DSP/BIOS McASP Driver User Guide

USER'S GUIDE

Document Revision History

Rev No	Author(s)	Revision History	Date	Approval(s)
0.1	Pratik Joshi	Migrated from older version of user guide	May 24, 2007	Initial Draft
0.2	Pratik Joshi	Change Release version	May 5, 2007	
0.3	Pratik Joshi	Change data structure, enumeration and sample application section	June 8, 2007	
0.4	Pratik Joshi	Updated BIOS version to 5.31.08	August 29, 2007	
0.5	Nagarjuna K	Updated for DM648/DM6437/C6452/C6424 packjage	November 15, 2007	
0.6	Chandan Nath	Updated for adding compiler switches in build options	May 21, 2008	

implied or expressed warranties in this document and is not responsible for the products based from this document

TABLE OF CONTENTS

TABLE OF FIGURES

1 Introduction

The API reference guide serves as a software programmer's handbook on working with the McASP mini-driver module. This reference guide provides necessary information regarding how this module is designed.

1.1 Terms & Abbreviations

Term	Description
☞	This bullet indicates important information. Please read such text carefully.
❑	This bullet indicates additional information.

1.2 References

1.	spruel1_mcasp.pdf	TMS320DM6437/DM648/C6452/C6424 Multichannel Audio Serial Port (McASP)

1.3 S/W Support

The document provides an overall understanding of the TI McASP mini-driver architecture and its usage. This mini-driver conforms to the IOM mini-driver pattern and interfaces with the application code using the GIO class driver.

This driver uses CSL register overlays for the corresponding hardware platform. More details about this and the tools to build this driver can be found in the system release notes. This driver requires DSP/BIOS and above to operate. The system release notes also contain details on the BIOS version that the driver is compatible with.

1.4 Driver Components

The device driver described here is an IOM mini-driver. It is implemented as the lower layer of a 2-layer device driver model. The upper layer is called the class driver and is the generic DSP/BIOS GIO module. The class driver provides an independent and generic set of APIs and services for a wide variety of mini-drivers and allows the application to use a common interface for I/O requests.

For more information about the IOM device driver model, see the DSP/BIOS Device Driver Developer's Guide (SPRU616).

This document pertains to the McASP mini-driver.

1.5 Default Driver Configuration

Please refer to section 8.1.4 for the driver default configuration

1.6 Driver Capabilities

The significant driver features are:

- Transmit and receive handled by different channels
- Handles interfacing of an EDMA channel to service the McASP transmit/receive section
- Supports interrupt based servicing of the McASP serializers
- Each channel can be used by a task
- Both channels, if active, operate independent of each other for a instance
- Also supports assigning all serializers to one channel
- Handles interrupts generated by the McASP and notifies the application of the same
- Cache is properly handled when McASP is in EDMA mode
- Allows user to configure McASP as per user requirements

1.7 System Requirements

Refer to release notes for the details on Target environment, BIOS version, XDC version…

2 Installation Guide

2.1 Component Folder

Upon installing the McASP driver the following directory structure is found in the driver's directory.

Figure 1. McASP Driver Directory Structure

This top level mcasp folder contains mcasp driver psp header file and XDC package files (package.bld, package.xdc and package.xs)

- **build:** This folder contains mcasp driver library project file. The generated driver library shall be included in the application where MCASP driver have to be used.
- **docs:** This folder contains architecture document, datasheet, release notes and user guide.

  Architecture document contains the driver details which can be helpful for the developers as well as consumers to understand the driver design.

  Datasheet gives the idea about the memory consumption by the driver and description of the top level APIs.

  Release Note gives the details about system requirements, steps to Install/Uninstall the package.This document list the known issues of the driver.

  User Guide provides information about how to use the driver. It contains description of sample applications which guide the end user to make their applications using this driver.
- **Lib:** This folder contains libraries generated in all the configuration modes(debug, idebug, irelease and release)
- **Package:** This folder contains files generated by XDC tool.
- **src:** This folder contains i2c driver source files. It also contains header files that are used by the driver.

2.2 Build

This section describes for each supported target environment:

- Applicable build options
- Supported configurations
- How to enable supported configuration and featured capabilities to allow user for customization

The component might be delivered to user in different formats:

- Source-less i.e. binary executables and object libraries only.
- Source-inclusive i.e.The entire source code is used to implement the driver is included in the delivered product.
- Source-selective ie. Only a part of the overall source is included. This delivery mechanism might be required either because ;certain parts of the driver require soruce level extensions and/or customization at the user's end or because,specific parts of the driver is exposed to user at the source level to insure user's software development.

When source is included as part of the product delivery, the CCS project file is provided as part of the package. When object format is distributed, the driver header files are part of the "src" folder and the driver library is provided in "*pspdrivers**drivers**mcasp**lib"* folder.

2.3 Build Options

To compile driver, change build options as mentioned below:

Optimization level should be configured for –o2 for release and irelease configurations.

The build folder contains a CCS project file that builds the driver into a library for debug and release mode.
Following compiler switches are used to compile for different options.

- **_DEBUG**
  This is used as a flag to compiler whether to include the debug statements inserted in the code into the final image. This flag helps to build DEBUG image of the program. For RELEASE images this is not passed to the compiler.
- **CHIP_XXXX**
  The CSL layer is written in a common file for all the variants of a SOC. This flag differentiates the variant we are compiling for, for e.g. - CHIP_DM648, and the CSL definitions for that variant appropriately gets defined for register base addresses, num of ports of a peripheral etc.

- **MCASP_INSTRUMENTATION_ENABLED**
  This flag is passed to the compiler to include the instrumentation code parts into the final image/lib of the program. This helps build the iRelease/iDebug versions of the image/lib with a common code base.

3 Run-Timer Interfaces/Integration Guide

This section discusses the DSP/BIOS McASP driver run-time interfaces that comprise the API classification & usage scenarios and the API specification itself in association with its data types and structure definitions.

3.1 Symbolic Constants and Enumerated Data types

This section summarizes all the symbolic constants specified as either #define macros and/or enumerated C data types. Described alongside the macro or enumeration is the semantics or interpretation of the same in terms of what value it stands for and what it means.

It is typical to classify the data types into logical groups and list them in alphabetical order for ease of use.

Table -1 DSP/BIOS McASP Driver Configuration defines

Defines	Description
PSP_MCASP_NUM_PORTS	Number of McASP ports on the SoC
PSP_MCASP_NUM_CHANS	Number of channels supported by the McASP driver

3.2 Data Structures

This section summarizes the entire user visible data structure elements pertaining to the DSP/BIOS McASP device driver configuration interfaces.

The file **"psp_mcasp.h"** has two data structures to configure the McASP device driver instance and the I/O channel used in the communication.

The "**PSP_mcaspDevParams**" contains the parameters used to configure the McASP IO mini device driver instance. The device parameters are explained in the Table - 2

Table -2 DSP/BIOS McASP IO Mini Driver's Device Configuration parameters

Parameters	Description
enablecache	Set to "TRUE" if the Submitted buffers are in cacheable memory
isDataBufferPayloadStructure	Set to "TRUE" if the Submitted buffers have payload information present or absent
mcaspHwSetup	Pointer to the PSP_McaspHwSetup and it contains McASP configurations.

The "**PSP_mcaspChanParams**" contains the parameters used to configure the McASP IOM device driver channel object instance. The channel parameters are explained in the Table ***-3***:

Table -3 Channel setup parameters for DSP/BIOS McASP IO mini driver

Parameters	Description
noOfSerRequested	Serializer requested by channel. Channel can ask for both.
indexOfSersRequested	Multi Serializer numbers requested by channel
mcaspSetup	Setup information for xmt/rcv sections of the McASP
isDmaDriven	This parameters determines whether channel operates in DMA mode
channelMode	The parameter informs Channel mode
wordWidth	The parameter informs the driver what is the width word (not slot)
userLoopJobBuffer	Buffer to be transferred when the loop job is running
userLoopJobLength	Number of bytes of the user loop job buffer for each serialiser
edmaHandle	Handle to the EDMA Driver

gblCbk	callback required when global error occurs must be callable directly from the ISR context

4 Driver API Classification

4.1 DSP/BIOS McASP Driver Initialization

DSP/BIOS call initialization routine which internally invokes "**MCASP_IOM_init ()**" API of the McASP IO mini driver, which in turn initializes the global data used by the McASP driver.

The initialization function sets the "inUse" field of both the McASP port object instance and the channel object instance to "FALSE" to make sure that the driver is not being used by any applications.

4.2 Driver Binding

The binding function for the McASP IOM driver "mcasp_mdBindDev ()" API is called by the DSP/BIOS initialization routine after MCASP_IOM_init () function is called.

The binding function should typically perform the following actions.

- Acquire the resources needed by the driver such as McASP and memory for port and channel object instances.
- Configure the McASP device to match the DSP data format mode of Audio Codec.

