User Guide

BIOSPSP DA830 02.00.01

June 06, 2009

This page has been intentionally left blank.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:
Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

This page has been intentionally left blank.

TABLE OF CONTENTS

1 Top level Information

1.1 Introduction

This chapter introduces the **DA830 BIOS PSP** to the user by providing a brief overview of the purpose and construction of the **DA830 BIOS PSP** along with hardware and software environment specifics in the context of **DA830 BIOS PSP** deployment.

1.1.1 Overview

The **DA830 BIOS PSP** is aimed at providing fundamental software abstractions for on-chip resources and plugs the same into DSP/BIOS operating system so as to enable and ease application development by providing suitably abstracted interfaces.

1.1.2 Terms and Abbreviations

API	Application Programmer's Interface
CSL	TI Chip Support Library – primitive h/w abstraction.
IP	Intellectual property
ISR	Interrupt Service Routine
OS	Operating System
ID	Installation Directory
MMC	Multimedia Card
SD	Secure Digital

1.1.3 References

1	SPRS483	DA830 SoC reference Guide

1.1.4 Supported Services and features

DA830 BIOS PSP provides the following:

- Device drivers for UART, I2C, SPI, McASP, GPIO, PSC, MMCSD, NAND, LCDC Raster and LCDC LIDD controllers.
- Sample applications that demonstrate use of drivers for UART (loop back & Echo Test), I2C (writes to an on board EEPROM), SPI (writes to an on board Serial Flash), McASP (captures and plays a tone), GPIO (user switch and LED interaction), LCDC Raster (displays 3 color stripes with a scrolling line over it) and LCDC LIDD (displays a welcome message on the HDM24216H-2 24x2 character display). MMCSD, NAND (reads and writes to the storage device via Block Media interface).
- rCSL and Examples for selected peripherals

1.1.5 System Requirements

The following products are required to be installed prior to using **DA830 BIOS PSP**

- EDMA 3 LLD – This package (DA830 BIOS PSP) is compatible with EDMA 3 LLD versioned 02.00.01.04
- DSP-BIOS versioned 6.20.00.37
- XDC Tools versioned 3.15.00.50
- CCS version 4.0.0.14 (version 4 Beta5)
- Code Generation Tools 6.1.9
- XDS 510 USB Emulator (Optional) – EVM has on board emulator
- DA830 EVM CPU Board (Revision B)
- DA830 User Interface Module
- HDM24216H-2 24X2 character display

1.2 Installation Guide

This chapter discusses the **DA830 BIOS PSP** installation, how and what software and hardware components to be availed in order to complete a successful installation (and un-installation) of **DA830 BIOS PSP**.

1.2.1 Installation and Usage Procedure

1.2.1.1 Installation procedure for DSP/BIOS

1. Install the following products mentioned in system requirements sections, as per instructions provided along with the products.
2. Install the PSP package (BIOSPSP_xx_yy_zz_bb_DA830_Setup.exe)using the self extracting installer
3. Install EDMA-3 LLD Device Driver into preferred drive / folder
4. The environment variable 'EDMA3LLD_BIOS6_INSTALLDIR' is used in the sample application projects for referring to the EDMA3 driver libraries. This environmental variable is created and updated by the EDMA3 LLD driver during its installation. Please ensure that this environmental variable is pointing to the EDMA3 LLD driver install directory intended to be used along with this package. This is more important when there are multiple EDMA3 LLD installations as the EDMA3 LLD installer updates this environment variable with latest installation version. (e.g. If EDMA3 LLD Driver is installed into c:\edma3_lld\ then environment variable EDMA3LLD_BIOS6_INSTALLDIR=c:\edma3_lld\)
5. For building the downloadable images refer to section 1.3
6. Download the executable image, with file extension *.x674* (as the soc is of C674 ISA) of required application onto platform using CCS.
7. Run the program.
8. **Please avail the help on package locations and API information help from cdoc help that is available at <install dir>\packages\docs\DA830\cdoc\index.html**

1.2.1.2 Un-Installation

1. Uninstall the PSP package by using the uninstall.exe in the package directory.
2. Un-install the products (listed in system requirements) as per instructions provided with the product(**optional and at user's discretion**)

1.2.2 PSP Component Folder

This section details the files and directory structure of the installed **DA830 BIOS PSP** in the system. A view graph of the actual directory tree (as seen in the final deployed environment is inserted here for clarity.

1.2.2.1 Top level PSP Directory structure:

Figure 1: BIOS PSP Top level directory structure

The sections below describe the folder contents.

pspdrivers_

This contains the device drivers and other PSP components. It is the top level installation directory

docs

Contains release notes and users' guide for this PSP package.

cslr

This contains the register level chip support for DA830 and usage examples.

examples

Contains the sample applications for drivers provided as part of this package

platforms

Contains platform specific modules like codec drivers, interface modules etc., which may be specific to the EVM/Platform

All drivers are organized under ti/psp/ directory under their individual directories. For example, UART driver falls under ti/psp/uart.

1.2.2.2 Driver Directory structure:

Each driver directory (**ti/psp/<peripheral>**) is further organized as follows:

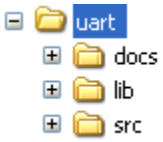


Figure 2: DA830 PSP driver directory structure

docs
it Contains peripheral specific documentation like Architecture documentation, doxygen documentation help files etc.

lib
Contains generated driver library file(s)

src
Contains the source file(s) for the BIOS PSP driver module

1.2.2.3 "examples" Directory structure:

Each driver sample application (**ti/psp/examples/<peripheral>**) is further organized as follows

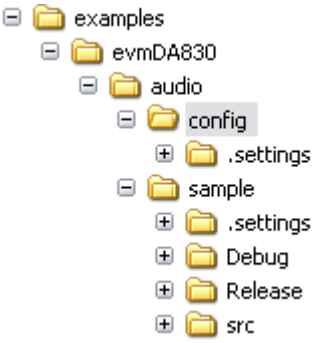


Figure 3: DA830 PSP driver sample application directory structure

evmDA830
Contains the EVM/platform specific examples

.settings
Contains CCS project settings and preferences

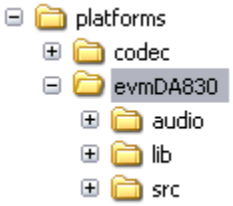
<few project files>
CCS4 project files

src
Contains the source file(s) for sample application program

Debug/Release
Contains the Debug/Release optimization profile level executables

1.2.2.4 platforms Directory structure:

Each platform related specific driver modules are further organized as



docs

Contains documentation related to the component

lib

Contains generated library file(s)

src

Contains source file(s)

1.3 Integration Guide

This chapter discusses the **DA830 BIOS PSP** package usage. As part of the PSP package, a demo application is provided to check the basic functionality for each of the device/driver.

1.3.1 Building the PSP Sample Applications

The PSP package contains separate sample applications for each of the BIOS drivers (except PSC).

To build the BIOS sample application

- **CCS v4 GUI based compilation:**
  1. Build the required libraries in the command line (**Please refer to section "**Building the BIOS PSP Driver Modules")
  2. Setup the CCS4 to set DA830 platform and use the appropriate DSP gel file.
  3. Load CCS project**.**
     - Open C/C++ perspective
     - "CCS-> Project->Open Existing Project" menu item.
     - Point to the directory of the sample application needed to run.
       - Example: z:\pspdrivers_xx_yy_zz_bb\packages\ti\psp\examples\evmDA830\uart
     - Select the projects "uartSample_configuration" and "uartSample" to be opened.
     - Set required Debug/Release configuration for both the "uartSample_configuration" and the "uartSample" projects.
     - "CCS-> Project->Rebuild Active Project" This builds the .x674 executable
       - Example: uartSample.x674
  4. Run->Launch TI Debugger. Right Click on 674X debug profile and set it as the debug scope.
  5. Use – "Target->Connect" to connect to DSP target, the GEL would configure and setup DSP to be used by the CCS window.
  6. Use "Target->Load Program" to download the .x674 executable.
     - Example: Z:\pspdrivers\pspdrivers_\packages\ti\psp\examples\evmDA830\uart\<Debug/Release>/uartSample.x674

- **Command line based compilation:**

  1. For building all examples at one go:
  1. Go to the examples directory:
     - Example: ***Z:\pspdrivers\pspdrivers_\packages\ti\psp\examples\evmDA830***
  2. Execute the following command ***Z:\pspdrivers\pspdrivers_\packages\ti\psp\examples\evmDA830>xdc clean –PR .*** *(optional – only for clean build)*

Z:\pspdrivers\pspdrivers_\packages\ti\psp\examples\evmDA830> xdc -PR .

