DM648/C6452 Audio Driver

USER'S GUIDE

Document Revision History

Rev No	Author(s)	Revision History	Date	Approval(s)
1.0	Pratik Joshi	Initial Draft	February 2nd, 2007	Initial Draft
1.4	Narendra	Updated for release 0.4	March 23, 2007	
1.8	Pratik Joshi	Updated for release 0.6	May 5th, 2007	
1.9	Pratik Joshi	Change Audio API description as per Video user guide	May 8th, 2007	
2.0	Pratik Joshi	Updated for release 1.10.00.XX	June 15, 2007	
2.1	Pratik Joshi	Added DM648 EVM limitation section	July 06,2007	
2.2	Pratik Joshi	Added MACRO and its description for ONE_TO_ONE and FOUR_TO_ONE test	Aug 08, 2007	
2.3	Nagarjuna K	Updated for both DM648 and C6452	January 18, 2008	

TABLE OF CONTENTS

TABLE OF FIGURES

1 Introduction

This document is the reference guide for the audio driver and it explains how to configure and use the driver.

DSP/BIOS applications use the driver typically through either GIO or SIO/DIO class drivers. For more information on the DSP/BIOS device driver model and the GIO class driver, refer to the References section of this document.

1.1 Terms & Abbreviations

Term/Abbreviation	Description
API	Application Programmers Interface
IP	Intellectual Property
EDMA	Enhanced Direct Memory Access Controller
IOM	I/O Mini Driver Model

1.2 References

1.3 S/W Support

This audio device driver has been developed for the DSP/BIOS operating system using the TI supplied Chip Support Library. For more details on the version numbers refer to the release notes in the root of the installation.

1.4 Supported Services and Features

The DSP/BIOS Audio driver provides the following functional services and features:

- Multi-instantiable and re-entrant safe driver.
- Designed for (but not limited to) use with codec drivers.
- Features supported by the Audio driver, which can be directly used by the Audio codec driver are:
  - Supports run-time Start/Stop of the Audio Play and Record operation
  - Supports Pause-Resume feature for Audio Playback operation when integrated with the audio codec specific driver
  - Supports Mute ON/OFF feature for Audio Playback operation when integrated with the audio codec specific driver.

1.5 System Requirements

The ***DM648/C6452 Audio Driver User Guide*** is supported on platforms characterized by the following requirements

Hardware:

- Target Board: DM648/C6452 EVM Board
- Emulation Setup: XDS 510 USB or XDS 560 PCI Emulator

Software:

- Code generation tools: CCS 3.3.38.2
- Operating System: DSP/BIOS 5.31.08
- Code generation tools 6.0.8
- XDC 2.94.01.03

2 Installation Guide

2.1 Component Folder

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

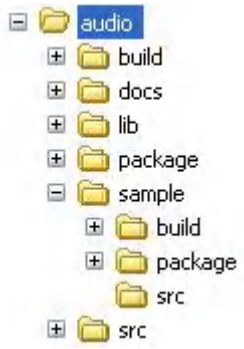


Figure 1. Audio Driver Directory Structure

- **build :** This folder contains the driver library project and driver library. The driver library shall be included in the application where Audio driver have to be used.
- **docs :** This folder contains Release notes document and User Guide Document.
    - 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.
- **package :** This folder contains RTSC packaging related files.
- **lib:** This folder contains the audio driver libraries for debug and release mode inside lib folder in path **DM648/C6452\Debug** and **DM648/C6452\Release** respectively
- **sample :** This folder contains the sample applications that demonstrates the use of the driver. This sample applications demonstrates basic features of the driver. User can use this sample application as reference to make their applications. In addition to that this folder also contains the .tci files for McASP instance.
    - **build :** This subfolder in sample folder contains sample application library as well as stand alone sample application project. User can run specific sample application by adding sample applications library in executable project.
    - **src:** This folder contains the sample applications, one for stand alone audio project and one for combo

- **src:** This folder contains the source files of Audio driver.

2.2 Build

This section describes for each supported target environment, the applicable build options, supported configurations and how to select the featured capabilities and how to enable the allowed user customizations for the software to be installed and how the same can be realized.

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 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 driver root folder i.e. ti\sdo\pspdrivers\drivers\mcasp folder and the driver library is provided in lib\DM648/C6452 folder inside individual driver folder.

2.3 Build Options

Please refer to section 8.2 for driver specific build option.

The build folder contains a CCS project file that builds the driver into a library for debug and release mode.

Build options – *"iDebug"* and *"iRelease"* are provide to enable audio Instrumentation in the driver.

3 DSP/BIOS AUDIO DRIVER Structures

This section discusses about the initialization details and initialization structures used in the audio driver.

Most members of these structures directly reflect the McASP peripheral register settings. The driver **does not** check the validity of these parameters. It is the application's responsibility to pass proper value according to the McASP register description. Kindly refer McASP Peripheral Reference Guide for more details.

NOTE: Audio driver independently does not have any significance. It acts as an interface between application and McASP device driver. McASP device driver is responsible for the actual data transfer between the memory and external world. Audio driver is responsible for the audio codec related configurations only.

3.1 Initialization details

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

To have Audio device driver included in the application, corresponding TCI file i.e. *"dm648_audio_mcasp.tci"* for DM648 and *"c6452_audio_mcasp.tci"* for C6452 must be included in BIOS TCF file of the application for using McASP instance - 0 of the driver. This .tci file can be found in audio sample directory.