4.3 IOM Channel Creation

After the successful completion of McASP driver initialization and binding, the user can create an abstraction of the communication path between the application and McASP IOM driver. The channel instances are created through a call to mdCreateChan () function, which runs as a result of a call from GIO_create () API. The McASP driver allows user to create two channels (in input and output mode) for a given McASP device instance on the SoC. McASP IOM driver returns error status if the application attempts to create a channel in unsupported channel mode by the driver.

4.4 Channel Control

McASP IO Mini driver implements device specific control functionality, which may useful for application designers. Application may invoke the control functionality through calls to GIO_control ().The control functions supported by the McASP driver shall be useful when the any Audio Codec specific IOM driver uses the McASP IOM. McASP IOM driver supports the following control functionality.

4.4.1 PSP_CHAN_TIMEDOUT

- **SYNOPSIS**

- Int GIO_control(GIO_Handle gioChan, Uns cmd, Ptr args);

- **ARGUMENTS**
  - **gioChan –** Channel handle
  - **cmd -** PSP_CHAN_TIMEDOUT
  - **args -** NULL
- **RETURN VALUE**
  - IOM_COMPLETED
- **DESCRIPTION**

Resets the channel. Application can call this IOCTL in case when it encounters timeout.

- **LIMITATIONS/CONSTRAINTS**

None

4.4.2 PSP_CHAN_RESET

- **SYNOPSIS**
  - Int GIO_control(GIO_Handle gioChan, Uns cmd, Ptr args);
- **ARGUMENTS**
  - **gioChan –** Channel handle
  - **cmd -** PSP_CHAN_RESET
  - **args -** NULL
- **RETURN VALUE**
  - IOM_COMPLETED
- **DESCRIPTION**

Resets the channel and state machine.

- **LIMITATIONS/CONSTRAINTS**

None

4.4.3 PSP_MCASP_CNTRL_SET_FORMAT_CHAN

- **SYNOPSIS**
  - Int GIO_control(GIO_Handle gioChan, Uns cmd, Ptr args);
- **ARGUMENTS**

- **gioChan –** Channel handle
- **cmd -** PSP_MCASP_CNTRL_SET_FORMAT_CHAN
- **args -** address of PSP_McaspHwSetupData type structure

○ **RETURN VALUE**

- IOM_COMPLETED - if no error
- IOM_EBADARGS– if error

○ **DESCRIPTION**

This command is used when user wants to modify/format the channel configuration. User needs to provide the configuration along with the channel handle. State machine is put in to reset and receive/transmit section is configured using user provided configuration.

○ **LIMITATIONS/CONSTRAINTS**

None

4.4.4 PSP_MCASP_CNTRL_GET_FORMAT_CHAN

○ **SYNOPSIS**

- Int GIO_control(GIO_Handle gioChan, Uns cmd, Ptr args);

○ **ARGUMENTS**

- **gioChan –** Channel handle
- **cmd -** PSP_MCASP_CNTRL_GET_FORMAT_CHAN
- **args –** address of PSP_McaspHwSetupData type structure

○ **RETURN VALUE**

- IOM_COMPLETED - if no error
- IOM_EBADARGS– if error

○ **DESCRIPTION**

This command is used when user wants to modify/format the channel configuration. User needs to provide the configuration along with the channel handle. State machine is put in to reset and receive/transmit section is configured using user provided configuration.

○ **LIMITATIONS/CONSTRAINTS**

None

4.4.5 PSP_MCASP_CNTRL_SET_GBL_REGS

- **SYNOPSIS**
  - Int GIO_control(GIO_Handle gioChan, Uns cmd, Ptr args);
- **ARGUMENTS**
  - **gioChan –** Channel handle
  - **cmd -** PSP_MCASP_CNTRL_SET_GBL_REGS
  - **args –** address of PSP_McaspHwSetup type structure
- **RETURN VALUE**
  - IOM_COMPLETED - if no error
  - PSP_E_INVAL_STATE– if arg is NULL or devicehandle is NULL
  - PSP_MCASP_ABORTED – if driver is not able to configure McASP peripheral
- **DESCRIPTION**

This command will reset both receive and transmit and configure it with user provided configuration

- **LIMITATIONS/CONSTRAINTS**

None

4.4.6 PSP_MCASP_SET_DIT_MODE

- **SYNOPSIS**
  - Int GIO_control(GIO_Handle gioChan, Uns cmd, Ptr args);
- **ARGUMENTS**
  - **gioChan –** Channel handle
  - **cmd -** PSP_MCASP_SET_DIT_MODE
  - **args –** address of a value to configure in DITCTL register
- **RETURN VALUE**
  - IOM_COMPLETED - if no error
  - PSP_E_INVAL_STATE– if arg is NULL or devicehandle is NULL
- **DESCRIPTION**

This command configures DITCTL register of McASP with the user provided

- **LIMITATIONS/CONSTRAINTS**

None

4.4.7 PSP_MCASP_SET_DLB_MODE

- **SYNOPSIS**
  - Int GIO_control(GIO_Handle gioChan, Uns cmd, Ptr args);
- **ARGUMENTS**
  - **gioChan –** Channel handle
  - **cmd -** PSP_MCASP_SET_DLB_MODE
  - **args –** address of a variable ot boolean type to set/reset digital loopback mode.
- **RETURN VALUE**
  - IOM_COMPLETED - if no error
  - PSP_E_INVAL_STATE– if arg is NULL , devicehandle is NULL, there is a packet in receive/transmit queue, no. of serializer requested in receive and transmit channel are not same,
  - PSP_MCASP_ABORTED – if driver is unable to configure McASP in digital loop back mode
- **DESCRIPTION**

This command sets/resets McASP in digital loop back mode. If passed argument it TRUE, McASP is configured in loop back mode otherwise McASP is taken out of loop back mode.

- **LIMITATIONS/CONSTRAINTS**
  - TDM Slots should be set to 32 only (Refer to PSP_McaspHwSetupData Data structure for setting TDM Slots).
  - Both receive and transmit channels should be created with the same number of serializers.
  - Serializers used at the create time should be in pair of digital loop back. For example, Serializer – 0 is always connected to serializer – 1 in loop back mode. So channels should be created with these two serializers. Similarly if other serializes are used, they should be in similar pair.

4.4.8 PSP_MCASP_STOP_PORT

- **SYNOPSIS**
  - Int GIO_control(GIO_Handle gioChan, Uns cmd, Ptr args);

- **ARGUMENTS**
  - **gioChan –** Channel handle
  - **cmd -** PSP_MCASP_STOP_PORT
  - **args -** NULL
- **RETURN VALUE**
  - IOM_COMPLETED - if no error
  - IOM_EBADARGS– if stop command is already called successfully and again stop command is called to stop the record/playback
- **DESCRIPTION**

This command stops both state machine and frame sync generator. Record/Playback continues until the floating queue is not empty and after that state machine and frame sync generator is put to stop. To resume the record/playback process, user needs to issue PSP_MCASP_ START_PORT IOCTL.

- **LIMITATIONS/CONSTRAINTS**

None

4.4.9 PSP_MCASP_START_PORT:

- **SYNOPSIS**
  - Int GIO_control(GIO_Handle gioChan, Uns cmd, Ptr args);
- **ARGUMENTS**
  - **gioChan –** Channel handle
  - **cmd -** PSP_MCASP_START_PORT
  - **args -** NULL
- **REUTRN VALUE**
  - IOM_COMPLETED - if no error
  - IOM_EBADARGS - if error is reported by driver
- **DESCRIPTION**

Driver will start accepting requests from application for record/playback. If the requests are already queued in request queue, driver will start record/playback from there.

- **LIMITATIONS/CONSTRAINTS**

None

4.4.10 PSP_CTRL_McASP_MODIFY_LOOPJOB:

- **SYNOPSIS**
  - Int GIO_control(GIO_Handle gioChan, Uns cmd, Ptr args);
- **ARGUMENTS**
  - **gioChan –** Channel handle
  - **cmd -** PSP_CTRL_McASP_MODIFY_LOOPJOB
  - **args -** instance of PSP_mcaspChanParams structure. Please refer to section **Error! Reference source not found.** for the details of structure
- **RETURN VALUE**
  - IOM_COMPLETED - if no error
  - IOM_EBADARGS - if error is reported by driver
  - IOM_ENOTIMPL – if driver is in Interrupt mode
- **DESCRIPTION**

User defined loopjob buffer or internal loopjob will be transferred when the loop job is running. Loopjob will come in picture only when driver is starved for request from application. There are two types of loopjob, one is user provided loopjob where user provides the loopjob buffer and the other is internal loopjob. In internal loopjob, driver uses an internal buffer filled with 0.