2. For building individual examples:

Z:\pspdrivers\pspdrivers_\packages\ti\psp\examples\evmDA830\uart> xdc clean *(optional – only for clean build)*

Z:\pspdrivers\pspdrivers_\packages\ti\psp\examples\evmDA830\uart > xdc

1.3.2 BIOS PSP EVM Library Module

1.3.2.1 Description

The sample applications available in the package demonstrate the usage of the BIOS PSP drivers for DSP BIOS 6.20.xx on EVM DA830 platform. For successful operation of the applications, some basic initialization (e.g. configuring the pin multiplexers for the peripherals used etc) needs to performed. These initialization steps however are dependent on the SoC specifically.

Apart from this, the sample application may also have to do tasks specific to EVM on which it is intended to run. For instance, in case of LCDC or NAND applications for EVM DA830, one needs to select the LCDC or NAND peripheral since same pins are shared between them. This is achieved by configuring GPIO expander IC on the UI board.

The above mentioned initialization sequence, though necessary for a sample application to run successfully, become too much of a code information for a first time user of the sample application who would just like to have a look at the code and get a feel of the driver usage example.

Hence, in order to abstract the platform (EVM) specific initialization, these routines are organized as a separate library - ti.psp.platforms.evmDA830.a674. This library has the routines for the platform/EVM specific tasks. This helps in making the actual sample application simpler.

The platform directory has EVM specific code required by each module. All the EVM related information is placed inside file <module>_evmInit.c. This contains the code for any driver creation function required by the module, PINMUX settings for the module, any configuration required to be done by using the driver (like IO expander) etc. This folder also contains an entry in the configuration (*.cfg) file required for the creation of "dependency" drivers which will be used by that sample application.

The evmInit library files can be found under <ID>\packages\ti\psp\platforms\evmDA830 and contain:

1. Platform specific initialization routines in xxx_evmInit.c
2. Platform specific init configuration in xxx.cfg
3. Platform initialization library ti.psp.platforms.evmDA830.a674

Note: MMCSD and NAND are not XDC modules, so a file named <module>_startup is added for initializing these drivers. The routines in this file initialize the EDMA, Block Media and the specific modules.

1.3.2.2 Building the EVM Library Module

1. Go the EVM installation directory
   i. Example:
   Z:\pspdrivers\pspdrivers_\platforms\evmDA830
2. Execute the following command
   i. Example:
   Z:\pspdrivers\pspdrivers_\platforms\evmDA830> xdc

1.3.3 Building the BIOS PSP Driver Modules

1. For building all modules at one go:(please note: this also builds examples)

1. Go to the package install directory:
   - Example:
   Z:\pspdrivers\pspdrivers_

2. Execute the following command
   Z:\pspdrivers\pspdrivers_> xdc clean –PR . *(Optional – only for clean build; this also cleans the executables generated by sample application)*
   Z:\pspdrivers\pspdrivers_> xdc -PR .

2. For building individual modules

1. Go to the example directory
   - Example:
   Z:\pspdrivers\pspdrivers_\packages\ti\psp\uart

2. Execute the following command
   Z:\pspdrivers\pspdrivers_\packages\ti\psp\uart> xdc clean *(optional – only for clean build)*
   Z:\pspdrivers\pspdrivers_\packages\ti\psp\uart > xdc

NOTE:

1. Build (compilation) output

a. The examples/sample applications when compiled, generate by default, executables, with extension .x674, with *"debug"* and *"whole_program"* profiles in the directory Debug and Release respectively

b. The driver modules (or libraries) when compiled, by default generate libraries (archives), with extension .a674, in the lib directory. The profile for libraries will always be "whole_program_debug".

2. Building libraries using CCSv4 GUI

a. Building the libraries (driver modules) via CCSv4 is not supported in the current BIOSPSP package (due to the limitation of CCS4 to support xdc library pjt)

3. Ensure the following for a proper building of libraries and sample applications

a. pspdrivers_\packages\config.bld

i. contains the root directories for the Code Generation tools. It may be needed to edit and set the same to point to the same.

Example: If the code generation tools are installed in "C:\Program Files\C6000Code Generation Tools 6.1.9" then edit the config.bld file as shown:

rootDirPre = "C:\Program Files\"

rootDirPost = "Code Generation Tools 6.1.9"

by default the rootDir shall point to:

C:\Program Files\Texas instruments\CCSv4\tools\compiler\c6000

ii. Target selection

Currently only C674 target is supported, hence all other target options in "Build.targets" should be commented

b. The following environmental variables must be set

i. **XDCPATH –** Should include BIOS v6 package installation directory, XDC tools package directory, EDMA3 2.00.01.04 package directory and this PSP package installation directory.

Example:

z:\pspdrivers\pspdrivers_\packages;

c:\Program Files\Texas Instruments\edma3_lld_2_00_01_04\packages;

c:\Program Files\Texas Instruments\xdctools_3_15_00_50\packages;

c:\Program Files\Texas Instruments\bios_6_20_00_37\packages;

ii. EDMA3LLD_BIOS6_INSTALLDIR – Should be set to point the EDMA3 v 2.00.01.04 package installation directory.

Example: c:\Program Files\Texas Instruments\edma3_lld_2_00_01_04

The default installation directory for the EDMA package is *c:\Program Files\Texas Instruments\edma3_lld_<version_string>*

c. BIOS, Code Generation Tools and XDC versions selection

i. Ensure that the BIOS and XDC version mentioned in this guide in the section "System Requirements" is already installed in the users PC.

ii. The CCS version 4 enables selection of the DSP/BIOS and XDC version selection for the project, when multiple installations are present. In the *"Project->Properties->TI Build Settings"* menu ensure that the appropriate selections for DSP/BIOS support and RTSC support are made. One can also ensure other properties like Code Generation Tools version, device variant etc are properly selected.

d. Other options (*"Project->Properties->C/C++ Build")*

i. Ensure that the *xdcpath* options, under XDC are correctly pointing to the BIOS PSP package, EDMA package (if using EDMA).

ii. Ensure that the build configuration file path point the BIOSPSP package config.bld

iii. Target is set to ti.targets.C674

iv. Platforms is set to ti.platforms.evmDA830

v. Under advanced options for XDC, ensure that the build profile is as required (debug or whole_program).

vi. The BIOSPSP executables are, by default configured to be, generated with .x674 option. This can be changed to .out or any other extension as required in the "Build Settings" tab in "artifact extension" field

e. Target configuration (CCS setup in CCS3)

i. Proper target configuration should be chosen and setup. CCSv4, enables this by "Target->New Target Configuration" menu. Import the new configuration for your EVM (dskDA830 in this case) from: *C:\Program Files\Texas Instruments\CCSv4\common\targetdb\boards*

f. Compilation order

i. The libraries or the modules on which the sample application depends must be compiled first before compiling the sample application. For example, the audio sample application depends upon, McASP driver module, the audio and codec platform specific modules, the PSC module. These should be built before the application is built.

1.3.4 CSL Layer usage example

Sample code is provided to demonstrate the usage of CSL Register Layer with selected peripherals examples. The sample application building for CSL examples are similar to that of the driver sample applications explained above. For more information on CSL layer usage, please refer to the user guide located at, pspdrivers_\ti\psp\cslr\docs\cslr_userguide.doc.

1.3.5 On board DIP Switch Configuration

The following is the default switch configuration. Please refer EVM reference guide from the EVM manufacture for more information on these switches.

CPU Board KEY DIP Switches Configurations

SW3

1	■	
2	■	
3	■	
4	■	

SW5

1	■	
2	■	

3	■	
4	■	
5	■	
6	■	

SW2

1	■	
2	■	
3	■	
4	■	
5	■	
6	■	

2 UART driver

2.1 Introduction

This section is the reference guide for the UART device driver which explains the features and tips to use them.

DSP/BIOS applications use the driver typically through APIs provided by Stream layer, to transmit and receive serial data. The following sections describe in detail, procedures to use this driver, configure among others…. It is recommended to go through the sample application to get a feel of initializing and using the UART driver.