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

Note: Here TCI file's configuration is based on SIO-DIO class driver model. Please refer to mini driver document for how to create SIO/DIO instance.

1) UDEV object's configuration

TCI Configuration Parameters	Description
initFxn - Init Function	Pointer to application function to initialize DM648/C6452 audio ports like enabling pin muxing.
fxnTable - Function Table Pointer	*AUDIO_AIC33_FXNS*. This is a global variable which points to the AUDIO driver APIs.
fxnTableType - Function Table Type	*IOM_Fxns*
deviceId - Device Id	Specify which McASP port to use. For example to use McASP 0 this should be given as 0.
params – Pointer to Port parameter	N/A, not used by this driver
Device Global Data Pointer	N/A, not used by this driver

Final tci file should contain the following details which were explained above:

```
var udevCodec1 = bios.UDEV.create("udevCodec1");
udevCodec1.fxnTable = prog.extern("AUDIO_AIC33_FXNS");
udevCodec1.initFxn = prog.extern("AUDIO_AIC33_init1");
```

udevCodec1.fxnTableType = "IOM_Fxns";
udevCodec1.params = null;
udevCodec1.deviceId = 0;
udevCodec1.deviceGlobalDataPtr = null;
2) DIO object's configuration

TCI Configuration Parameters	Description
comment – comment	Comment for DIO adapter
useCallBackFxn - fxns Table	Use callback version of DIO function table, "false" in audio driver case
deviceName – device name	Name of the UDEV object's instance
chanParams – channel parameters	An optional pointer to an object of type *PSP_audio_cfg* as defined in the header file *psp_audio.h*. This pointer will point to an audio driver's configuration structure. In BIOS TCI file, this structure object passed as an argument.

Final tci file should contain the following details which were explained above:
var dioCodec = bios.DIO.create("dioCodec");
dioCodec.comment = "DIO Adapter for IOM McASP Codec driver" ;
dioCodec.deviceName = prog.get("udevCodec1");
dioCodec.useCallBackFxn = false;
dioCodec.chanParams = prog.extern("audio_cfg");

3.2 Enumeration

3.2.1 PSP_audio_inputMode

"psp_audio.h" file contains *PSP_audio_inputMode* enum that is used for the input/record source configuration. Input can be taken either from MIC or LINE_IN.

Enum Members	Description
PSP_AUDIO_MIC_IN	AIC33 will recognize audio input from MIC jack Here MIC Bias output is powered down
PSP_AUDIO_LINE_IN	AIC33 will recognize audio input from LINE IN jack
PSP_AUDIO_MIC_IN_2_0_V	AIC33 will recognize audio input from MIC jack Here MIC Bias output is powered to 2.0 V
PSP_AUDIO_MIC_IN_2_5_V	AIC33 will recognize audio input from MIC jack Here MIC Bias output is powered to 2.5 V
PSP_AUDIO_MIC_IN_AVDD	AIC33 will recognize audio input from MIC jack Here MIC Bias output is connected to AVDD

3.2.2 PSP_audio_outputMode

"psp_audio.h" file contains *PSP_audio_outputMode* enum that is used for the output/playback source configuration. Output can be given to either speaker or LINE_OUT.

Enum Members	Description
PSP_AUDIO_SPK_OUT	AIC33 will give audio output from speaker jack
PSP_AUDIO_LINE_OUT	AIC33 will give audio output from LINE OUT jack

3.2.3 PSP_audioAicMode

"psp_audio.h" file contains *PSP_audioAicMode* enum that is used for select audio codec mode of operation. AIC33 can be configured in either master mode or slave mode.

Enum Members	Description
PSP_AUDIO_AIC_SLAVE	AIC33 will configure in slave mode and McASP will configure in master mode.
PSP_AUDIO_AIC_MASTER	AIC33 will configure in master mode and McASP will configure in slave mode.

3.3 Data structure

3.3.1 PSP_audio_cfg

"psp_audio.h" file contains *PSP_audio_cfg* data structure that is passed while mdCreateChan call which is defined with DIO AUDIO parameters in TCF file of application. The members of this structure are explained below:

Structure Members	Description
mode	Datatype = PSP_audioAicMode, Audio codec mode of operation. Master or slave PSP_AUDIO_AIC_MASTER – AIC33 is in master mode PSP_AUDIO_AIC_SLAVE – AIC33 is in slave mode NOTE: This parameter must be configured with PSP_AUDIO_AIC_MASTER as McASP can not be configured in master mode due to DM648/C6452 EVM limitation
inputFreq	Datatype = Uint32, Input frequency to McASP module. This parameter is of importance only in case when McASP is in master mode
sampleFreq	Datatype = Uint32, Output sampling rate. This parameter will set sampling rate for AIC33 or McASP, whichever is master.
inputGain	Datatype = Uint32, Input gain control. The value should be in range of 1 and 99
outputGain	Datatype = Uint32, Output gain controls. The value should be in range of 1 and 99
inputMode	Datatype = PSP_audioInputMode, Pointer to the Input mode structure to select input source, MIC_IN or LINE_IN. In case of MIC_IN use can configure mic bias voltage.
outputMode	Datatype = PSP_audioOutputMode, Audio output from system through Speaker out or LINE OUT. This parameter has no effect. Changing this

	parameter will not take effect as DM648/C6452 EVM supports only LINE OUT

3.3.2 PSP_McaspHwSetupDataClk

"psp_mcasp.h" file contains *PSP_McaspHwSetupDataClk* data structure. While mdCreateChan call of McASP driver, audio driver pass default value of this structure's object.