- **LIMITATIONS/CONSTRAINTS**

None

4.4.11 PSP_CTRL_McASP_MUTE_ON:

- **SYNOPSIS**
  - Int GIO_control(GIO_Handle gioChan, Uns cmd, Ptr args);
- **ARGUMENTS**
  - **gioChan –** Channel handle
  - **cmd -** PSP_CTRL_McASP_MUTE_ON
  - **args -** NULL
- **RETURN VALUE**
  - IOM_COMPLETED – if no error
  - IOM_EBADARGS – if MUTE is already ON

- IOM_ENOTIMPL - if INPUT channel handle as an first argument

- **DESCRIPTION**

The request buffer is replaced with a mute buffer filled with 0 at the time of playback. When driver is in MUTE state, it will accept the requests from the application. Driver will playback the audio normally with only difference being the request buffer address is replaced with mute buffer. User will not hear any playback in the MUTE state.

- **LIMITATIONS/CONSTRAINTS**

None

4.4.12 PSP_CTRL_McASP_MUTE_OFF:

- **SYNOPSIS**
  - Int GIO_control(GIO_Handle gioChan, Uns cmd, Ptr args);
- **ARGUMENTS**
  - **gioChan –** Channel handle
  - **cmd -** PSP_CTRL_McASP_MUTE_OFF
  - **args -** NULL
- **RETURN VALUE**
  - IOM_COMPLETED – if no error
  - IOM_EBADARGS – if MUTE is already ON
  - IOM_ENOTIMPL - if INPUT channel handle as an first argument
- **DESCRIPTION**

The mute buffer is replaced with request buffer at the time of playback. This command will take driver back into a normal playback state.

- **LIMITATIONS/CONSTRAINTS**

None

4.4.13 PSP_MCASP_DEVICE_RESET:

- **SYNOPSIS**
  - Int GIO_control(GIO_Handle gioChan, Uns cmd, Ptr args);
- **ARGUMENTS**
  - **gioChan –** Channel handle
  - **cmd -** PSP_MCASP_DEVICE_RESET

- **args -** NULL

- **RETURN VALUE**
  - IOM_COMPLETED – if no error
  - IOM_EBADARGS – if Parameters are invalid
  - IOM_EBADMODE - if Registers are not reset properly

- **DESCRIPTION**

Driver will stop edma transfer and will stop the state machine. Once this command is issue successfully record and playback will not happen. All the requests in both receive and transmit queues are aborted and peripheral is configured with the default values.

- **LIMITATIONS/CONSTRAINTS**

Once this command is issue successfully record and playback will not happen. The steps to reconfigure device is provided in document BIOS_AUDIO_Driver_UserGuide.doc, section 3.3.4.1

4.4.14 PSP_MCASP_CNTRL_AMUTE:

- **SYNOPSIS**
  - Int GIO_control(GIO_Handle gioChan, Uns cmd, Ptr args);

- **ARGUMENTS**
  - **gioChan –** Channel handle
  - **cmd -** PSP_MCASP_CNTRL_AMUTE
  - **args -** NULL

- **RETURN VALUE**
  - IOM_COMPLETED – if no error
  - IOM_EBADMODE - if AMUTE register is not set properly
  - IOM_EBADARGS - if Parameters are invalid

- **DESCRIPTION**

The audio mute control register controls the McASP audio mute (AMUTE) output pin.

- **LIMITATIONS/CONSTRAINTS**

None

4.4.15 PSP_MCASP_GPIO_CONFIG:

- **SYNOPSIS**
  - Int GIO_control(GIO_Handle gioChan, Uns cmd, Ptr args);
- **ARGUMENTS**
  - **gioChan –** Channel handle
  - **cmd -** PSP_MCASP_GPIO_CONFIG
  - **args –** Uint32 variable, which has PDIR register value and this value will be stored in PDIR register.
- **RETURN VALUE**
  - IOM_COMPLETED – if no error
  - IOM_EBADMODE - if argument is NULL
  - IOM_EBADARGS - if Parameters are invalid
- **DESCRIPTION**

Argument value set in PDIR register, which Enables/Disables the GPIO mode, depends on argument value.

- **LIMITATIONS/CONSTRAINTS**

None

4.4.16 PSP_MCASP_QUERY_AMUTE:

- **SYNOPSIS**
  - Int GIO_control(GIO_Handle gioChan, Uns cmd, Ptr args);
- **ARGUMENTS**
  - **gioChan –** Channel handle
  - **cmd -** PSP_MCASP_QUERY_AMUTE
  - **args –** Uint32 variable, this variable will get AMUTE register value.
- **RETURN VALUE**
  - IOM_COMPLETED – if no error
  - IOM_EBADMODE - if AMUTE register is not read properly
- **DESCRIPTION**

It returns AMUTE register value.

- **LIMITATIONS/CONSTRAINTS**

None

4.4.17 PSP_McASP_PAUSE:

- **SYNOPSIS**
  - Int GIO_control(GIO_Handle gioChan, Uns cmd, Ptr args);
- **ARGUMENTS**
  - **gioChan –** Channel handle
  - **cmd -** PSP_McASP_PAUSE
  - **args -** NULL
- **RETURN VALUE**
  - IOM_COMPLETED – if no error
  - IOM_ EBADARGS - if Parameters are invalid
- **DESCRIPTION**

Pause will take maximum of two packets time to take an effect. When driver is in PAUSE state requests will be queued in driver's request queue. Instead of transferring actual recorded/played back data, driver will transfer a buffer filled with 0s to ensure no frame sync loss takes place.

- **LIMITATIONS/CONSTRAINTS**

None

4.4.18 PSP_McASP_RESUME:

- **SYNOPSIS**
  - Int GIO_control(GIO_Handle gioChan, Uns cmd, Ptr args);
- **ARGUMENTS**
  - **gioChan –** Channel handle
  - **cmd -** PSP_McASP_RESUME
  - **args -** NULL
- **RETURN VALUE**
  - IOM_ COMPLETED – if no error
  - IOM_EBADARGS - if PAUSED is not ON
- **DESCRIPTION**

Resumes record/playback operation from where it was PAUSED. If the requests are already queued, driver will take the first request in the queue and start the operation. If the there are not pending requests buffer filled with 0 is given for EDMA transfer until application submits a requests.

- **LIMITATIONS/CONSTRAINTS**

None

4.4.19 PSP_CTRL_RCV_GPIO_INPUT:

- **SYNOPSIS**
  - Int GIO_control(GIO_Handle gioChan, Uns cmd, Ptr args);
- **ARGUMENTS**
  - **gioChan –** Channel handle
  - **cmd -** PSP_CTRL_RCV_GPIO_INPUT
  - **args -** NULL
- **RETURN VALUE**
  - IOM_ COMPLETED - if no errors
- **DESCRIPTION**

This configures AFSR and ACLKR pins in GPIO input mode.

- **LIMITATIONS/CONSTRAINTS**

None

4.5 Transfer Requests Submission

Application can submit a transfer request using McASP class driver API GIO_submit () which in turn creates 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 McASP IOM driver must handle command code passed to it as part of the IOM_Packet structure. Depending on the command code, it either handles the code or returns the IOM_ENOTIMPL (not implemented) error code.
The currently supported McASP IOM mini-driver command codes are: IOM_READ, IOM_WRITE, IOM_ABORT, and IOM_FLUSH.

Note: Maximum size of single request by an application is 16K samples.

- **IOM_READ.** Drivers that support input channels must implement IOM_READ.

- **IOM_WRITE.** Drivers that support output channels must implement IOM_WRITE.

- **IOM_ABORT and IOM_FLUSH.** To abort or flush I/O requests already submitted, all I/O requests pending in the mini-driver must be completed and returned to the class driver. 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 cbFxn for the channel.

  - ❖ While aborting, all input and output requests are discarded.
  - ❖ While flushing, all pending output requests and processed normally and all pending input requests are discarded. This requires the processing of each IOM_Packet in the original order they were queued up to the channel. Until all the requests are flushed, the calling task i.e. the task from where IOM_FLUSH is called will wait till all the requests are flushed. Driver will not accept any packet requests until flush is complete.

4.6 ISR processing for transfer completion interrupt

The McASP IOM driver handles the EDMA transfer completion interrupt, which is raised by the EDMA after a transfer completion by the EDMA Transfer Controller. McASP IOM driver uses two McASP synchronized EDMA channels, one in input mode(McASP data reception) and one in output mode(McASP data transmission) for data transfer. When transfer EDMA completion interrupt occurs, the driver fetches the completed packet from the head of the IO packet queue for the channel and submits back the packet to the upper layer which intern calls the corresponding application callback.

4.7 IOM Channel Deletion

Application can free the resources held by the channel, if the channel is currently not in use, by calling GIO_delete () API. The corresponding "mdDeleteChan ()" function of the McASP IOM driver shall run from the application context and should de-allocate the specified channel object.

4.8 McASP IO Mini driver unbinding

The "mdUnBindDev ()" shall free resources allocated by the "mdBindDev ()" function.

4.9 McASP IOM driver's API Usage Scenarios/Integration Example

The McASP IOM driver is a generic one that may work across many codec drivers. The audio codec specific driver has to supply the device and channel specific configuration parameters to use the services of the McASP IOM driver.