2.1.1 Key Features

- Multi-instantiable and re-entrant driver
- Each instance can operate as an receiver and or transmitter
- Supports Polled, Interrupt and DMA Interrupt Mode of operation
- Supports buffering on Transmit operation if enabled

2.2 Installation

The UART device driver is a part of PSP package for DA830 platform and would be installed as part of whole package installation. For high level design information please refer to the driver architecture guide that came with this package (available at <ID>\ti\psp\uart\docs)

2.2.1 UART Component folder

On installation of PSP package for DA830, the UART driver can be found at <ID>\ ti\psp\uart\

As show above the uart folder contains sub-folder, contents of which are described below.

- **uart -** The uart folder is the place holder for the entire UART driver, documents and the build configuration files. UART driver is implemented as IDriver under DSP/BIOS™ operating system. Stream defined APIs could be used to interface to UART driver. This folder contains the build configuration file (package.bld), the UART module specification file (Uart.xdc),the module script file (Uart.xs) and the miscellaneous files required for compiling the UART library.
- **docs –** Holds UART driver's architecture. Please note that the API reference would be found as a part of cdoc help (**<install dir>\packages\docs\DA830\cdoc\index.html)**
- **src –** Place holder for UART driver's source code.

2.2.2 Build Options

UART device driver will be built with "whole_program_debug" mode. When built successfully the respective library will be available at <ID>\ti\psp\uart\lib\ < ti.psp.uart.a674>

2.3 Features

This section details the features of UART and how to use them in detail.

2.3.1 Multi-Instance

The UART driver can operate on all the instance of UART on DA830. Different instances are specified during driver creation time. Supported instance are 0 through 2 with device ID 0 through 2 respectively.

These instances could be operated simultaneously with configurations supported by UART driver.

The device ID could be specified using the `instNum` field of structure Uart_Params. There are two ways in which a new instance of the UART driver can be created.

1. Static creation – static creation of the driver is done in the "cfg" file of the application. The creation happens at compile time.
2. Dynamic creation – Dynamic creation of driver is done in the application source files and the creation happens at runtime.

(i.e. Uart_Params.`instNum = 0x0`, Uart_Params.`instNum = 0x1`, so on…)

2.3.2 Each Instance as Transmitter and / or receiver

Each instance of the UART driver can be used for creating channels for transmit and receive operation. This could be achieved by opening a stream Channel as an INPUT channel and opening a stream Channel as an OUTPUT channel. The type of Channel is specified while creating the channel (using `Stream_create()` specify `"DriverTypes_OUTPUT" or "DriverTypes_INPUT"`). The configuration parameters are explained in the sections to follow.

2.4 Configurations

Following tables document some of the configurable parameter of UART. Please refer to Uart.h (Uart.xdc) /CDOC documentation for complete configurations and explanations.

2.4.1 Module configuration

The following parameters are module wide configurable parameters

Variable	Description
txBufferingEnable	Option to select if TX should allow buffering of data
paramCheckEnable	Option to select if parameter checking at the function entry should be enabled or disabled.
debugPrintEnable	Option to enable/disable the debug prints.
edmaEnable	Option to be enabled during the operation of the UART in EDMA mode.
TASKLET_PRIORITY	priority of the SWI tasklet used for handling the interrupts
MAX_ISR_LOOP	The number of iterations permitted in the ISR while processing the pending interrupts.

2.4.2 Uart_Params

This structure defines the device configurations, expected to supply while instantiating the driver (formerly known as devParams)

Members	Description
instNum	Instance number of the UART to be created.
enableCache	Whether the driver should deal with cache operations during EDMA mode of operation. Default is "true". If this is not set the user has to take care of the cache operations.
fifoEnable	Whether the HW FIFO for the device is to enabled
opMode	Operational mode of the UART driver (interrupt/EDMA/polled).
loopbackEnabled	If the driver/device should work in loopback mode
baudRate	The baudrate to be set for the HW Instance
stopBits	Number of stop bits for data transfer
charLen	Data word length for Tx/Rx
Parity	parity setting(Even/odd/No parity)
rxThreshold	FIFO data threshold for RX to raise a receive data interrupt
fc	Whether any flow control for data transfer should be used and if to be used, what should be the type of flow control to be used?
edmaRxTC/edmaRxTC	EDMA TCs for transmit and receive
hwiNumber	The hardware interrupt number to which the event of the (current) UART instance is assigned.
polledModeTimeout	The data transfer timeout for polled mode of operation

2.4.3 Uart_ChanParams

Applications could use this structure to configure the channel specific configurations. This is provided when driver channels are created (e.g. stream_create)

Members	Description
hEdma	Handle to the EDMA driver. Used in the EDMA mode of operation.

2.4.4 Polled Mode

The configurations required for polled mode of operation are:

Instance configuration *opMode* should be set to Uart_OpMode_POLLED. Additionally the timeout parameter for the data transfer operation can be configured as required. For example, polledModeTimeout could be set to 1000000 ticks, while the default value is BIOS_WAIT_FOREVER.

2.4.5 Interrupt Mode

The configurations required for interrupt mode of operation are:

Instance configuration *opMode* should be set to Uart_OpMode_INTERRUPT. Additionally the *hwiNumber* assigned by the application for the UART CPU events group should be passed, so that the driver can enable proper interrupts. It is recommended to start from the sample application and modify it further to meet the need of the actual application.

2.4.6 DMA Interrupt Mode

The configurations required for DMA Interrupt mode of operation are:

Instance configuration *opMode* should be set to Uart_OpMode_DMAINTERRUPT. Additionally the *hwiNumber* assigned by the application for the UART CPU events group should be passed, so that the driver can enable proper interrupts. Also, as part of *chanParams*, the handle to the EDMA driver, hEdma, should be passed by the application. Note that *"UART_EDMA_SUPPORT"* variable should be supplied as a compiler switch for proper operation in this mode so the sample application initializes the edma driver and passes the appropriate *chanParams.*

Note: The module configuration variable *edmaEnable* should be set to true so that the EDMA code is enabled.

2.5 Control Commands

Following table describes some of important the control commands, for a comprehensive list please refer the IOCTL defined in `Uart.h(Uart.xdc)`.

Command	Arguments	Description
Uart_IOCTL_SET_BAUD	Uart_BaudRate *	Configures the baud rate for the UART instance
Uart_IOCTL_SET_STOPBITS	Uart_NumStopBits *	Configures the number of stop bits for the instance
Uart_IOCTL_SET_DATABITS	Uart_NumStopBits *	Configures the word length for transmission and reception
Uart_IOCTL_SET_PARITY	Uart_Parity *	Configures the parity for data transmission and reception
Uart_IOCTL_SET_FLOWCONTROL	Uart_FlowControl *	Configures the flow control for the data transmission/reception
Uart_IOCTL_SET_TRIGGER_LEVEL	Uart_RxTrigLvl *	Configures the trigger level the receive fifo full level

Uart_IOCTL_RESET_RX_FIFO	None	Resets the hardware receive FIFO
Uart_IOCTL_RESET_TX_FIFO	None	Resets the hardware transmit FIFO
Uart_IOCTL_CANCEL_CURRENT_IO	None	Cancels the current IO operation request I progress
Uart_IOCTL_GET_STATS	Uart_Stats *	Passes the statistics of driver operation to the user
Uart_IOCTL_CLEAR_STATS	None	Resets/Clears the driver statistics
Uart_IOCTL_FLUSH_ALL_REQUEST	None	Cancels all the I/O operations queued
Uart_IOCTL_SET_POLLEDMODETIMEOUT	Uint32 *	Change the value for polled mode timeout

2.6 Use of UART driver through Stream APIs

Following sections explain the use of parameters of Stream calls in the context of UART driver. Note that no effort is made to document the use of Stream calls; any UART specific requirements are covered below.

2.6.1 Stream_create

Parameter Number	Parameter	Specifics to UART
1	Device Name string	Unique identifier used to identify this driver. Please note the name should be same as specified while creating the driver. (Either through CFG or Uart_create ()
2	IO Type	Should be "DriverTypes_INPUT" when UART requires to received data and "DriverTypes_OUTPUT" when UART requires to transmit
3	Stream_Params *	Parameters required for the creation of the stream (e.g. channel parameters)
4	Error_Block *	Pointer to the application supplied error block

2.6.2 Stream_control

Parameter Number	Parameter	Specifics to UART

1	Stream_handle	Handle returned by `Stream_create`
2	Command	IOCTL command defined by UART driver
3	Arguments	Misc arguments if required by the command
4	Error_Block *	Pointer to the Application supplied error block

2.6.3 Stream_write/read

Parameter Number	Parameter	Specifics to UART
1	Channel Handle	Handle returned by `Stream_create`
3	Pointer to buffer	Should be pointer to variable of type pointer to Buffer.
4	Size	Size of the transaction
5	Error_Block *	Pointer to the Application supplied error block

2.6.4 Uart_create

Parameter Number	Parameter	Specifics to UART
1	Uart_params *	Pointer to the Uart_params structure required for the Driver creation

2.7 Sources that need re-targeting

2.7.1 SoC level changes

When the driver has to adapt to SoC level changes the two files Uart.xs (Module Script File) and the SoC script file soc.xs need to be updated with the changes.