Structure Members	Description
clkSetupClk	Datatype = Uint32, Clock details ACLK(R/X)CTL Note: Refer McASP peripheral user guide for more information of ACLK(R/X)CTL
clkSetupHiClk	Datatype = Uint32, High clock details AHCLK(R/X)CTL Note: Refer McASP peripheral user guide for more information of AHCLK(R/X)CTL
clkChk	Datatype = Uint32, Configures receive/transmit clock failure detection R/XCLKCHK Note: Refer McASP peripheral user guide for more information of R/XCLKCHK

3.3.3 PSP_McaspHwSetupData

"psp_mcasp.h" file contains *PSP_McaspHwSetupData* data structure. While mdCreateChan call of McASP driver, audio driver pass default value of this structure's object.

Structure Members	Description
mask	Datatype = Uint32, To mask or not to mask - R/XMASK Note: Refer McASP peripheral user guide for more information of R/XMASK
fmt	Datatype = Uint32, Format details as per - R/XFMT Note: Refer McASP peripheral user guide for more information of R/XFMT
frSyncCtl	Datatype = Uint32, Configure the rcv/xmt frame sync - AFSR/XCTL Note: Refer McASP peripheral user guide for more information of AFSR/XCTL
tdm	Datatype = Uint32, Specifies which TDM slots are active - R/XTDM Note: Refer McASP peripheral user guide for more information of R/XTDM
intCtl	Datatype = Uint32, Controls generation of McASP interrupts -

	R/XINTCTL Note: Refer McASP peripheral user guide for more information of R/XINTCTL
stat	Datatype = PSP_audioInputMode, Status register (controls writable fields of STAT register) -R/XSTAT Note: Refer McASP peripheral user guide for more information of R/XSTAT
evtCtl	Datatype = PSP_audioOutputMode, Event control register - R/XEVTCTL Note: Refer McASP peripheral user guide for more information of R/XEVTCTL
clk	Datatype = PSP_McaspHwSetupDataClk Clock settings for rcv/xmt

Default value of PSP_McaspHwSetupData's object

```
/* Transmit channel default register configuration */
PSP_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,
                                           MSB first,
                                           0 bit delay from between fsync and data*/
        /* .afsxctl  = */   0x00000000,    /*burst mode,
                                           frame sync pulse width - 1 clk bit
                                           clk is external
                                           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
         },
```

```
/* Receive channel default register configuration */
PSP_McaspHwSetupData mcaspRcvSetup =
        /* .rmask= */     0xFFFFFFFF,   /*all the data bits are used*/
        /* .rfmt     = */   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,
                                           MSB first,
```

```
                                        0 bit delay from between fsync and data*/
    /* .afsrctl  = */    0x00000000,    /*burst mode,
                                        frame sync pulse width - 1 clk bit
                                        clk is external
                                        Rising edge is validated*/
    /* .rtdm     = */    0x00000003,     /*n-th (2 here)TDM slot will be accounted - for burst mode no
                                        use*/
    /* .rintctl  = */    0x00000000,    /*reset any existing status bits*/
    /* .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
     },
```

4 DSP/BIOS Audio Driver API classification

4.1 DSP/BIOS Audio Driver Initialization

DSP/BIOS calls initialization routine which internally invokes "AUDIO_AIC33_init()" API of the Audio driver, which in turn initializes the global data used by the McASP.

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 Audio driver "audio_mdBindDev ()"is called by the DSP/BIOS initialization routine.

The binding function should typically perform the following actions.

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

4.3 IOM Channel Creation

After the successful completion of driver initialization and binding, the user can create an abstraction of the communication path between the application and Audio driver. The channel instances are created through a call to audio_mdCreateChan () function, which runs as a result of a call from application call to channel creation API. The Audio driver allows user to create two channels (in input and output mode) for a given McASP device instance on the SoC. The device IOM driver returns error status if the application attempts to create a channel in mode other than either input or output. For a given McASP instance only one input and one output channel can be created.

4.4 Control Commands

The device IO Mini driver implements device specific control functionality which may be useful for application designers. The device IOM driver supports the following control functionality.

4.4.1 AUDIO_STOP

Syntax

Int SIO_ctrl(stream, cmd, arg)

Parameters

Stream

stream handle

cmd

AUDIO_STOP

args

NULL

Return Value
if successful - SYS_OK
if unsuccessful - non-zero device-dependent error value

Description
Stops record/playback operation depending of the i/o stream passed.

Constraints
NONE

Example

```
stat = SIO_ctrl(outStream,AUDIO_STOP, NULL);
```

4.4.2 AUDIO_START

Syntax
Int SIO_ctrl(stream, cmd, arg)

Parameters

Stream

stream handle

cmd

AUDIO_START

args

NULL

Return Value
if successful - SYS_OK
if unsuccessful - non-zero device-dependent error value

Description
Starts record/playback operation depending of the i/o stream passed.