Before data communication between an application and a device can begin, a channel instance handle must be returned to the application by a call to GIO_create () API. The channel handle represents a unique communication path between the application and McASP device driver. All subsequent operations that talk to the driver shall use this channel handle. A channel object typically maintains

data fields related to a channel's mode, I/O request queues, and possibly driver state information. Application should relinquish channel resources by deleting all channel instances when they are no longer needed through a call to GIO_delete (). Application shall call GIO_submit () APIs to submit an I/O request to driver. The device independent layer shall construct an I/O packet and submits the packet to the IOM layer to do the I/O operation. When a mini-driver completes its processing, usually in an ISR context, it calls its associated callback function to pass the IO packet back to the upper layer and the class driver in turn calls the application specified callback for that particular I/O request. The submit/callback function pair handles the passing of IO packets between the application and the McASP IOM layer of the driver. Before an IO packet is passed back to the device independent layer, the mini-driver must set the completion status field and the data size field in the IO Packet. This status value and size are returned to the application call that initially made the I/O request.

5 Driver Data Structures and Enumerations

This section describes the Initialization details, data structures, enumerations and macros for the List module.

5.1 Initialization Details

To use the Record and Play McASP device driver, a device entry must be added and configured in the DSP/BIOS configuration tool.

To have McASP device driver included in the application, corresponding TCI file have to be included in BIOS (TCF i.e. "dm648audio_mcasp.tci" for DM648) must be included in BIOS TCF file of the application Project.

The following are the device configuration settings required to use the McASP driver.

TCI Configuration Parameters	Description
initFxn - Init Function	AUDIO_AIC33_init1. This function initializes McASP driver's structures and enabling pin muxing.
fxnTable - Function Table Pointer	AUDIO_AIC33_FXNS. This is codec driver function table, which points to the Audio driver APIs.
fxnTableType - Function Table Type	IOM_Fxns
deviceId - Device Id	Specify McASP Device's ID. e.g. 0
params – Pointer to Port parameter	NULL
Device Global Data Pointer	NULL, not used by this driver

5.2 Data Structures

This section lists the data structures available in the McASP module.

5.2.1 PSP_Mcasp_DevParams

Detailed Description

Device Parameters to initialize Device Object.
This structure passes a handle to setup the McASP device registers. The application provides the McASPHwSetup structure to initialize the device.

Field Documentation

Structure Members	Description
enablecache	Datatype = Boolean , Submitted buffers are in cacheable memory
isDataBufferPayloadStructure	Datatype = Boolean , Submitted buffers have payload information present or absent
mcaspHwSetup	Datatype = PSP_McaspHwSetup*, Initial setup for the McASP

5.2.2 PSP_Mcasp_ChanParams

Detailed Description

Channel setup parameters.

Application specifies channel requirements and passes information to initialize the EDMA channel and interrupt handling for the channel. This is passed as a parameter to mdCreateChan.

Field Documentation

Structure Members	Description
noOfSerRequested	Datatype = Uint16, Serializer requested by channel. Channel can ask for all.
indexOfSersRequested[PSP_MCASP_MAX_SERS]	Datatype = Uint32, Feed index of required requested Serializers to be used in the data transfer.
mcaspSetup	Datatype = PSP_McaspHwSetupData *, Setup information for xmt/rcv sections of the McASP.
isDmaDriven	Datatype = Bool, This parameters determines whether channel operates in EDMA mode all EDMA parameters would be read only if this is TRUE.
channelMode	Datatype = PSP_McaspOpMode, Specifies mode of operation (TDM or DIT) for transmit channel.
wordWidth	Datatype = Uint32, The parameter informs the driver what is the width word (not slot).
userLoopJobBuffer	Datatype = Ptr, Buffer to be transferred when the loop job is running.
userLoopJobLength	Datatype = Uint16, Number of bytes of the userloopjob buffer for each serializer.
edmaHandle	Datatype = PSP_Handle, Handle to the EDMA Driver.
gblCbk	Datatype = PSP_mcaspGblCallback, callback required when global error occurs must be callable directly from the ISR context.
noOfChannels	Datatype = Uint32, No. of channels to be transmitted. This input is valid only for TDM in DSP mode of communication.

5.2.3 PSP_Mcasp_PktAddrPayload

Detailed Description

Structure passed through IOM_Packet addr field.
In order to pass data for the DIT Channel Status and User Data RAM registers, the addr field of the IOM_Packet is cast to PSP_Mcasp_PktAddrPayload. The addr field holds pointers to PSP_McaspChStatusRam and PSP_McaspUserDataRam in addition to the EDMA to be configured.

NOTE: The PSP_McaspUserDataRam and PSP_McaspChStatusRam fields will be read only for a transmit channel operating in DIT mode.

Field Documentation

Structure Members	Description
chStat	Datatype = PSP_McaspChStatusRam *, Channel Status RAM info.
userData	Datatype = PSP_McaspUserDataRam *, User Data RAM info.
writeDitParams	Datatype = Bool, Determines whether Channel Status and User Data get written in case of interrupt mode.
addr	Datatype = Uint32 *, Actual address to program EDMA with Address of data word if transactions are interrupt driven.

5.2.4 McASP Obj

Detailed Description

Channel Object.
This structure maintains the current channel state. It also holds information on DMA channel being used and holds the application callback function to be called in case of an interrupt. This structure is initialized by mdCreateChan and a pointer to this is passed down to all other channel related functions. Lifetime of the data structure is from its creation by mdCreateChan till it is invalidated by mdDeleteChan.

Field Documentation

Structure Members	Description
status	Datatype = PSP_McaspChannelStatus, Keeps tab of whether channel is already in use takes value of UNALLOCATED or ALLOCATED.
mode	Datatype = Uint16, mode for channel.
inUse	Datatype = Uint16, Channel is in use or idle.
channelOpMode	Datatype = PSP_mcaspMode, Mode of operation: Transmit or Receive.
portHandle	Datatype = struct McASPPortObj_t *, Pointer to McASP device Back pointer to device configuration structure.

ioHandle	Datatype = PSP_Handle, ddc channel Handle.
dataPacket	Datatype = IOM_Packet *, current active I/O packet.
cbFxn	Datatype = IOM_TiomCallback, Callback Function.
cbArg	Datatype = Ptr, Arguments for callback function.

5.2.5 PortObj

Detailed Description

Port Object.

This data structure holds the current device state. It also holds the handle to the McASP. The data structure is initialized during mcasp_mdBindDev, which is called during DSP- BIOS initialization, and is persistent until it is invalidated by mcasp_mdUnBindDev.

Field Documentation

Structure Members	Description
inUse	Datatype = Bool, Marks if port is currently in use
instNum	Datatype = Uint16 , Preserve instance number in port
isDataBufferPayloadStructure	Datatype = Bool, McASP handle for initial port configuration.
XmtObj	Datatype = McASPChannelObj , Holds transmit channel to the McASP.
RcvObj	Datatype = McASPChannelObj , Holds receive channel to the McASP.

5.2.6 PSP_McaspHwSetupGbl

Hardware setup global structure

Structure Members	Description
pfunc	Datatype = Uint32, Configure McASP to GPIO pins or McASP Pins uses PFUNC register.
pdir	Datatype = Uint32, Configure McASP pins to input mode or output mode uses PDIR register.
ctl	Datatype = Uint32, Configure McASP global control register uses GBLCTL register.
ditCtl	Datatype = Uint32, Configure McASP to DIT mode uses DITCTL register.
dlbMode	Datatype = Uint32, Configure McASP to loopback mode uses DLBEN register.
amute	Datatype = Uint32, Configure McASP to mute mode uses AMUTE register.

serSetup[PSP_MCASP_MAX_SERS]	Datatype = Uint32, Configure McASP all serializers to receive/transmit mode uses SRCTL register.

5.2.7 PSP_McaspHwSetupDataClk structure

Hardware setup data clock structure

Structure Members	Description
clkSetupClk	Datatype = Uint32, User can set ACLKCTL register to configure clock for internal/external, can set clock divider value.
clkSetupHiClk	Datatype = Uint32, User can set AHCLKCTL register to configure clock for internal/external, can set clock divider value.
clkChk	Datatype = Uint32, User can set clock check register uses CLKCHK register.

5.2.8 PSP_McaspHwSetupData structure

Hardware setup data structure

Structure Members	Description
mask	Datatype = Uint32, Configure mask values as per equirement uses R/X MASK register.
fmt	Datatype = Uint32 Configure McASP data transfer formats uses R/X FMT register Using this register user can configure following bits Slot size, bit delay, order bit, padding, rotation.
frSyncCtl	Datatype = Uint32, Configure McASP frame sync clocks uses R/X FMTCTL register Using this register user can configure following bits: Number of slots, receive frame sync duration, frame sync external/internal, frame sync polarity.
tdm	Datatype = Uint32, Configure McASP for particular TDM slots are active or not uses R/XTDM register.
intCtl	Datatype = Uint32, Configure McASP interrupt controller register uses R/X INTCTL register.
stat	Datatype = Uint32, Configure McASP error status register uses R/X STAT register.
evtCtl	Datatype = Uint32, Configure McASP event control register uses R/X EVTCTL register.
clk	Datatype = PSP_McaspHwSetupDataClk, Configure McASP clocks for eceive/transmit serializers.