2.7.2 EVM level changes

None

2.8 Use of Global interrupt disable and enable

In functions like uartCancelCurrentIo and uartCancelAllIo, some parts of the code are guarded with _disable_interrupts(..) and _restore_interrupts(). This is because these functions are dynamically called during IOCTL calls. The enable and disable interrupt calls avoid context switch from any new interrupts.

2.9 EDMA3 Dependency

When the UART driver is configured in EDMA mode (compile time configuration is needed) UART driver relies on EDMA3 LLD driver to move data from/to application buffers to peripheral; Please note that EDMA3 LLD driver would not be part of this release. Please ensure that current PSP release is compliant with version of EDMA3 driver being used from the system requirements section of this document.

2.9.1 Used Paramset of EDMA 3

When the UART driver is configured to operate in the EDMA mode of operation, it required two EDMA channels (1 TX, 1 RX) for the operation. These channels are reserved for the TX and RX events on every instance basis. These channels are used for the life time of the driver. Please refer to the soc user guide for the exact channels available for the UART. The UART driver **does not** use any spare paramsets.

2.10 Known Issues

Please refer to the top level release notes that came with this release.

2.11 Limitations

Please refer to the top level release notes that came with this release.

2.12 Uart Sample application

2.12.1 Description

This sample demonstrates the use of the Uart driver in polled, interrupt and edma modes.

The Uart driver is configured statically in uartSample.cfg file. The uartParams used in Uart.create is globally declared in uartSample.cfg file.

The uartSample.cfg file contains the remaining BIOS configuration. The most important lines in this file which the application may need to pull into his cfg file are as follows.

ECM.eventGroupHwiNum[0] = 7;

ECM.eventGroupHwiNum[1] = 8;

ECM.eventGroupHwiNum[2] = 9;

ECM.eventGroupHwiNum[3] = 10;

These lines configure the ECM module and map Uart events to CPU interrupts (e.g. if the Uart event number is 69 which falls in ECM group 2. Here ECM group 2 is mapped to HWI_INT9).

The main() function configures the PINMUX and uses the Psc module to enable the Uart peripheral.

The echo() task exercises the Uart driver. It uses Stream APIS to create uart channels and read and write to them.

2.12.2 Build

This sample can be built using the path

<ID>/psp/examples/evmDA830/uart

IMPORTANT NOTE: .cdtbuild contains references to %EDMA3LLD_BIOS6_INSTALLDIR% environment variable and links with edma3

libraries. This is required because by default the Uart driver library is built with –EDMA ENABLE.

There is also another XDC based project file available for users familiar with XDC build

<ID>/psp/examples/evmDA830/uart/package.bld

This project file includes uartSample.cfg which brings in all the required packages.

This project requires setup of XDCPATH environment variable. The XDCPATH must contain the following -

<EDMA3_INSTALL_DIR>/packages; <PSPDRIVERS_INSTALL_DIR>/packages;

2.12.3 Setup

You need to connect the evmDA830 to the host PC using a NULL Model cable. On the host PC an application like HyperTerminal or Terra-Term needs to be setup for appropriate COM port, baud rate, stop bits etc.

2.12.4 Output

When the sample runs, it will output the following string to the Uart output channel.

"UART Demo Starts: INPUT a file of size 1000 bytes".

The user needs to type or send 1000 bytes. This sample application will echo the received characters to the terminal. The user could make use of the "**sample.txt**" file provided with the package at ti\psp\examples\evmDA830\uart. This file is a test file which contains 1000 characters of data.

3 I2C driver

3.1 Introduction

This document is the reference guide for the I2C device driver which explains the features and tips to use them.

DSP/BIOS applications use the driver typically through APIs provided by Stream layer, to transmit and receive serial data. The following sections describe in detail, procedures to use this driver, configure among others…. It is recommended to go through the sample application to get a feel of initializing and using the I2c driver

3.1.1 Key Features

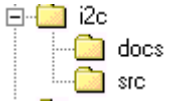
- Multi instantiable and re-entrant driver
- Each instance can operate as an receiver and/or transmitter
- Supports Polled, Interrupt and DMA Interrupt Mode of operation

3.2 Installation

The I2c device driver is a part of PSP package for DA830 platform and would be installed as part of whole package installation. For high level design information please refer to the driver architecture guide that came with this package (available at <ID>\ti\psp\i2c\docs)

3.2.1 I2C Component folder

On installation of PSP package for DA830, the I2C driver can be found at <ID>\ ti\psp\i2c\



As show above the i2c folder contains sub-folder, contents of which are described below.

- **i2c -** The i2c folder is the place holder for the entire I2C driver, documents and the build configuration files. I2C driver is implemented as IDriver under DSP/BIOS™ operating system. Stream defined APIs could be used to interface to I2C driver. This folder contains the build configuration file(package.bld),the I2C module specification file (I2c.xdc),the module script file (I2c.xs) and the miscellaneous files required for compiling the I2C library.
- **docs –** Holds I2c driver's architecture. Please note that the API reference would be found as a part of cdoc help (<install dir>\packages\docs\DA830\cdoc\index.html)
- **src –** Place holder for I2C driver's source code.

3.2.2 Build Options

I2C device driver will be built with "whole_program_debug" mode. When built successfully the respective library will be available at <ID>\ti\psp\i2c\lib\ < ti.psp.i2c.a674>

3.3 Features

This section details the features of I2C and how to use them in detail.

3.3.1 Multi-Instance

The I2C driver can operate on all the instance of I2C on DA830. Different instances are specified during driver creation time. Supported instance are 0 through 2 with device ID 0 through 2 respectively.

These instances could be operated simultaneously with configurations supported by I2C driver.

The device ID could be specified using the `instNum` field of structure I2c_Params. There are two ways in which a new instance of the I2C driver can be created.

3. Static creation – static creation of the driver is done in the "cfg" file of the application. The creation happens at compile time.
4. Dynamic creation – Dynamic creation of driver is done in the application source files and the creation happens at runtime.

(i.e. I2c_Params.`instNum = 0x0`, I2c_Params.`instNum = 0x1`, so on…)

3.3.2 Each Instance as Transmitter and/or receiver

Each I2C driver instance can be used for transmit and receive operation (only in half duplex mode due to nature of I2C peripheral). This could be achieved by opening a stream Channel as an INPUT channel and opening a stream Channel as an OUTPUT channel. The type of Channel is specified while creating the channel (using `Stream_create ()specify "DriverTypes_OUTPUT" or "DriverTypes_INPUT"`). The configuration parameters are explained in the sections to follow.

3.4 Configurations

Following tables document some of the configurable parameter of I2C. Please refer to I2c.h (I2c.xdc) for complete configurations and explanations.

3.4.1 Module configuration

The following parameters are module wide configurable parameters

Variable	Description
`paramCheckEnable`	Option to select if parameter checking at the function entry should be enabled or disabled.
`edmaEnable`	Option to be enabled during the operation of the I2c in EDMA mode.

3.4.2 I2c_Params

This structure defines the device configurations, expected to supply while creating the driver. This is provided when driver channels are created (e.g. stream_create).

Members	Description
`instNum`	Instance number of the driver.
`enableCache`	Whether the driver should deal with cache operation in DMA Interrupt (EDMA) mode of operation. Default is true. If this is not set the user has to take care of the cache operations

opMode	Whether the I2C driver should operate in Polled or Interrupt or DMA Interrupt (EDMA) Mode
ownAddr	Own address of the I2C instance being instantiated. This is programmable and is provided as a parameter here.
loopbackEnabled	If the driver/device works in loopback mode
numBits	The number of bits per data word to be for the data transfer operations
busFreq	The frequency at which the clock (SCL) should operate
addressing	Whether 7 bit addressing or extended (10-bit) addressing mode is used. This is a Boolean – false for 7 bit addressing and true for extended addressing
edma3EventQueue	The EDMA event queue the application will use in DMA Interrupt (EDMA) mode of operation. Default is queue number zero.
hwiNumber	The hardware interrupt number assigned for I2C events
polledModeTimeout	The data transfer timeout for polled mode of operation

3.4.3 I2c_ChanParams

Applications could use this structure to configure the channel specific configurations.