Constraints
AUDIO_STOP command has to be issued before AUDIO_START is issued otherwise driver will return error.

Example

```
stat = SIO_ctrl(outStream,AUDIO_START, NULL);
```

4.4.3 AUDIO_PAUSE

Syntax

Int SIO_ctrl(stream, cmd, arg)

Parameters

Stream
stream handle

cmd
AUDIO_PAUSE

args
NULL

Return Value
if successful - SYS_OK
if unsuccessful - non-zero device-dependent error value

Description
"Pause" record/playback operation of passed IO stream. No audio record/playback will take place when driver is in pause state.

Constraints
NONE

Example

```
stat = SIO_ctrl(outStream,AUDIO_PAUSE, NULL);
```

4.4.4 AUDIO_RESUME

Syntax
Int SIO_ctrl(stream, cmd, arg)

Parameters

Stream
stream handle

cmd
AUDIO_RESUME

args
NULL

Return Value
if successful - SYS_OK
if unsuccessful - non-zero device-dependent error value

Description
"Resume" record/playback operation of passed i/o stream. Call this command to continue with record/playback operation after "AUDIO_PAUSE" command is issued.

Constraints
This command should be called after "AUDIO_PAUSE", if "AUDIO_ RESUME" command called before "AUDIO_PAUSE" then driver returns an error message.

Example

```
stat = SIO_ctrl(outStream,AUDIO_RESUME, NULL);
```

4.4.5 AUDIO_MUTEON

Syntax
Int SIO_ctrl(stream, cmd, arg)

Parameters

Stream
stream handle

cmd
AUDIO_MUTEON

args
NULL

Return Value
if successful - SYS_OK
if unsuccessful - non-zero device-dependent error value

Description
Mute the output stream. User will not hear any audio from LINE_OUT when driver is in "MUTEON" state

Constraints
'AUDIO_MUTEON' Command is not applicable for input channel.

Example

```
stat = SIO_ctrl(outStream,AUDIO_MUTEON, NULL);
```

4.4.6 AUDIO_MUTEOFF

Syntax
Int SIO_ctrl(stream, cmd, arg)

Parameters

Stream
stream handle

cmd
AUDIO_MUTEOFF

args
NULL

Return Value

if successful - SYS_OK
if unsuccessful - non-zero device-dependent error value

Description
Take driver out of Mute state for the output stream.

Constraints
This command should be called after "AUDIO_MUTEON". 'AUDIO_MUTEOFF' command is not applicable for input channel.

Example

```
stat = SIO_ctrl(outStream,AUDIO_MUTEOFF, NULL);
```

4.4.7 AUDIO_INPUT_SAMPLERATE

Syntax
Int SIO_ctrl(stream, cmd, arg)

Parameters

Stream
stream handle

cmd
AUDIO_INPUT_SAMPLERATE

args
address of a new samplerate value

Return Value
if successful - SYS_OK
if unsuccessful - non-zero device-dependent error value

Description
Modifies input stream's sample rate

Constraints
Sample rate of input stream should not be changed while recording is going on. This may result in unexpected behavior.

Example

```
newSampleRate = 48000;

stat = SIO_ctrl(inStream,AUDIO_INPUT_SAMPLERATE,&newSampleRate);
```

4.4.8 AUDIO_OUTPUT_SAMPLERATE

Syntax
Int SIO_ctrl(stream, cmd, arg)

Parameters

Stream

stream handle

cmd

AUDIO_OUTPUT_SAMPLERATE

args

address of a new samplerate value

Return Value
if successful - SYS_OK
if unsuccessful - non-zero device-dependent error value

Description
Modifies output stream's sample rate

Constraints
Sample rate of output stream should not be changed while playback is going on. This may result in unexpected behavior.

Example

```
newSampleRate = 48000;

stat = SIO_ctrl(outStream,AUDIO_OUTPUT_SAMPLERATE,&newSampleRate);
```

4.4.9 AUDIO_IN_SELECT

Syntax
Int SIO_ctrl(stream, cmd, arg)

Parameters

Stream

stream handle

cmd

AUDIO_IN_SELECT

args

NULL

Return Value
if successful - SYS_OK
if unsuccessful - non-zero device-dependent error value

Description
Changes the input/record source based on the structure configuration. "audio_cfg" structure is passed to reconfigure AIC33 completely. It implies that if user modifies any other parameter of the structure, AIC33 will be configured with that configuration. It will not retain any old configuration of AIC33.

Constraints
DM648/C6452 EVM does not support MIC as an input source. Configuring MIC IN as an input source will throw an error from driver.

Example

```
audio_cfg.inSelect = PSP_AUDIO_MIC_IN;
stat = SIO_ctrl(inStream,AUDIO_IN_SELECT, (Arg) &audio_cfg);
```

4.4.10 AUDIO_OUT_SELECT

This command is not supported due to hardware limitation

4.4.11 AUDIO_IN_GAIN_CONTROL

Syntax

Int SIO_ctrl(stream, cmd, arg)

Parameters

Stream

stream handle

cmd

AUDIO_IN_GAIN_CONTROL

args

address of a new gain value

Return Value

if successful - SYS_OK
if unsuccessful - non-zero device-dependent error value

Description

Changes gain of input channel according to new gain value passed.