5.2.9 PSP_McaspHwSetup structure

Hardware setup structure

Structure Members	Description
glb	Datatype = PSP_McaspHwSetupGbl, Value to be loaded in global control register uses GLBCTL register.
rx	Datatype = PSP_McaspHwSetupData, Receiver settings
tx	Datatype = PSP_McaspHwSetupData, Transmitter settings.
emu	Datatype = Uint32, Power down emulation mode params - PWRDEMU.

5.3 Enumerations

This section lists the enumerations available in the McASP module.

5.3.1 PSP_McaspChanMode

```
/** Enumeration for channel mode */
typedef enum {
  PSP_MCASP_CHAN_FREE    = 0,        /**< Channel not allocated */
  PSP_MCASP_CHAN_XMT_DIT = 1u,       /**< Transmit channel: DIT mode */
  PSP_MCASP_CHAN_XMT_TDM = 2u,       /**< Transmit channel: TDM mode */
  PSP_MCASP_CHAN_RCV     = 3u        /**< Receive channel */
} PSP_McaspChanMode;
```

5.3.2 PSP_McaspChannelStatus

```
/** brief  Enumeration for channel allocation status */
typedef enum {
  PSP_MCASP_UNALLOCATED = 0,         /**< Channel not allocated  */
  PSP_MCASP_ALLOCATED = 1            /**< Channel already in use */
} PSP_McaspChannelStatus;
```

5.3.3 PSP_McaspSerializerStatus

```
/** Enumeration for serializer status */
typedef enum PSP_McaspSerializerStatus_t
{
  PSP_MCASP_SER_FREE = (0u),         /**< Serializer not allocated */
  PSP_MCASP_SER_XMT  = (1u),         /**< Serializer configured to transmit */
```

```
  PSP_MCASP_SER_RCV  = (2u)          /**< Serializer configured to receive  */
} PSP_McaspSerializerStatus;
```

5.3.4 PSP_McaspSerializerNum

```
/** Enumeration for the serializer numbers */
typedef enum {
   SERIALIZER_0 = 0,              /** SRCTL0 */
   SERIALIZER_1 = 1,              /** SRCTL1 */
   SERIALIZER_2 = 2,              /** SRCTL2 */
   SERIALIZER_3 = 3,              /** SRCTL3 */
   SERIALIZER_4 = 4,              /** SRCTL4 */
   SERIALIZER_5 = 5,              /** SRCTL5 */
   SERIALIZER_6 = 6,              /** SRCTL6 */
   SERIALIZER_7 = 7,              /** SRCTL7 */
   SERIALIZER_8 = 8,              /** SRCTL8 */
   SERIALIZER_9 = 9,              /** SRCTL9 */
   SERIALIZER_10 = 10,            /** SRCTL10 */
   SERIALIZER_11 = 11,            /** SRCTL11 */
   SERIALIZER_12 = 12,            /** SRCTL12 */
   SERIALIZER_13 = 13,            /** SRCTL13 */
   SERIALIZER_14 = 14,            /** SRCTL14 */
   SERIALIZER_15 = 15             /** SRCTL15 */
} PSP_McaspSerializerNum;
```

5.3.5 Macros

#define CHANOBJINIT

Value:

```
{
   /* .status                    = */   PSP_MCASP_UNALLOCATED
   /* .mode                      = */   NULL
   /* .inUse                     = */   FALSE
   /* .channelMode               = */   PSP_MCASP_RX
   /* ._PortObj                  = */   NULL
   /* .DDC_mcaspChanHandle       = */   NULL
   /* .datapacket                = */   NULL
   /* .cbFxn                     = */   NULL
   /* .cbArg                     = */   NULL
}
```

#define PSP_MCASP_FLUSHED 2u
Data packet is flushed

#define PSP_MCASP_ABORTED 3u
Data packet is aborted

#define PSP_MCASP_INVALID 0xFFFF
Generic in/validate status

#define PSP_MCASP_DMA_ERROR (-15)
Interrupt due to: EDMA transfer error

#define PSP_MCASP_SYNC_ERROR (-16)
Interrupt due to: Transmit/Receive Sync Error

#define PSP_MCASP_XMT_UNDERRUN_ERROR (-17)
Interrupt due to: Transmit Buffer Underrun

#define PSP_MCASP_RCV_OVERRUN_ERROR (-18)
Interrupt due to: Receive Buffer Overrun

#define PSP_MCASP_RCV_BAD_CLK_ERROR (-19)
Interrupt due to Clock failure

#define PSP_MCASP_XMT_BAD_CLK_ERROR (-20)
Interrupt due to Clock failure

#define PSP_MCASP_LAST_SLOT 0x10
Interrupt due to: Last Slot

#define PSP_MCASP_DATA_RDY 0x20
Interrupt due to: Data Ready

#define PSP_MCASP_STAFRM 0x40
Interrupt due to: Start of Frame

#define PSP_MCASP_DMAERR 0x80
Interrupt due to: DMA channel error

#define PSP_MCASP_NUM_CHANS 2
Port can have two channels, one each for IOM_INPUT and IOM_OUTPUT

#define PSP_MCASP_NUM_PORTS 2
Trinity supports only one McASP instance

#define PSP_MCASP0_NUM_SERS 5
The Mcasp0 has Ten serializers

#define PSP_MCASP1_NUM_SERS 5

The Mcasp1 has Eight serializers

#define PSP_MCASP_SM_RESET 0
Reset value of xmt/rcv state m/c and frame sync

#define PSP_MCASP_TDM_MODE 0
Mode for channel operation

#define PSP_MCASP_TWO_PKTS_QUEUED 2
When xmt is taken out of reset, it needs two words to service it. This is used to keep tally of this requirement.

#define PSP_MCASP_CNTRL_SET_FORMAT_CHAN 130
Control command: Format channel

#define PSP_MCASP_CNTRL_GET_FORMAT_CHAN 131
Control command: Format channel

#define PSP_MCASP_CNTRL_SET_GBL_REGS 132
Control command: Set registers affecting McASP device.

#define PSP_MCASP_SET_DLB_MODE 133
Control command: Set digital loopback mode.

#define PSP_MCASP_SET_DIT_MODE 134
Control command: Set DIT mode

#define PSP_MCASP_CNTRL_AMUTE 137
Control command: Set it size

#define PSP_MCASP_START_PORT 141
Control command: Start McASP serial port in Dynamic mode

#define PSP_MCASP_STOP_PORT 142
Control command: Start McASP serial port in Dynamic mode

#define PSP_MCASP_WORDLEN_8 8
Word length is 8

#define PSP_MCASP_WORDLEN_12 12
Word length is 12

#define PSP_MCASP_WORDLEN_16 16
Word length is 16

#define PSP_MCASP_WORDLEN_20 20
Word length is 20

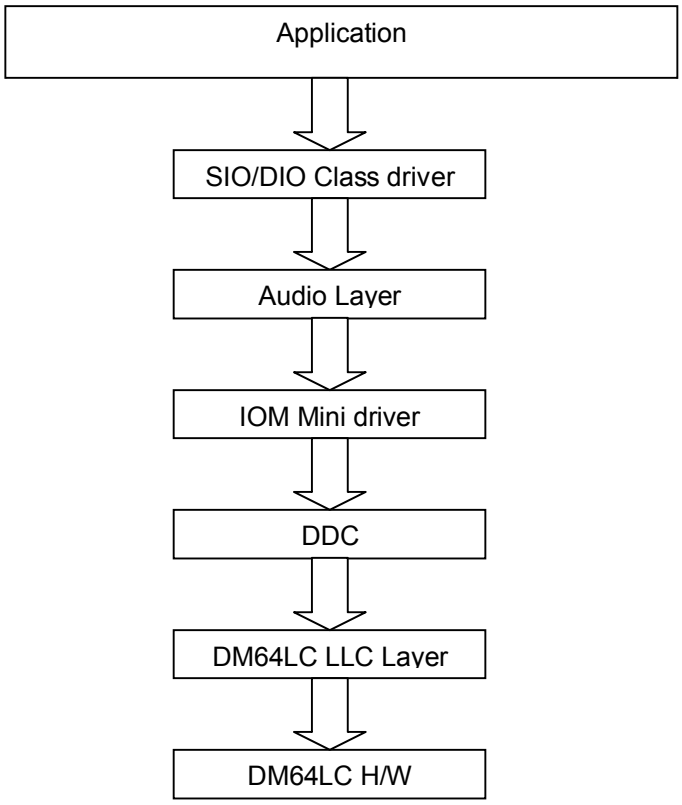
#define PSP_MCASP_WORDLEN_24 Word length is 24	24
#define PSP_MCASP_WORDLEN_32 Word length is 32	32
#define PSP_MCASP_XERR Error code	0x100

6 Porting Guide

This section describes porting of McASP driver on different TI platforms.