These channel parameters are supplied as part of the chanParams in the Stream Parameters structure during channel open via Stream_create() API. This configures the operation mode and communication mode of the channel.

Members	Description
hEdma	The handle to the EDMA driver. Required only when operating in DMA interrupt mode. Also, note that when operating in DMA interrupt mode, that module configuration variable edmaEnable should be set to true.
masterOrSlave	Whether the instance/channel is in Master mode or Slave mode

hEdma is assigned the value of "hEdma" system wide variable. This is available only when the EDMA driver is used and linked with the final executable. This holds a valid value when a call to edm3init() is made. This is to be done by the application. This is demonstrated in the sample application and guarded by the macro I2C_EDMA_SUPPORT. This macro should be enabled in EDMA mode of operation.

3.4.4 Polled Mode

The configurations required for polled mode of operation are:

Instance configuration opMode should be set to I2c_OpMode_POLLED. Additionally the timeout parameter for the data transfer operation can be configured as required. For example, polledModeTimeout could be set to 1000 Ticks, while the default value is BIOS_WAIT_FOREVER.

3.4.5 Interrupt Mode

The configurations required for interrupt mode of operation are:

Instance configuration opMode should be set to I2c_OpMode_INTERRUPT. Additionally the hwiNumber assigned by the application for the I2C CPU events group should be passed, so that the driver can enable proper interrupts.

It is recommended to start from the sample application and modify it further to meet the need of the actual application.

3.4.6 DMA Interrupt Mode

The configurations required for DMA Interrupt mode of operation are:

Instance configuration opMode should be set to I2c_OpMode_DMAINTERRUPT. Additionally the hwiNumber assigned by the application for the I2C CPU events group should be passed, so that the driver can enable proper interrupts. The module configuration variable edmaEnable should be set to true. Also, as part of chanParams, the handle to the EDMA driver, hEdma, should be passed by the application.

Note that "I2C_EDMA_SUPPORT" variable should be supplied as a compiler switch for proper operation in this mode so the sample application initializes the EDMA driver and passes the appropriate channel parameters.

3.4.7 I2c_DataParam

The I2c_DataParam structure is one the most important structures that needs to be passed as a buffer in the Stream_read/write calls.

For I2C communication, the device needs not just the actual data for transfer but additional details also like the address of the device that it should communicate to, communication control bit flags (START/STOP etc) and any other parameters as demanded by the case. All these are collected under one structure called the DataParam structure.

Members	Description
slaveAddr	The address of the slave device that this data transfer operation is intended for
buffer	The actual data that should be sent out on the SDA line
bufLen	The length of the data that should be sent out in the SDA line
flags	The flags for current data transfer (explained below)
param	Reserved for future use

The flags member of the DataParam structure defines the control signal that are needed to be generated for the current operation. For example, if slave device demands that current transfer should not generate a stop bit, then this can be controlled by not specifying the I2C_STOP flag in the flags member. However, please note that the flags should contain a meaningful combination for the current transfer and should be supported on the instance and the slave device for that transfer.

3.5 Control Commands

Following table describes some of important the control commands, for a comprehensive list please refer the ICOTL defined in `I2c.h(I2c.xdc)`.

Command	Arguments	Description
I2c_IOCTL_SET_BIT_RATE	UInt32 *	Configures the bus frequency for the I2C instance
I2c_IOCTL_GET_BIT_RATE	UInt32 *	Passes the current bus frequency for the I2C instance
I2c_IOCTL_CANCEL_PENDING_IO	None	Cancels all the pending I/O requests
I2c_IOCTL_BIT_COUNT	UInt32 *	Configures the data bit length for transmission and reception
I2c_IOCTL_NACK	None	Configures the I2C instance to generate NACK when required
I2c_IOCTL_SET_OWN_ADDR	UInt32 *	Configures the own address for current instance
I2c_IOCTL_GET_OWN_ADDR	UInt32 *	Passes the current own address set for the current instance
I2c_IOCTL_SET_POLLEDMODETIMEOUT	UInt32 *	Change the value for polled mode timeout

3.6 Use of I2C driver through Stream APIs

Following sections explain the use of parameters of Stream calls in the context of I2C driver. Note that no effort is made to document the use of Stream calls; any I2C specific requirements are covered below.

3.6.1 Stream_create

Parameter Number	Parameter	Specifics to I2C
1	Device Name string	Unique identifier used to identify this driver. Please note the name should be same as specified while creating the driver. (Either through CFG or I2c_create `()`
2	IO Type	Should be "`DriverTypes_INPUT`" when I2C requires to received data and "`DriverTypes_OUTPUT`" when I2C requires to transmit
3	Stream_Params *	Parameters required for the creation of the stream (e.g. channel parameters)
4	Error_Block *	Pointer to the application supplied error block

3.6.2 Stream_control

Parameter Number	Parameter	Specifics to I2C
1	Stream_handle	Handle returned by `Stream_create`
2	Command	IOCTL command defined by I2C driver
3	Arguments	Misc arguments if required by the command
4	Error_Block *	Pointer to the Application supplied error block

3.6.3 Stream_write/read

Parameter Number	Parameter	Specifics to I2C
1	Channel Handle	Handle returned by `Stream_create`
3	Pointer to buffer	Should be pointer to type `DataParam Structure` that holds the I2C transfer information like buffer, slave address and other flag. This is explained below
4	Size	Size of the transaction
5	Error_Block *	Pointer to the Application supplied error block

3.6.4 I2c_create

Parameter Number	Parameter	Specifics to I2C
1	I2c_params *	Pointer to the I2c_params structure required for the Driver creation

3.7 Sources that need re-targeting

3.7.1 SoC level changes

When the driver has to adapt to SoC level changes the two files I2c.xs (Module Script File) and the SoC script file soc.xs need to be updated with the changes.

3.7.2 EVM level changes

None

3.8 EDMA3 Dependency

When the I2C driver is configured in EDMA mode (compile time configuration is needed) I2C driver relies on EDMA3 LLD driver to move data from/to application buffers to peripheral; Please note that EDMA3 LLD driver would not be part of this release. Please ensure that current PSP release is compliant with version of EDMA3 driver being used from the system requirements section of this document.

3.8.1 Used Paramset of EDMA 3

I2C driver uses TWO paramsets of EDMA3; if there are no paramsets are available the I2C driver creation would fail. These paramsets are used through the life time of I2C driver. No link paramsets are used.

3.9 Known Issues

Please refer to the top level release notes that came with this release.

3.10 I2C Sample application

3.10.1 Description

This sample demonstrates the use of the I2c driver in polled, interrupt and edma modes.

This example uses the I2c bus to write an array of data to the CAT24WC256 EEPROM memory of the evmDA830. Once the data has been written, the I2c bus again is used to read the same data from the EEPROM memory. The data read is then compared with the data that was written, and if it matches then the operation is considered a success.

The reads and writes to the EEPROM memory are accomplished by use of both the I2c and the Stream modules, in combination. The I2c driver is used to configure and set up the I2c bus, and the Stream module APIs are used to perform the actual reads and writes to the EEPROM memory, via the I2c bus.

The I2c driver is configured statically in the i2cSample.cfg file.

The i2cSample.cfg file contains important BIOS configuration settings, which are required in order for the I2c driver to work properly. The most important lines in this file are:

```
ECM.eventGroupHwiNum[0] = 7;
ECM.eventGroupHwiNum[1] = 8;
ECM.eventGroupHwiNum[2] = 9;
ECM.eventGroupHwiNum[3] = 10;
```

The above configuration settings are needed to correctly set up the ECM module and map the I2c event to CPU interrupt. For example the I2c event number is 36, which falls under ECM group 1. Here ECM group 1 is mapped to HWI_INT8, and this is the HWI number used when configuring i2cParams at runtime (explained further below).

Further I2c static configuration is done in the i2cSample.cfg file, which uses the I2c instance parameters (i2cParams) to create I2C instance using I2c.create API.

Once initialization has completed, the main() function runs, configuring the PINMUX. Following this, the user defined task "echoTask()" runs, which creates Stream I2c read and write handles. These handles are then used when calling the Stream_write() and Stream_read() APIs to actually write and read data to and from the EEPROM memory.

3.10.2 Build

This sample can be built using path

<ID>/psp/examples/evmDA830/i2c

IMPORTANT NOTE: .cdtbuild contains references to %EDMA3LLD_BIOS6_INSTALLDIR% environment variable and links with edma3 libraries. This is required because by default the I2c driver library is built with EDMA ENABLE.