Constraints

Value of gain should be in the range of 1 and 99

Example

```
newGain = 75;
stat = SIO_ctrl(outStream,AUDIO AUDIO_IN_GAIN_CONTROL,(Arg)&newGain);
```

4.4.12 AUDIO_OUT_GAIN_CONTROL

Syntax

Int SIO_ctrl(stream, cmd, arg)

Parameters

Stream

stream handle

cmd

AUDIO_OUT_GAIN_CONTROL

args

address of a new gain value

Return Value
if successful - SYS_OK
if unsuccessful - non-zero device-dependent error value

Description
Changes gain of output channel according to new gain value passed.

Constraints
Value of gain should be in the range of 1 and 99

Example

```
newGain = 75;
stat = SIO_ctrl(outStream,AUDIO AUDIO_OUT_GAIN_CONTROL,(Arg)&newGain);
```

4.4.13 AUDIO_DEVICE_RESET

Syntax
Int SIO_ctrl(stream, cmd, arg)

Parameters

Stream
stream handle

cmd
AUDIO_DEVICE_RESET

args
NULL

Return Value
if successful - SYS_OK
if unsuccessful - non-zero device-dependent error value

Description
Resets the audio driver peripheral i.e. McASP. After this command is issued successfully, audio record and playback will not continue. Please follow below mentioned steps for restarting record and playback again.
Step 1 – Issue AUDIO_DEVICE_RESET command
Step 2 – Free the allocated memory, if any
Step 3 – Delete both the channels
Step 4 – Create both the channels again
Step 5 – (Optional) Only in case of McASP in master mode (i.e. AIC33 in slave mode) call IOCTL "PSP_CTRL_RCV_GPIO_INPUT".
Step 6 – Prime the driver

Constraints
None

Example

```
stat = SIO_ctrl(outStream, AUDIO_DEVICE_RESET,NULL);
```

4.4.14 AUDIO_MODIFY_LOOPJOB

Syntax

Int SIO_ctrl(stream, cmd, arg)

Parameters

Stream

stream handle

cmd

AUDIO_MODIFY_LOOPJOB

args

address McASP channel param structure

Return Value

if successful - SYS_OK
if unsuccessful - non-zero device-dependent error value

Description

Enables application provided loopjob or internal loopjob depending on the input parameter passed. If no buffer is provided then driver will enable internal loopjob. Loopjob only takes effect when driver is starved of the requests.

Constraints

NONE

Example

```
PSP_mcaspChanParams chanPrams;
unsigned short loopBuf[512];

chanParams.userLoopJobBuffer = & loopBuf[0];
chanParams.userLoopJobLength = 512;

stat = SIO_ctrl(outStream,AUDIO_MODIFY_LOOPJOB,(Arg)&chanParams);
```

4.4.15 AUDIO_SET_AIC33_SLAVE_ADDRESS

Syntax

Int SIO_ctrl(stream, cmd, arg)

Parameters

Stream

stream handle

cmd

AUDIO_SET_AIC33_SLAVE_ADDRESS

args

address of new AIC33's slave address

Return Value

successful - SYS_OK

Description
This IOCTL is used for changing the slave address with the slave address of AIC that application wants to configure. In DM648/C6452 EVM there are in total of 4 AICs. For example, If a user wish to modify the output gain of any of the AIC, user needs to call this IOCTL to set the slave address of AIC that he wishes to configure and then call AUDIO_OUT_GAIN_CONTROL to modify the gain. Please refer to the example above for reference.

Constraints
None

Example

```
Uint16 aic33SlaveAddress;
Uint32 newGain = 75;
Uint16 Aic33SlaveAddtess = 0x1B;

stat = SIO_ctrl (      outStream,
                       AUDIO_SET_AIC33_SLAVE_ADDRESS,
                       (Arg)&aic33SlaveAddress );
stat = SIO_ctrl(outStream,AUDIO_OUT_GAIN_CONTROL,(Arg)&newGain);
```

4.5 IOM Channel Deletion

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

4.6 Audio IO Mini driver unbinding

The resources allocated by the "audio_mdBindDev ()" function are freed by calling device "audio_mdUnBindDev ()".

5 Audio Codec API's

This chapter describes the functions, control commands, data structures, enumerations and macros for the AUDIO driver module.

5.1 API Definition

5.1.1 aic33Codec_config

Syntax
Int PAL_ aic33Codec_config (Uint32 Id, PSP_audio_cfg audioCfg);*

Parameters
Id
Audio codec ID
audioCfg
Audio Codec configuration structure

Return Value
It returns the handle on successful opening of a device. It returns error if the device could not be opened.

Description
This function configures AIC33 codec.

Constraints
None

5.1.2 aic33_closeCodec

Syntax
Void aic33_closeCodec (aic33_CodecHandle aic33handle);

Parameters
aic33handle
Handle of the opened instance

Return Value
None

Description
This function call will close the AIC33 codec instance.

Constraints
None

5.1.3 aic33_inGainControl

Syntax
int aic33_inGainControl (Uint8 newGain, PSP_audio_cfg audioCfg);*

Parameters
newGain
New gain setting for the input channel. New gain value should be in the range of 1 to 99 otherwise function will return an error

audioCfg
Audio codec configuration structure

Return Value
On failure returns -1, on success returns 0.

Description
This function configures the gain of Input channel.