6.1 Porting Description

The figure below shows MCASP device driver architecture and changes those are required at the driver layers while porting MCASP device driver to any other Platform.



Driver Architecture

There will not be any change required in the SIO/DIO Layer, IOM Layer, DDA Layer and DDC layer while porting MCASP device driver on any TI platform. This Layer will be used as-is.

Audio Layer

Audio layer is designed in such a way that it can be used with both McASP driver as well as McASP driver. The job of the audio layer is:

1. To perform the pinmux settings specific to the platform
2. To configure external audio codec to work with McASP driver. Here AIC33 audio codec is used.

While porting of McASP driver for other platform, above mentioned two things one needs to take care. One needs to perform the pinmux setting as per the new platform and configuration of audio codec based on the audio codec that is being used with McASP audio driver.

LLC Layer

This layer provides the abstraction to the Driver core on different platforms. This layer is specific to a specific platform. Mainly this layer should be having register overlaying, macro definitions. If not register overlaying is used then this layer will have low-level APIs to communicate with the hardware
This layer should be having an as-is map of the peripheral device registers in the processor's memory map. Peripheral device registers map may differ from one platform to other. This change needs to be incorporated while porting driver code from once platform to another platform at LLC layer.

7 McASP as a Standalone Driver

McASP driver can work as a standalone driver. In that case user needs to take care device creation, device initialization, device pinmuxing, channel creation, receive and transmit requests at the desired time, handling of application layer callback (if desired) and channel deletion.

7.1 Initialization of McASP driver

To use the McASP device driver, a device entry must be added and configured in the DSP/BIOS configuration tool.

The following are the device configuration settings required to use the McASP driver. Note: This has to be done for all of the required driver instances. Following table illustrates device creation in tcf file for GIO mode class driver. For SIO mode class driver please refer to user guide of mini driver.

TCF Configuration Parameters	Description
initFxn - Init Function	"MCASP_IOM_init()" - this function is doing initialization of McASP ports. In TCF file pass pointer of this function as an argument.
fxnTable - Function Table Pointer	McASP_IOM_FXNS. This is a global variable which points to the McASP driver APIs.
fxnTableType - Function Table Type	IOM_Fxns
deviceId - Device Id	Specify McASP port number. For example '0'.
params – Pointer to Port parameter	An optional pointer to an object of type PSP_mcaspDevParams as defined in the header file psp_mcasp.h. This pointer will point to a device parameter structure. In BIOS TCF files, this structure object is passed as an argument. Application should declare and initialize the structure object properly. Note: More information of PSP_mcaspDevParams structure's fields are describing in section 3.2.
Device Global Data Pointer	N/A, not used by this driver

7.1.1 Device Pinmuxing

Device pinmuxing is a part of McASP peripheral initialization. User needs to carry out pinmuxing to enable McASP device. Make sure pinmuxing is done before channel creation.

7.2 Channel Creation

McASP channels can be created either in TDM (Time Division Multiplex) mode or DIT (Digital Audio Interface Transmit) mode. Please configure channel parameters as follows:

7.2.1 Channel Parameter Configuration in TDM mode

Channel Parameters	Description	Field value
noOfSerRequested	Datatype = Uint16, Serializer requested by channel. Channel can ask for all.	If user wants to use one serializer or more serializers for record/playback operation then make this field value equal to 1 or number of serializers that user uses in operation respectively.
indexOfSersRequested	Datatype = Uint32, Feed index of required requested Serializers to be used in the data transfer.	If user uses serializer 1 for record/playback operation then this field value is SERIALIZER_1. In case of multiple serializers used by user then put serializer number in this array's element.
mcaspSetup	Datatype = PSP_McaspHwSetupData *, Setup information for xmt/rcv sections of the McASP.	pointer of PSP_McaspHwSetupData structure's object. This object contain initial register configuration of rcv/xmt channel. Note: Refer McASP's device user guide for receive and transmit channel's register configuration in TDM mode
isDmaDriven	Datatype = Bool, This parameters determines whether channel operates in EDMA mode all EDMA parameters would be read only if this is TRUE.	If user wants to use DMA mode for operation then this field value is TRUE. Note: McASP driver support only DMA mode, Interrupt mode is not supported.
channelMode	Datatype = PSP_McaspOpMode, Specifies mode of operation (TDM or DIT) for transmit channel.	PSP_MCASP_TDM_MODE
wordWidth	Datatype = Uint32, The parameter informs the driver what is the width word (not slot).	If user wants 32 bits word width for operation then this field value is PSP_MCASP_WORDLEN_32
userLoopJobBuffer	Datatype = Ptr, Buffer to be transferred when the loop job is running.	pointer of loop job buffer
userLoopJobLength	Datatype = Uint16, Number of samples of the userloopjob buffer for each serializer.	length of loop job buffer
edmaHandle	Datatype = PSP_Handle, Handle to the EDMA Driver.	Handle of the EDMA Driver
gblCbk	Datatype = PSP_mcaspGblCallback, callback required when global error occurs	Pointer of global callback function. This function called when one of the below error occurred:

	must be callable directly from the ISR context.	1 Frame sync error 2 Transmit underrun error 3 Receive overrun error 4 DMA error
noOfChannels	Datatype = Uint32, No. of channels to be transmitted. This input is valid only for TDM in DSP mode of communication.	If user wants to use 1 channel for operation then this field value is 1u.

In TDM mode payload information is not present. This information passed in *PSP_mcaspDevParams* at initialization time.

The following is the *PSP_mcaspDevParams* structure information:

Structure Members	Description	Field value
enablecache	Datatype = Bool, This parameter determines whether submitted buffers are in cacheable memory or not	If user wants to enable cache for submitted buffers then this field value is TRUE, if user not wants to enable cache for submitted buffers then this field value is FALSE.
isDataBufferPayloadStructure	Datatype = Bool, This parameter determines whether payload information present or absent	FALSE
mcaspHwSetup	Datatype = PSP_McaspHwSetup *, setup information for McASP configurations.	Pointer of PSP_McaspHwSetup structur's object. This object contain initial register configuration of McASP.

7.2.2 Channel Parameter Configuration in DIT Mode

Channel Parameters	Description	Field value
noOfSerRequested	Datatype = Uint16, Serializer requested by channel. Channel can ask for all.	If user wants to use one serializer or more serializers for record/playback operation then make this field value equal to 1 or number of serializers that user uses in operation respectively.
indexOfSersRequested	Datatype = Uint32, Feed index of required requested Serializers to be used in the data transfer.	If user uses serializer 1 for record/playback operation then this field value is SERIALIZER_1. In case of multiple serializers used by user then put serializer number in this array's element.

mcaspSetup	Datatype = PSP_McaspHwSetupData *, Setup information for xmt/rcv sections of the McASP.	Pointer of PSP_McaspHwSetupData structure's object. This object contain initial register configuration of respective rcv/xmt channel. Note: Refer McASP's device user guide for receive and transmit channel's register configuration in DIT mode
isDmaDriven	Datatype = Bool, This parameters determines whether channel operates in EDMA mode all EDMA parameters would be read only if this is TRUE.	If user wants to use DMA mode for operation then this field value is TRUE. Note: McASP driver support only DMA mode, Interrupt mode is not supported.
channelMode	Datatype = PSP_McaspOpMode, Specifies mode of operation (TDM or DIT) for transmit channel.	PSP_MCASP_DIT_MODE
wordWidth	Datatype = Uint32, The parameter informs the driver what is the width word (not slot).	If user wants 16 bits word width for operation then this field value is PSP_MCASP_WORDLEN_16
userLoopJobBuffer	Datatype = Ptr, Buffer to be transferred when the loop job is running.	pointer of loop job buffer
userLoopJobLength	Datatype = Uint16, Number of samples of the userloopjob buffer for each serializer.	length of loop job buffer
edmaHandle	Datatype = PSP_Handle, Handle to the EDMA Driver.	Handle to the EDMA Driver
gblCbk	Datatype = PSP_mcaspGblCallback, callback required when global error occurs must be callable directly from the ISR context.	Pointer of global callback function. This function called when one of the below error occurred: 1 Frame sync error 2 Transmit underrun error 3 Receive overrun error 4 DMA error
noOfChannels	Datatype = Uint32, No. of channels to be transmitted. This input is valid only for TDM in DSP mode of communication.	If user wants to use 2 channels for operation then this field value is 2u.

In DIT mode payload information is present. This information can be passed during IO calls using *PSP_Mcasp_PktAddrPayload structure. This structure would be sent instead of the data buffer pointer in submit API. In applications (normal TDM) where we need only a flat data buffer need to be sent to driver, it can be sent so using data buffer pointer in the submit API.* The selection between these two modes of data handling mechanism (data buffer pointer of submit API interpreted as flat buffer or *PSP_Mcasp_PktAddrPayload structure) can be decided by* a member in *PSP_mcaspDevParams* at initialization time.