There is also another XDC based project file available for users familiar with XDC build.

<ID>/psp/examples/evmDA830/i2c/package.bld

This project file includes i2cSample.cfg which brings in all the required packages.

This project requires setup of XDCPATH environment variable. The XDCPATH must contain the following -

<EDMA3_INSTALL_DIR>/packages; <PSPDRIVERS_INSTALL_DIR>/packages;

3.10.3 Setup

No special setup is needed to run the I2c example.

Warning: Please note that the sample application erases the EEPROM during the execution, before it starts with the read/write test.

3.10.4 Output

After the completion of read/write operation with the following messages are printed on the CCS console

I2C :Start of I2C sample application

I2C CAT24WC256 EEPROM write/read test started

I2C CAT24WC256 EEPROM Read/write test passed

I2C :End of I2C sample application

4 SPI driver

4.1 Introduction

This section is the reference guide for the SPI device driver which explains the features and tips to use them.

DSP/BIOS applications use the SPI driver typically through APIs provided by Stream layer, to transmit and receive serial data. The following sections describe in detail, procedures to use this driver, configure among others…. It is recommended to go through the sample application to get a feel of initializing and using the Spi driver

4.1.1 Key Features

- Multi-instantiable and re-entrant driver
- Each instance can operate as an receiver and or transmitter
- Supports Polled, Interrupt and DMA Interrupt Mode of operation

4.2 Installation

The Spi device driver is a part of PSP package for DA830 platform and would be installed as part of whole package installation. For high level design information please refer to the driver architecture guide that came with this package (available at <ID>\ti\psp\spi\docs).

4.2.1 SPI Component folder

On installation of PSP package for DA830, the SPI driver can be found at <ID>\ ti\psp\spi\

As show above the spi folder contains sub-folder, contents of which are described below.

- **spi -** The spi folder is the place holder for the entire SPI driver, documents and the build configuration files. SPI driver is implemented as IDriver under DSP/BIOS™ operating system. Stream defined APIs could be used to interface to SPI driver. This folder contains the build configuration file(package.bld),the SPI module specification file (Spi.xdc),the module script file (Spi.xs) and the miscellaneous files required for compiling the SPI library.
- **docs –** Holds Spi driver's architecture. Please note that the API reference would be found as a part of cdoc help (<install dir>\packages\docs\DA830\cdoc\ index.html)
- **src –** Place holder for SPI driver's source code.

4.2.2 Build Options

SPI device driver will be built with "whole_program_debug" mode. When built successfully the respective library will be available at <ID>\ti\psp\spi\lib\ < ti.psp.spi.a674>

4.3 Features

This section details the features of SPI and how to use them in detail.

4.3.1 Multi-Instance

The SPI driver can operate on all the instance of SPI on DA830. Different instances are specified during driver creation time. Supported instance are 0 through 2 with device ID 0 through 2 respectively.

These instances could be operated simultaneously with configurations supported by SPI driver.

The device ID could be specified using the `instNum` field of structure Spi_Params. There are two ways in which a new instance of the SPI driver can be created.

1. Static creation – static creation of the driver is done in the "cfg" file of the application. The creation happens at compile time.
2. Dynamic creation – Dynamic creation of driver is done in the application source files and the creation happens at runtime.

(i.e. Spi_Params.`instNum = 0x0`, Spi_Params.`instNum = 0x1`, so on…)

4.3.2 Each Instance as Transmitter and / or receiver

Each SPI instance can be used for creating channels for transmit and receive operation. The same channel can be used for both transmit and receive operation. This could be achieved by opening a stream Channel as an INOUT channel . The type of Channel is specified while creating the channel (using `Stream_create ()specify "DriverTypes_INOUT"`). The configuration parameters are explained in the sections to follow.

4.4 Configurations

Following tables document some of the configurable parameter of SPI. Please refer to Spi.h (Spi.xdc) for complete configurations and explanations.

4.4.1 Module configuration

The following parameters are module wide configurable parameters

Variable	Description
`paramCheckEnable`	Option to select if parameter checking at the function entry should be enabled or disabled.
`edmaEnable`	Option to be enabled during the operation of the SPI in EDMA mode.

4.4.2 Spi_Params

This structure defines the device configurations, expected to supply while creating the driver.

Members	Description
`instNum`	Hardware instance number of the SPI to be created.(0-2 are available on DA830).
`opMode`	Selects the driver mode of operation. (polled mode, interrupt mode or edma mode).

outputClkFreq	The output clock frequency the SPI instance should generate in case of master mode of operation. This will be internally used by the driver to calculate and program the divider values.
loopbackEnabled	option used to configure if the driver should be in loop back mode (TRUE – loopback is enabled, FALSE – loopback mode is disabled).
spiHWCfgData	Structure containing the hardware specific information.
hwiNumber	The hardware interrupt number assigned for SPI events
polledModeTimeout	This specifies the timeout value to be used for each transfer operation. It is used only in the POLLED mode of operation of the driver

4.4.3 Spi_ChanParams

Applications could use this structure to configure the channel specific configurations.

Members	Description
hEdma	The handle to the EDMA driver. Required only when operating in DMA interrupt mode. Also, note that when operating in DMA interrupt mode, that module configuration variable edmaEnable should be set to true.

4.4.4 Spi_DataParam

This buffer is used to submit data transfer requests to the SPI driver.

Members	Description
outBuffer	Pointer to the output buffer specified by the application. Can be specified as NULL in case of only read operation
inBuffer	Pointer to the buffer to hold the input data. Can be specified as NULL in case of only write operation.
bufLen	Total buffer length. Should be the size of the total transceive operation.
chipSelect	The chip select to be used for selecting the slave device.
dataFormat	The data format to be used by the SPI (out of the 4 different data formats supported by it.)
flags	Flags to indicate the current operation (Read/write etc).
param	Parameter kept for future use.

Note: The SPI driver is in tranceive mode hence it is required to provide both the input and output buffers in case of a transceive operation. In case that the application wants to perform either a read only or write only operation, it is sufficient for it to provide the input buffer or the output buffer only. The other buffer can be specified as NULL.

4.4.5 Polled Mode

The configurations required for polled mode of operation are:

Instance configuration opMode should be set to Spi_OpMode_POLLED. Additionally the timeout parameter for the data transfer operation can be configured as required. For example, polledModeTimeout could be set to 1000 Ticks, while the default value is WAIT_FOREVER.

For polled mode of operation the driver does not implement the task sleeping in in between checks for data ready status, during data transfer. This is because, while in sleep the data may arrive and the data may go unread. This can be more prevalent with increasing data clock frequencies. This non use of task sleep result in a tight while loop for checking data ready status during transfers and may block out other tasks in the system from executing, for the timeout duration set by the user. Hence, it is advised that in slave mode interrupt mode of operation may be used.

4.4.6 Interrupt Mode

The configurations required for interrupt mode of operation are:

Instance configuration opMode should be set to Spi_OpMode_INTERRUPT. Additionally the hwiNumber assigned by the application for the SPI CPU events group should be passed, so that the driver can enable proper interrupts.

4.4.7 DMA Interrupt Mode

The configurations required for DMA Interrupt mode of operation are:

Instance configuration opMode should be set to Spi_OpMode_DMAINTERRUPT. Additionally the hwiNumber assigned by the application for the SPI CPU events group should be passed, so that the driver can enable proper interrupts. The module configuration variable edmaEnable should be set to true. Also, as part of chanParams, the handle to the EDMA driver, hEdma, should be passed by the application. It is recommended to start from the sample application and modify it further to meet the need of the actual application.

4.4.8 Slave Mode

The option of slave mode (or master mode) of operation, should be supplied along with the HWConfig (device parameter) structure (masterOrSlave field) in device parameters, while instantiation of the module. This is because the mode of operation is fixed for one instance and cannot be changed dynamically or per-channel per instance. Also note that in slave mode of the device only one channel can be opened. Note that "SPI_EDMA_SUPPORT" variable should be supplied as a compiler switch for proper operation in this mode so the sample application initializes the EDMA driver and passes the appropriate channel parameters.

4.5 Control Commands

Following table describes some of important the control commands, for a comprehensive list please refer the ICOTL defined in `Spi.h(Spi.xdc)`.