Constraints
None

5.1.4 aic33_outGainControl

Syntax
int aic33_outGainControl (Uint8 newGain, PSP_audio_cfg audioCfg);*

Parameters
newGain
New gain setting for the input channel. New gain value should be in the range of 1 to 99 otherwise function will return an error

audioCfg
Audio codec configuration structure

Return Value
On failure returns -1, on success returns 0.

Description
This function configures the gain of Input channel.

Constraints
None

5.1.5 aic33_sampleRateControl

Syntax
int aic33_sampleRateControl (PSP_audio_cfg audioCfg);*

Parameters
audioCfg
Audio codec configuration structure

Return Value
On failure returns -1, on success returns 0.

Description
This function configures AIC33 for passed sample rate.

Constraints
None

6 Architecture

The device driver described here is part of an IOM mini-driver. That is, it is implemented as the lower layer of a 2-layer device driver model. The upper layer is called the class driver and can be either the DSP/BIOS GIO, SIO/DIO, or PIP/PIO modules. 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. **Error! Reference source not found.** shows over all DSP/BIOS device driver architecture.

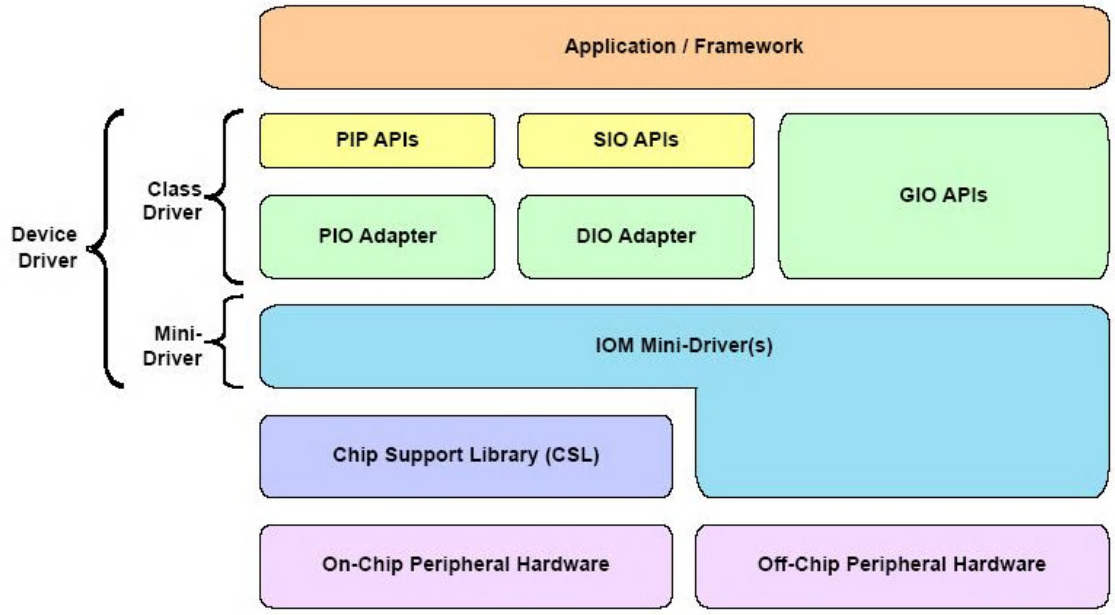


Figure 2. DSP/BIOS Device Driver Model

Mini-driver implementation is split into a codec-specific portion and a generic portion that will work across many different codecs. **Error! Reference source not found.** shows the data flow between the components in a system in which the mini-driver is split into a generic part i.e. McASP driver and audio codec-specific part.