The following is the *PSP_mcaspDevParams* structure information:

Structure Members	Description	Field value
enablecache	Datatype = Bool, This parameter determines whether submitted buffers are in cacheable memory or not	If user wants to enable cache for submitted buffers then this field value is TRUE, if user not wants to enable cache for submitted buffers then this field value is FALSE.
isDataBufferPayloadStructure	Datatype = Bool, This parameter determines whether data buffer pointer of submit API isinterpreted as flat buffer or *PSP_Mcasp_PktAddrPayload structure*	TRUE
mcaspHwSetup	Datatype = PSP_McaspHwSetup *, setup information for McASP configurations.	Pointer of PSP_McaspHwSetup structur's object. This object contain initial register configuration of McASP.

The following is the *PSP_McaspChStatusRam* structure information:

Structure Members	Description	Field value
chStatusLeft	Datatype = Uint32, Left channel status register is an array of 6 elements.	24 bytes of left channel status registers value
chStatusRight	Datatype = Uint32, Right channel status register is an array of 6 elements	24 bytes of right channel status registers value

The following is the *PSP_McaspUserDataRam* structure information:

Structure Members	Description	Field value
userDataLeft	Datatype = Uint32, Left channel user data register is an array of 6 elements	24 bytes of left channel user data registers value
userDataRight	Datatype = Uint32, Right channel user data register is an array of 6 elements	24 bytes of right channel user data registers value

The following is the *PSP_Mcasp_PktAddrPayload* structure information:

Structure Members	Description	Field value
chStat	Datatype = *PSP_McaspChStatusRam* *, Channel Status RAM info.	Pointer of PSP_McaspChStatusRam structure's object.
userData	Datatype = *PSP_McaspUserDataRam* *, User Data RAM info.	Pointer of PSP_McaspUserDataRam structure's object.
writeDitParams	Datatype = Bool, Determines whether Channel Status and User Data information is present or absent.	If user wants to pass channel status information and user data information then TRUE in this field otherwise FALSE in this field.
addr	Datatype = Uint32 *, Actual address to program EDMA with Address of data word if transactions are interrupt driven.	Buffer address.

Please note:

Note1

In case of transfers with SPDIF meta data - user data, channel status for SPDIF frame, it is suggested to have each IOP to cater one spdif block and send each IOP (submit) one by one. Sending submits one by one is mandatory as otherwise there is a possibility of user data and channel status getting messed up between IO requests. However for the transfer that does not involve the user data, channel status we could have any transfer size per IO request and IO request can be queued to the driver.

Note2

In DIT mode the wordlength should be 32. As the MCASP takes 24bits of data from the buffer to fill up spdif data field and as slot size is(must be) set to 32bits this(SPDIF) is mode, it is mandatory that for each sub frame of spdif (left channel or right channel) application has to provide 32bits of data.

That means

1) Buffer to the driver in this mode should be of Uint32 type and each double word would cater to one subframe. Hence users should send different channels (left/right) of data in different double words.

2) Set the wordwidth parameter of chanparams to PSP_MCASP_WORDLEN_32.

8 Example Application

This section describes the example application that is included in the package. This sample application can be run as is for quick demonstration, but the user will benefit most by using these samples as sample source code in developing new applications.

8.1 The McASP Sample Application for DM648/C6452

8.1.1 Introduction

The sample application is a representative test program. Initialization of McASP driver is done by calling initialization function from BIOS. The sample application will perform audio record and playback in a continuous loop. It will record the audio from the line-in port and playback the audio through line-out/headphone-out port.

8.1.2 Directory structure for sample application

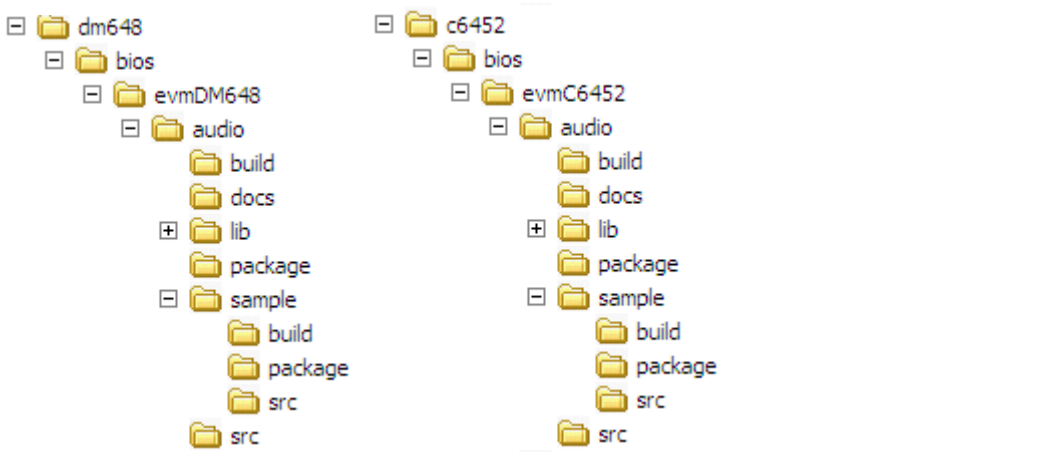


Figure 2. McASP sample directory structure for DM648 and C6452

Top level folder shown in the above figure contains header and tci files required specifically for sample application along with XDC packaging files(package.bld and package.xdc)

build: This subfolder in the sample folder contains project for sample appliaction.

src: This folder contains mcasp sample application source files. It also contains header files related to mcasp driver that are used by the sample application if any.

Package: This folder contains files generated by XDC tool

8.1.3 Building the Application

The sample application for DM648/C6452 is located in the pspdrivers_\packages\ti\sdo\pspdrivers\system\dm648\bios\evmDM648\audio\sample for DM648 and pspdrivers_\packages\ti\sdo\pspdrivers\system\c6452\bios\evmC6452\audio\sample for C6452 folder. The sample can be rebuilt directly from its project file using Code Composer studio.

Please follow below steps to build sample application:

- Open CCS 3.3 setup. Import proper CCS configuration file. Set the proper CCS Gel file (Refer DM648/C6452_BIOS_PSP_Release_Notes.doc for details). Click on “Save & Quit” button and exit the setup.
- Open pspdrivers_\packages\ti\sdo\pspdrivers\system\dm648\bios\evmDM648\audio\sample\build\dm648_evm_audio_st_sample.pjt for DM648 or pspdrivers_\packages\ti\sdo\pspdrivers\system\c6452\bios\evmC6452\audio\sample\build\c6452_evm_audio_st_sample.pjt for C6452.
- Compile this project using Project->Build
- By default sample application is in FOUR_TO_ONE mode. This implies, application records using 4 serializers and playbacks using one serializer.

8.1.4 Loading the Application

The sample application is loaded and executed via Code composer studio. It is a good idea to reset the board before loading Code Composer.

At the audio layer McASP driver is configured with following values at the bind time.

```
static LLC_McaspHwSetup appMcaspHwSetup = {        \
   {
     /* .pfunc    = */ 0x00000000,          /*mcasp or GPIO pin-will be set by driver from chanparams*/
     /* .pdir     = */ 0x00000001,          /*Direction of pin -will be set by driver */
     /* .gblctl   = */ 0x00000000,          /*control clk, hclk, statemachine - reset and release - driver ctrl*/
     /* .ditctl   = */ 0x00000000,          /*DIT mode seting*/
     /* .dlbctl   = */ 0x00000000,          /*loop back mode setting*/
     /* .amute    = */ 0x00000000           /*Amute setting*/
     /* .srctl0   = */ 0x00000000,          /* init serialiser*/
     /* .srctl1   = */ 0x00000000           /* init serialiser*/
    },
     {
        /* .rmask    = */ 0xFFFFFFFF,
        /* .rfmt     = */ 0x00000000,
        /* .afsrctl  = */ 0x00000000,
        /* .rtdm     = */ 0x00000001,
        /* .rintctl  = */ 0x00000000,
        /* .rstat    = */ 0x000001FF,          /*reset any existing status bits*/
        /* .revtctl  = */ 0x00000000,
        {
          /* .aclkrctl  = */ 0x00000003,     /* \div = 1, clk = internal*/
          /* .ahclkrctl = */ 0x00008046,     /* div = 63, no inversion before divider ,Internal*/
          /* .rclkchk   = */ 0x00000000
        },
    },
    {
        /* .xmask    = */ 0xFFFFFFFF,
        /* .xfmt     = */ 0x00000000,
        /* .afsxctl  = */ 0x00000002,
        /* .xtdm     = */ 0x00000001,
```

```
        /* .xintctl  = */ 0x00000000,
        /* .xstat    = */ 0x000001FF,          /*reset any existing status bits*/
        /* .xevtctl  = */ 0x00000000,
        {
          /* .aclkxctl  = */ 0x00000023,      /* \div = 1, clk = internal*/
          /* .ahclkxctl = */ 0x00008046,      /* div = 46, no inversion before divider ,Internal*/
          /* .xclkchk   = */ 0x00000000
        },
    },
        0x00000001                            /*emu free*/
} ;
```