Command	Arguments	Description
Spi_IOCTL_CANCEL_PENDING_IO	None	Cancels all the pending I/O requests

Spi_IOCTL_SET_CS_POLARITY	Bool *	Configures the CS polarity to High or Low
Spi_IOCTL_SET_POLLEDMODETIMEOUT	UInt32 *	To change the value for polled mode timeout

4.6 Use of SPI driver through Stream APIs

Following sections explain the use of parameters of Stream calls in the context of SPI driver. Note that no effort is made to document the use of Stream calls; any SPI specific requirements are covered below.

4.6.1 Stream_create

Parameter Number	Parameter	Specifics to SPI
1	Device Name string	Unique identifier used to identify this driver. Please note the name should be same as specified while creating the driver. (Either through CFG or Spi_create ()
2	IO Type	Should be "DriverTypes_INPUT" when SPI requires to received data and "DriverTypes_OUTPUT" when SPI requires to transmit
3	Stream_Params *	Parameters required for the creation of the stream (e.g. channel parameters)
4	Error_Block *	Pointer to the application supplied error block

4.6.2 Stream_control

Parameter Number	Parameter	Specifics to SPI
1	Stream_handle	Handle returned by Stream_create
2	Command	IOCTL command defined by SPI driver
3	Arguments	Misc arguments if required by the command
4	Error_Block *	Pointer to the Application supplied error block

4.6.3 Stream_write/read

Parameter Number	Parameter	Specifics to SPI

1	Channel Handle	Handle returned by `Stream_create`
3	Pointer to buffer	Should be pointer to DataParam structure that holds the buffer, Chipselect and other flag
4	Size	Size of the transaction
5	Error_Block *	Pointer to the Application supplied error block

4.6.4 Spi_create

Parameter Number	Parameter	Specifics to SPI
1	Spi_params *	Pointer to the Spi_params structure required for the Driver creation

Note: Since the SPI data transfer operation is a transceiver operation, only "Stream_write" API is used for both write and Read operations. Also note that the channel will also be created only in DriverTypes_INOUT mode.

4.7 Use of GPIO as chip select

In some cases more than one chip select signal lines/signals might be required. In this case since the DA830 SoC supports only one chip select, an unused GPIO pin could be used to generate chip select signal/lines.

The SPI driver supports this feature of using a GPIO pin as chip select, by using GPIO module calls internally. (Please refer to GPIO CDOC and user guide for details on GPIO module)

Following are the steps to enable and use this feature in the applications:

1. Creation of GPIO instance

a. Create a handle to the GPIO module in the application cfg file:

Example:

```
/* Create a new params structure for GPIO */
var gpioPrms = new Gpio.Params();
/* Intialize the params structure */
gpioPrms.instNum = 0;
/* Let us assume GP0_13 –One needs to mark this pin and the associated back as not in use as anything else in the system. Also, in this use case ignore hwiNum */
gpioPrms.BankParams[0].inUse = Gpio.InUse_No;
gpioPrms.BankParams[0].hwiNum  = -1;
/* Mark 13th pin as not in use for anything else */
gpioPrms.BankParams[0].PinConfInfo[12].inUse = Gpio.InUse_No;
/* Create the GPIO instance and store it as Spi module global variable. This instance will be used to acted upon the GPIO using GPIO module APIs*/
```

Spi.gpio0 = Gpio.create(gpioPrms)

b. Proceed with rest of initialization for SPI and other modules. Please specify the GPIO pin number to be used as CS.

Example:

spiPrms.spiHWCfgData.gpioPinNo = 13

2. GPIO pin as chip select for each data transfer

a. The driver facilitates selection between the CS signal or GPIO signal to be used as Chip Select, for every transfer. If Spi_DataParam.flags contains Spi_GPIO_CS then GPIO line will be used as chip select else, the CS signal will be used as chip select. Thus, each transfer (read/write) could be destined for a slave on CS or GPIO.

Example:

```
Spi_DataParam    dataparam;
/* May contain other flags like Spi_CSHOLD */
 dataParam.flags = Spi_GPIO_CS;
 Here the slave on GPIO is selected, else the slave on CS selected
```

3. Note:

The chip select signal generated on the GPIO pin has the following constraints:

a. This, GPIO as chip select, feature is done by driver in software. Hence, it may not satisfy the strict timing requirements like a normal CS signal. For instance, the GPIO used as chip select is activated and deactivated just before actually writing the first word into SPIDAT and deactivated after a data transfer (word or whole request, depending on Spi_CSHOLD in Spi_DataParam.flags) is complete. So, here one can see that GPIO chip select is activated a little earlier than required and deactivated a little later than required. This adds to some latency in throughput of transfers.

b. In EDMA mode GPIO as chip select feature is available only if Spi_CSHOLD flag is included in the Spi_DataParams.flags for every transfer. This is beacuse, if CSHOLD is not set the GPIO chip select, must be toggled at every word transfer. However, in EDMA mode, the data transfer is done by EDMA master and the SPI driver does not have knowledge of word-by-word transfer. The only control points for the driver are at the start of EDMA transfer, at which the GPIO is activated and at the end of request for transfer (EDMA callback) where it is deactivated. This limits the availability of GPIO as chip select feature only with CSHOLD enabled in EDMA mode of operation.

c. Only one GPIO can be used for chip select. Hence only one additional Chip Select can be used.

4.8 Sources that need re-targeting

4.8.1 SoC level changes

When the driver has to adapt to SoC level changes the two files Spi.xs (Module Script File) and the SoC script file soc.xs need to be updated with the changes.

4.8.2 EVM level changes

None

4.9 EDMA3 Dependency

When the SPI driver is configured in EDMA mode (compile time configuration is needed) SPI driver relies on EDMA3 LLD driver to move data from/to application buffers to peripheral; Please note that EDMA3 LLD driver would not be part of this release. Please ensure that current PSP release is compliant with version of EDMA3 driver being used from the system requirements section of this document.

4.9.1 Used Paramset of EDMA 3

SPI driver uses TWO paramsets of EDMA3; if there are no paramsets are available the SPI driver creation would fail. These paramsets are used through the life time of SPI driver. No link paramsets are used.

4.10 Known Issues

Please refer to the top level release notes that came with this release.

4.11 Limitations

Please refer to the top level release notes that came with this release.

4.12 Spi Sample application

4.12.1 Description

This sample demonstrates the use of the Spi driver in polled, interrupt and edma modes.

This example uses the Spi bus to write an array of data to the W25X32 Spi Flash memory of the evmDA830. Once the data has been written, the Spi bus again is used to read the same data from the Spi Flash memory. The data read is then compared with the data that was written, and if it matches then the operation is considered a success.

The reads and writes to the Spi Flash memory are accomplished by use of both the Spi and the Stream modules, in combination. The Spi driver is used to configure and set up the Spi bus, and the Stream module APIs are used to perform the actual reads and writes to the Spi Flash memory, via the Spi bus.

The Spi driver is configured both statically in the spiSample.cfg file, as well as at run time in the spiSample_main.c and spiSample_io.c files.

The spiSample.cfg file contains important BIOS configuration settings, which are required in order for the Spi operations to work properly. The most important lines in this file are:

```
ECM.eventGroupHwiNum[0] = 7;
ECM.eventGroupHwiNum[1] = 8;
ECM.eventGroupHwiNum[2] = 9;
ECM.eventGroupHwiNum[3] = 10;
```

The above configuration settings are needed to correctly set up the ECM module and map the Spi event to CPU interrupt. For example the Spi event number is 37, which falls under ECM group 1. Here ECM group 1 is mapped to HWI_INT8, and this is the HWI number used when configuring spi Params (explained further below).

Further Spi static configuration is done in the spiSample.cfg file, which uses the Spi instance parameters (spi Params) to create Spi instance using Spi.create API.

Once initialization has completed, the main() function runs, configuring the PINMUX. Following this, the user defined task "echoTask()" runs, which creates Stream Spi read and write handles. These handles are then used when calling the Stream_write() and Stream_read() APIs to actually write and read data to and from the Spi Flash memory.

4.12.2 Build

This sample can be built using path

<ID>/psp/examples/evmDA830/spi

IMPORTANT NOTE: .cdtbuild contains references to %EDMA3LLD_BIOS6_INSTALLDIR% environment variable and links with edma3 libraries. This is required because by default the Spi driver library is built with EDMA ENABLE.

There is also another XDC based project file available for users familiar with XDC build.

<ID>/psp/examples/evmDA830/spi/package.bld

This project file includes spiSample.cfg which brings in all the required packages.