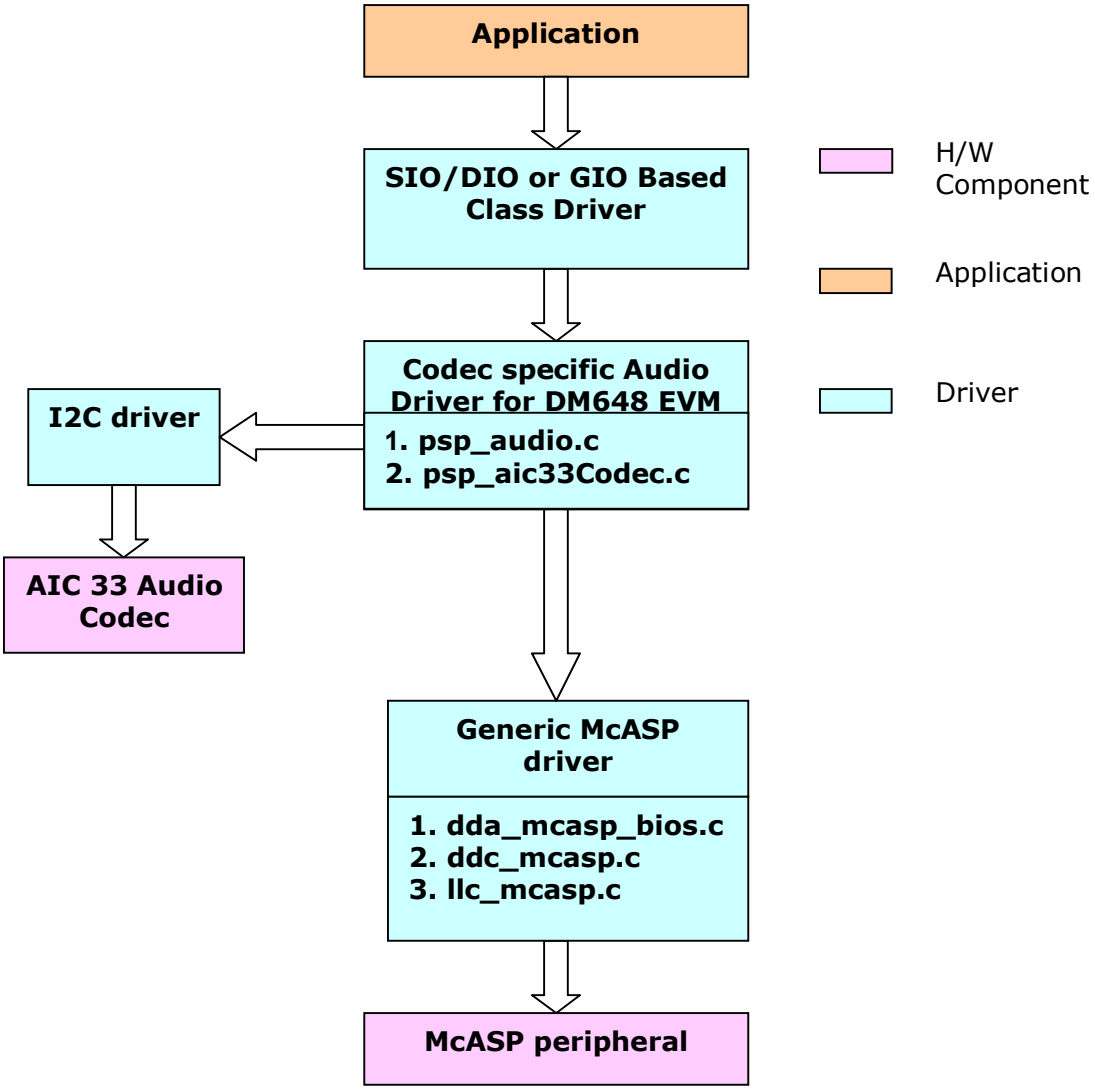


Figure 3. Codec Device Driver Partitioning

NOTE: Chip support library (CSL) here is register layer CSL only.

The codec specific portion of the mini-driver uses two codec specific functions, mdBindDev() and mdCreateChan(), to configuration audio codec. These functions then call mdBindDev() and mdCreateChan() in the generic driver to complete generic portions of the driver initialization. The only thing the codec-specific part does is to set up the codec and leaves the transfers of samples to the generic device driver. The fact that this device driver uses the generic device driver is hidden from the user in all aspects except that the generic device driver library has to be linked into the application. The function mdCreateDev() is responsible for configuring the codec It does minimal setup required and then calls a function called aic33Codec_config () function in psp_aic33Codec.c to perform most of the configuration work required to prepare audio codec for record and playback operation. The function mdCreateChan() generates the EDMA configuration used for the data transfers.

7 DM648/C6452 EVM Limitations

This section lists limitations imposed due to the DM648/C6452 EVM design.

1. As AIC33 supports 16, 20, 24 and 32 bits, channels should be created using any of these word widths only. Due to this McASP is not tested for word width other than the ones supported by 16, 20, 24 and 32 bits.

2. Although McASP master mode is supported in McASP driver, due to the design of the EVM, McASP can not be configured in master mode. If user tries to configure McASP in master mode, error will be returned from the audio driver.

3. MIC IN is not supported as a recording source. If user tried to configure AIC33 in MIC IN mode, error will be returned from the audio driver.

8 Example Applications

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

8.1 Writing Applications for AUDIO

This section provides guidance for user for writing their own application for AUDIO drivers.

8.1.1 File Inclusion

To write sample application user has to include following header files in the application:
psp_audio.h
This file contains configuration structures and defines for AUDIO driver configuration. It also contains ConfigCommand enum that is passed while calling IOCTL for audio driver from application.

8.1.2 Buffer Allocation and Management

Buffer allocation is dependent on number of requested serializer. Requested buffer size should be equal to **no_of_serializer * buff_size**. Where **no_of_serializer** represents number of serializers and **buff_size** represents number of samples per serializer per receive/transmit request

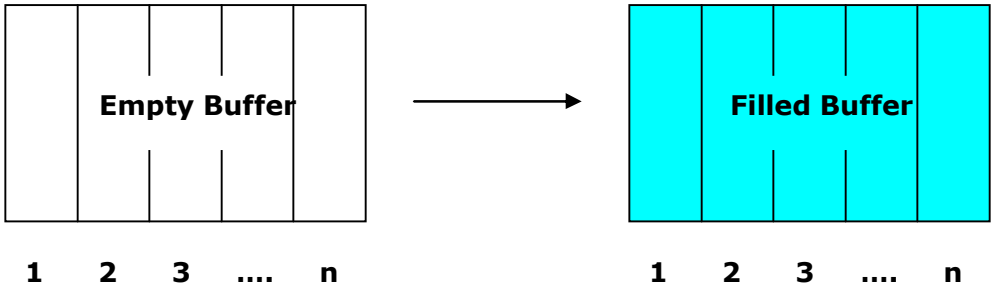
Below cases shows, how a receive buffer allocation is done using single serializer and multiple serializers. Same way buffer allocation is done for transmit channel as well.

