyncPolarity	Frame sync polarity (active high/active low)
clkPolarity	Bit clock polarity

5.5 Supported Data Formats

Mcbsp driver expects the data (samples) to be arranged in a specific format when requesting for an I/O transfer. These formats are explained under scenario of using 1-slot or multiple slots. The sections below capture the details of supported data formats.

MCBSP Mode	Data Format	Buffer Format
1-Slot	Interleaved data Format	*Mcbsp_BufferFormat_1SER_1SLOT*
Multi-Slot	Interleaved data Format	*Mcbsp_BufferFormat_1SER_MULTISLOT_NON_INTERLEAVED*
Multi-Slot	Non-interleaved data format	*Mcbsp_BufferFormat_1SER_MULTISLOT_INTERLEAVED*

5.5.1 1-Slot Data Format

This format is used when a single slot is used to transfer the data. The expected format is as depicted below:

[<**Slot1**-Sample**1**>, <**Slot1**-Sample**2**>…<**Slot1**-Sample**N**>]

The size (number of bytes) that would be required to specify during an I/O request is computed using the formula `size = <word width>*<number of samples N>`.

The key configurations (sample) are:

- `Mcbsp_ChanParams.dataFormat = Mcbsp_BufferFormat_1SER_1SLOT;`
- `Mcbsp_ChanParams. noOfTdmChans = 1;`
- The size of the I/O request is computed as `<No. of Bytes per Sample> * <No. of Samples>`. This value should be given as a size parameter to `mcbspSubmitChan` function.
- Idle Time data pattern length computation: Minimum length should be **<word width in bytes>** or an integral multiple of computed value. While allocating a buffer, allocate `<computed value> * <no. of slots enabled>`.

5.5.2 Multi-Slot Non-Interleaved Data Format

When configured in this mode, it is expected that driver is configured to use multiple slots. The expected data format is as depicted below. When configured to use multiple slots, the samples are expected to be contiguous for a given slot as shown below. It is assumed below that number of slots is 2 and number of samples is N.

[<**Slot1**-Sample**1**>, <**Slot1**-Sample**2**>..…<**Slot1**-Sample**N**>,
<**Slot2**-Sample**1**>, < **Slot2**-Sample**2**>….. < **Slot2**-Sample**N**>]

The key configurations (sample) are:

- `Mcbsp_ChanParams.dataFormat = Mcbsp_BufferFormat_1SER_NON_INTERLEAVED;`
- `Mcbsp_ChanParams.noOfTdmChans = N;`
- The size of the I/O request is computed as `<No. of Bytes per Sample> * <No. of Samples> * <No. of slots>`. This value should be given as a size parameter to `mcbspSubmitChan` function.
- Idle Time data pattern length computation: Minimum length should be **<word width in bytes>** or an integral multiple of computed value. While allocating a buffer, allocate `<computed value> * <no. of slots enabled>.`

5.5.3 Multi-Slot Interleaved Data Format

When configured to use multiple slots and interleaved format, the samples are expected to be interleaved for the slots, as depicted below. It is assumed below that number of slots is 2 and number of samples is N.

`[<Slot1-Sample1>, <Slot2-Sample1>…<Slot1-SampleN><Slot2-SampleN>]`

The key configurations (sample) are:

- `Mcbsp_ChanParams.dataFormat = Mcbsp_BufferFormat_1SER_INTERLEAVED;`
- `Mcbsp_ChanParams.noOfTdmChans = N;`
- The size of the I/O request is computed as `<No. of Bytes per Sample> * <No. of Samples> * <No. of slots>`. This value should be given as a size parameter to `mcbspSubmitChan` function.
- Idle Time data pattern length computation: Minimum length should be **<word width in bytes>** or an integral multiple of computed value. While allocating a buffer, allocate `<computed value> * <no. of slots enabled>.`

6 Integration

The MCBSP LLD depends on the following components:

a. CSL
b. EDMA3 LLD

These components need to be installed before the MCBSP driver can be integrated. The MCBSP driver is released in source code and in pre-built library. Applications can decide how to use the MCBSP driver.

The MCBSP Driver release notes indicate the version of the above components which that release is dependent upon. The next steps use the version numbers for illustrative purpose only.

6.1 Pre-built approach

In this approach, the application developers can decide to use the MCBSP driver pre-built libraries as is. The following steps need to be done:

a. The application developers modify their application configuration file to use the MCBSP package.

```
var Mcbsp = xdc.loadPackage('ti.drv.mcbsp');
```

b. Ensure that the XDCPATH is configured to have the path to the PDK package
c. This implies that XDC Configuration scripts will link the application using the MCBSP Driver libraries (`Module.xs`)
d. The application authors need to provide an OSAL implementation file for MCBSP and ensure that this is linked with the application; failure to do so will results in linking errors. Please refer to the MCBSP OSAL header file (`mcbsp_osal.h`) for more information on the API's which need to be provided.

6.2 Rebuild library

In this approach, the application developers can decide to use the MCBSP driver source code and add these files to the application project to rebuild the MCBSP driver code base. The following steps need to be redone:

a. Application developers should port the file "`mcbsp_osal.h`" to their operating system environment. *Developers are recommended to create a copy of this file and place it in their application directory.* They should use the file which is provided in the MCBSP installation only as a template. The goal here should be to map the `Mcbsp_osalXXX` macros to the OS calls directly thus reducing the overhead of an API callout. E.g.

```
#define Mcbsp_osalCreateSem()    (Void*)Semaphore_create(0, NULL, NULL)
```

b. Application developers should port the file "`mcbsp_types.h`" to the application environment. *Developers are recommended to create a copy of this file and place it in their application directory.*

c. Append the include path to the top level MCBSP package directory i.e. if the MCBSP package is installed in `C:\Program Files\Texas Instruments\mcbsp_C6657_1_0_0_0`; then make sure the include path is configured as `C:\Program Files\Texas Instruments\mcbsp_C6657_1_0_0_0\packages`

d. Add the MCBSP driver files listed in the `src` directory to the application build files

The approach above is highlighted in the MCBSP `example` directory.