At the channel create time, McASP driver is configured for receive/record and transmit/playback operation as per following configuration

```
LLC_McaspHwSetupData mcaspXmtSetup = { \
      /* .xmask    = */ 0xFFFFFFFF,         /*all the data bits are used*/
      /* .xfmt     = */ 0x000080F0,    /*no right rotation,
                                         DMA access,
                                         slot size = 32bits,
                                         pad with 0th bit from data,(currently this option disabled by next field)
                                         pad extra bits with 0,
                                         LSB first,
                                         0 bit delay from between fsync and data*/
      /* .afsxctl  = */ 0x00000000,    /*burst mode,
                                        frame sync pulse width - 1 clk bit
                                        clk is internal
                                        Rising edge is validated*/

      /* .xtdm     = */ 0x00000003,    /*n-th (2 here)TDM slot will be accounted - for burst mode no use*/
      /* .xintctl  = */ 0x00000000,    /*reset any existing status bits*/
      /* .xstat    = */ 0x000001FF,    /*reset any existing status bits*/
      /* .xevtctl  = */ 0x00000000,    /*DMA or INT mode*/
      {
          /* .aclkxctl  = */ 0x00000000,    /* \div = 1, clk = External*/
          /* .ahclkxctl = */ 0x0000003F,   /* div = 63, no inversion before divider ,External*/
          /* .xclkchk   = */ 0x00000000
      },
};
```

```
LLC_McaspHwSetupData mcaspRcvSetup = { \
      /* .rmask    = */ 0xFFFFFFFF,          /*all the data bits are used*/
      /* .rfmt     = */ 0x000080F0,    /*no right rotation,
                                         DMA access,,
                                         slot size = 16bits,
                                         pad with 0th bit from data,(currently this option disabled by next field)
                                         pad extra bits with 0,
                                         LSB first,
                                         0 bit delay from between fsync and data*/
      /* .afsrctl  = */ 0x00000000,    /*2slot tdm,
                                        frame sync pulse width - 1 clk bit
                                        clk is internal
```

```
                                        Rising edge is validated*/
    /* .rtdm     = */ 0x00000003,          /*n-th (2 here)TDM slot will be accounted*/
    /* .rintctl  = */ 0x00000000,          /*This will be filled up by driver internally*/
    /* .rstat    = */ 0x000001FF,          /*reset any existing status bits*/
    /* .revtctl  = */ 0x00000000,          /*DMA or INT mode*/
    {
        /* .aclkrctl  = */ 0x00000000,     /* \div = 1, clk = External*/
        /* .ahclkrctl = */ 0x0000003F,     /* div = 63, no inversion before divider ,External*/
        /* .rclkchk   = */ 0x00000000
    }
} ;
```

8.2 The McASP Sample Application for DM6437/C6424

8.2.1 Introduction

The sample application will perform audio record and playback in a continuous loop. It will record the audio from the line-in port and playback the audio through line-out/headphone-out port.

8.2.2 Directory structure for sample application

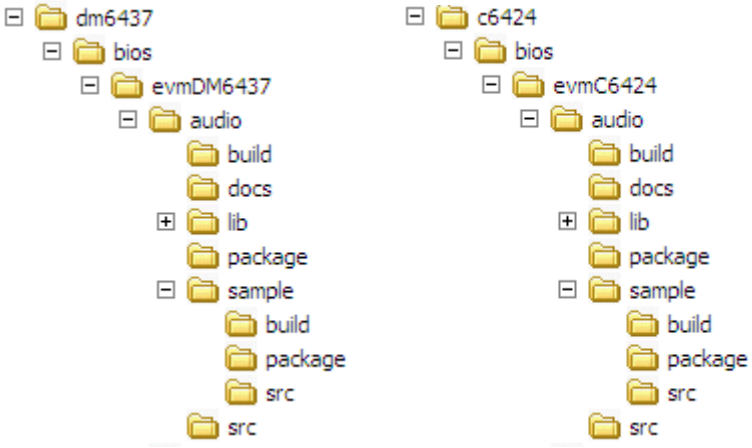


Figure 3. McASP sample directory structure for DM6437 and C6424

Top level folder shown in the above figure contains header and tci files required specifically for sample application along with XDC packaging files(package.bld and package.xdc)

build: This subfolder in the sample folder contains project for sample appliaction.

src: This folder contains mcasp sample application source files. It also contains header files related to mcasp driver that are used by the sample application if any.

Package: This folder contains files generated by XDC tool

8.2.3 Building the Application

The sample application for DM6437 is located in the \pspdrivers\system\DM6437\bios\ evmDM6437\ audio\sample folder and the sample application for C6424 is located in the \pspdrivers\system\C6424\bios\ evm6424\audio\sample. The sample can be rebuilt directly from its project file using Code Composer studio.

8.2.4 Loading the Applicaiton

The sample application is loaded and executed via Code composer studio. It is a good idea to reset the board before loading Code Composer.

At the audio layer McASP driver is configured with following values at the bind time.

```
static PSP_McaspHwSetup appMcaspHwSetup = {       \
        {
                /* .pfunc    = */ 0x00000000,              /*mcasp or GPIO pin-will be set by driver from
chanparams*/
                /* .pdir      = */ 0x00000001,                  /*Direction of pin -will be set by driver from
chanparams*/
                /* .gblctl   = */ 0x00000000,             /*control clk, hclk, statemachine - reset and release
- driver ctrl*/
                /* .ditctl   = */ 0x00000000,            /*DIT mode seting*/
                /* .dlbctl   = */ 0x00000000,             /*loop back mode setting*/
                /* .amute    = */ 0x00000000,              /*Amute setting*/
                /* .srctl0   = */ 0x00000000,                   /* init serialiser*/
                /* .srctl1   = */ 0x00000000             /* init serialiser*/
        },
        {
                /* .rmask    = */ 0xFFFFFFFF,
                /* .rfmt     = */ 0x00000000,
                /* .afsrctl  = */ 0x00000000,
                /* .rtdm     = */ 0x00000001,
                /* .rintctl  = */ 0x00000000,
                /* .rstat    = */ 0x000001FF,               /*reset any existing status bits*/
                /* .revtctl  = */ 0x00000000,
                {
                        /* .aclkrctl  = */ 0x00000023,             /* \div = 1, clk = internal*/
                        /* .ahclkrctl = */ 0x00008046,             /* div = 63, no inversion before divider
,Internal*/
                        /* .rclkchk   = */ 0x00000000
                },
        },
        {
                /* .xmask    = */ 0xFFFFFFFF,
                /* .xfmt     = */ 0x00000000,
                /* .afsxctl  = */ 0x00000002,
                /* .xtdm     = */ 0x00000001,
                /* .xintctl  = */ 0x00000000,
                /* .xstat    = */ 0x000001FF,              /*reset any existing status bits*/
                /* .xevtctl  = */ 0x00000000,
                {
                        /* .aclkxctl  = */ 0x00000023,             /* \div = 1, clk = internal*/
                        /* .ahclkxctl = */ 0x00008046,             /* div = 46, no inversion before divider
,Internal*/
                        /* .xclkchk   = */ 0x00000000
                },
        },
        0x00000001          /*emu free*/
} ;
```

At the channel create time, McASP driver is configured for receive/record and transmit/playback operation as per following configuration

```
PSP_McaspHwSetupData mcaspXmtSetup = { \
                    /* .xmask    = */ 0xFFFFFFFF,
                    /* .xfmt     = */ 0x000080F0,
                    /* .afsxctl  = */ 0x00000000,
                    /* .xtdm     = */ 0x00000003,
                    /* .xintctl  = */ 0x00000000,
                    /* .xstat    = */ 0x000001FF,
                    /* .xevtctl  = */ 0x00000000,
                    {
                              /* .aclkxctl  = */ 0x00000000,
                              /* .ahclkxctl = */ 0x0000003F,
                              /* .xclkchk   = */ 0x00000000
                    },

};
```

```
PSP_McaspHwSetupData mcaspRcvSetup = { \
                    /* .rmask    = */ 0xFFFFFFFF,
                    /* .rfmt     = */ 0x000080F0,

                    /* .afsrctl  = */ 0x00000000,

                    /* .rtdm     = */ 0x00000003,
                    /* .rintctl  = */ 0x00000000,
                    /* .rstat    = */ 0x000001FF,
                    /* .revtctl  = */ 0x00000000,
                    {
                              /* .aclkrctl  = */ 0x00000000,
                              /* .ahclkrctl = */ 0x0000003F,
                              /* .rclkchk   = */ 0x00000000
                    }
} ;
```

8.2.5 Pragma directives used in the Application

The prdInputBuf and prdOutputBuf are used by the application and need to be cache aligned at 128 bytes.