8.1.2.1 Buffer allocation for single serializer (receive/transmit)

User has to request receive/transmit buffer with the size of **no_of_serializer * buff_size** and pass it to driver. Here in this case **no_of_serializer** is equal to 1 and **buff_size** is equal to number of samples per serializer per receive/transmit request. Requested serializer fills the buffer and that buffer gives back to application.

8.1.2.2 Buffer allocation for multiple serializer

In case of user is using multiple serializers, user has to request receive/transmit buffer with the size of **no_of_serializer * buff_size** and pass it to driver. Here in this case **no_of_serializer** is equal to number requested serializer by a receive/transmit channel and buff_zise is equal to number of samples per serializer per receive/transmit request. All requested serializers fill the buffer and that buffer give back to application. Below figure shows the buffer allocation for multiple serializer.



Here, numbers represents serializer number.

Required Buffer size = no_of_serializer*buff_size

Where,
no_of_serializer– Number of Serializer
buff_size – Number of Samples per serializer

For example, no_of_serializer is equal to 4 and buff_size is equal to 1024 bytes, then required buffer size is equal to 4*1024 = 4096 bytes and 4 serializer fills buffer respectively.

8.2 Sample Application

The sample application is a representative test program. Initialization of AUDIO driver is done by calling initialization function from BIOS.

Audio driver features a compile time Macro option to provide ONE_TO_ONE (One serializer as input and one serializer as output) and FOUR_TO_ONE (4 serializer i.e. 8 mono channels and 1 serializer as output) options.

Applications are supposed to provide buffers of appropriate size. Buffer size should be equal to **buffer of each serializer * no. of serializers** (configured during compile time) to the driver. To facilitate the applications which use only one serializer (as in GIO example of DVSDK) ONE_TO_ONE option is provided.

In this release the audio drivers have been pre-built using both macros and application can use (link) them using RTSC query through CFG file that is part of the application project. Please note the statement "Settings.channel = ONE_TO_ONE "in the CFG file of dm648_evm_audio_st_sample_1_to_1.pjt for DM648 and c6424_evm_audio_st_sample_1_to_1.pjt for C6452, which would get the one to one library to application for linking. If a CFG file does not have this statement, then by default four to one library is linked to sample application.

In this package among the standalone audio sample applications, **dm648_evm_audio_st_sample_1_to_1.pjt for DM648 and c6424_evm_audio_st_sample_1_to_1.pjt for C6452** shows the usage of one to one library and **dm648_evm_audio_st_sample.pjt for DM648 and c6424_evm_audio_st_sample.pjt for C6452** shows the usage of four to one library.

Please note that the four to one mode is the default mode for the audio driver of this EVM and one to one mode is supported as add-on feature to support other existing applications.

A. All the applications which used the ONE_TO_ONE macro in the audio driver pjt file and uses one serializer for input and one serializer for output (GIO audio example in DVSDK) should

   1. In RTSC way, add **Settings.channel = Settings.ONE_TO_ONE** statement in the CFG file. Please refer the dm648_evm_audio_st_sample_1_to_1.pjt for DM648 and c6424_evm_audio_st_sample_1_to_1.pjt for C6452 and its CFG file for the actual usage (example given below)

```
"/* USE Audio Driver for DM648 */
var audio =
xdc.loadPackage('ti.sdo.pspdrivers.system.DM648.bios.evmDM648.audio');
var Settings =
xdc.module('ti.sdo.pspdrivers.system.DM648.bios.evmDM648.audio.Settings');
Settings.channel = Settings.ONE_TO_ONE;

"/* USE Audio Driver for C6452*/
var audio = xdc.loadPackage('ti.sdo.pspdrivers.system.C6452.bios.evm6452.audio');
var Settings =
xdc.module('ti.sdo.pspdrivers.system.C6452.bios.evm6452.audio.Settings');
Settings.channel = Settings.ONE_TO_ONE;
```

   2. Non RTSC (conventional applications) can add the library or library pjt of ONE to ONE audio library in the application project file as demonstrated in combo1 pjt (combined sample application) file.

B. All application which would use the 4 serializer for input and one for output (demo example showing multi-channel video and audio)

   1. In RTSC based application, just add lib from the audio driver package (by default audio driver with 4 to 1 capability would be delivered by xdc for linking)

   2. In legacy way, add the library or library pjt file into application project file

- Note: Following Components needs to be linked for successful build and functionality of the application.
  - McASP
  - I2C
  - PAL_OS
  - SoC specific PAL_SYS

- Note:
  The AIC33 codec driver uses I2C driver which is an asynchronous driver. However, the codec relies on the result of the operation which means, it needs a synchronous operation. Hence, it implements a semaphore to pend on completion of this operation. The semaphore is posted by from inside a callback registered to the I2C driver, by the codec driver, during open. However, the call back is called only in interrupt context. Thus the I2C driver must be configured to work in interrupt mode, by the sample application. If this not done the codec operations shall fail.

9 Appendix A - DSP/BIOS Audio Driver References

References

[1] AIC33 audio codec datasheet (07 Jul 2006). To download the datasheet please visit http://focus.ti.com/docs/prod/folders/print/tlv320aic33.html