This project requires setup of XDCPATH environment variable. The XDCPATH must contain the following -

<EDMA3_INSTALL_DIR>/packages; <PSPDRIVERS_INSTALL_DIR>/packages;

4.12.3 Setup

No special setup is needed to run the I2c example

4.12.4 **Output**

After successful completion of read/write operation following message is printed on the CCS console.

BIOS SPI:SPI sample transceive ended succesfully

5 GPIO module

5.1 Introduction

This document is the reference guide for the GPIO module which explains the features and tips to use the GPIO module in an application.

The GPIO module provides the APIs that are typically required by the application for the control of the GPIO pins. The following sections describe in detail, procedures to use this module, configure among others…. It is recommended to go through the sample application to get a feel of initializing and using the GPIO module.

5.1.1 Key Features

- Multi-instanceable and re-entrant module
- Standalone module and does not inherit IDriver and does not require streams
- Each instance can be used to operate upon GPIO settings
- Supports registering of interrupts and setting of GPIO availability status

5.2 Installation

The GPIO module is a part of PSP package for DA830 platform and would be installed as part of whole package installation. For high level design information please refer to the module architecture guide that came with this package (available at <ID>\ti\psp\gpio\docs)

5.2.1 GPIO Component folder

On installation of PSP package for DA830, the GPIO module can be found at <ID>\ ti\psp\gpio\

As show above the gpio folder contains sub-folder, contents of which are described below.

- **gpio -** The gpio folder is the place holder for the entire GPIO module, documents and the build configuration files. GPIO is implemented as a module under DSP/BIOS™ operating system. GPIO APIs can be used directly y to configure the GPIO banks pins and interrupts among others. This folder contains the build configuration file (package.bld), the GPIO module specification file (Gpio.xdc), the module script file (Gpio.xs) and the miscellaneous files required for compiling the GPIO library.
- **docs –** Holds GPIO module's architecture. Please note that the API reference would be found as a part of cdoc help (<install dir>\packages\docs\DA830\cdoc\ index.html)
- **src –** Place holder for GPIO module's source code.

5.2.2 Build Options

GPIO module will be built with "whole_program_debug" mode. When built successfully the respective library will be available at <ID>\ti\psp\gpio\lib\ < ti.psp.gpio.a674>

5.3 Features

This section details the features of GPIO module and how to use them in detail.

5.3.1 Multi-Instance

The GPIO module can operate on all the instance of GPIO on DA830. Different instances are specified during instance creation time. Supported instance are 0 only with instNum (or device ID) 0 only.

These instances could be operated simultaneously with configurations supported by GPIO module.

The device ID could be specified using the `instNum` field of structure Gpio_Params. There are two ways in which a new instance of the GPIO module can be created.

1. Static creation – static creation of the module is done in the "cfg" file of the application. The creation happens at compile time.
2. Dynamic creation – Dynamic creation of module is done in the application source files and the creation happens at runtime.

(i.e. Gpio_Params.`instNum = 0x0`)

5.3.2 Operating on each Instance

GPIO module can be operated upon as a standalone module. This could be achieved by calling the GPIO module APIs directly by the application. The instance creation needs configuration parameters. The configuration parameters are explained in the sections to follow.

5.4 Configurations

Following tables document some of the configurable parameter of GPIO. Please refer to Gpio.h (Gpio.xdc) for complete configurations and explanations.

5.4.1 Module configuration

The following parameters are module wide configurable parameters. Applications can configure these settings according to their specific requirements.

Variable	Description
paramCheckEnable	Option to enable/disable the input parameter checking at the function entry point.

5.4.2 Gpio_Params

This structure is used for the device configuration of the PSC module. The PSC module expects the application to supply this configuration during the creation of the module instance.

Members	Description
instNum	Hardware instance of the GPIO bank being created
BankParams	The bank configuration parameters. This contains the availability of a bank and/or its associated pins as a GPIO as yes or no and the HWI number assigned to this

	bank or pin as applicable. The pin/bank must be marked as GPIO_InUse_Yes to let the module know that the said GPIO bank/pin is not available as a GPIO or the pin/bank must be marked as GPIO_InUse_No to let the module know that the said GPIO bank/pin is available as a GPIO. By default, all the GPIO bank/pin(s) are marked as not available, by the module in the BankParams. Hence, it is sufficient for the application to pass the information of those bank/pin(s) which are available as GPIO. This can be done in the CFG file of the application while static instantiation (as shown by the sample application) or during dynamic instantiation.

5.5 Use of GPIO through module APIs

Following sections explain the use of parameters of module calls in the context of GPIO module. Any GPIO specific requirements are covered below.

5.5.1 Gpio_create

This call creates the GPIO module instance and returns the handle to the module instance. . In the sample application provided with this package, the static creation of the GPIO instance is shown, using Gpio.create, which also return a handle to the GPIO module instance. This handle is used for any further API calls of the GPIO module. In the sample application provided, the GPIO instance handle obtained statically in the CFG file (Program.global.gpio0) is accessed in the C file, by reference it as an extern variable (extern GPIO_Handle gpio0;).

Parameter Number	Parameter	Specifics to GPIO
1	Gpio_params *	Pointer to the Gpio_params structure required for the module instance creation

5.6 Sources that need re-targeting

5.6.1 SoC level changes

When the moudle has to adapt to SoC level changes the two files Gpio.xs(Module Script File) and the SoC script file soc.xs need to be updated with the changes.

5.6.2 EVM level changes

None

5.7 EDMA3 Dependency

The GPIO module does not depend on the EDMA3 LLD driver. It does not support any data transfer operations.

5.8 I/O operations

The GPIO module does not support any streaming read or write operations over GPIO pins or banks. It just facilitates the user to form any such APIs as wrappers around the basic GPIO data in and out operation APIs on its pins/group of pins. For example, if a scenario exists for the application to use a GPIO pin for continuously sending a pulse train over it, then the user could use GPIO_setPinVal() API to set the value at that GPIO pin to 1 and 0 alternatively. However, the pulse width (on and off times) must be taken care of by the user by introducing suitable delays between successive call to the GPIO_setPinVal(). Thus the user needs to write a wrapper around the GPIO APIs to suit the needs of usage scenario.

5.9 Interrupt handler registration

The GPIO module facilitates the registration of interrupt handlers for GPIO pin/bank as applicable. Some (or all) of the GPIO bank or pins support rising of interrupt events to the CPU. These events should be mapped to a particular HWI interrupt by the application (if required to use the interrupts) and the same information is passed to the GPIO module via BankParams during instantiation. The user application then may register interrupt handlers for this interrupt event by using the GPIO_regIntHandler() API. The user needs to give the function which should be register as the handler. The GPIO module validates if the GPIO bank or pin has a valid event and hence can rise and interrupt or not, and then dispatches the interrupt handler for this event and enables the interrupt. However, note that the GPIO module just registers the function and does not handle the interrupt by itself. There is no interrupt context in the GPIO module.

5.10 Known Issues

Please refer to the top level release notes that came with this release.

5.11 Limitations

5.11.1 Multi-instance support

The GPIO module now supports only one instance, fixed number of banks (eight) and fixed number of pins per bank (sixteen). This is a limitation, as there are issues in getting initialization done for variable length arrays (inside structures, instance parameters etc) through the RTSC framework.

5.11.2 In Use status

The GPIO module provides the APIs, Gpio_(get/set)PinUseStatus and Gpio_(get/set)BankUseStatus for checking if the pin or bank is in use (as a functional pin and hence not available as GPIO). These, APIs should be used before calling any GPIO module APIs on setting data or status for the pins/banks. Though, GPIO module shall make explicit check for use status for individual pin operations, it does not do it for group (or all pins in a bank) operations since it becomes an overkill every time, especially if the group of pins is used for data transfer etc. Hence, the application should make this check at least once before use of the required GPIO pins and then can proceed.

Please refer to the top level release notes that came with this release.

6 PSC module

6.1 Introduction

This document is the reference guide for the PSC module which explains the features and tips to use the PSC module in an application.

DSP/BIOS applications use the module typically through raw APIs provided by PSC module. It is recommended to go through any of the provided sample applications to get a feel of using the PSC module to enable the clock for the respective devices.

6.1.1 Key Features

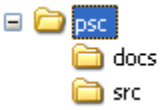
- Multi-instanceable and re-entrant module.
- Standalone module that does not inherit IDriver and also does not require streams.
- Each instance can be used to operate upon PSC settings

6.2 Installation

The PSC module is a part of PSP package for DA830 platform and would be installed as part of whole package installation. For high level design information please refer to the module architecture guide that came with this package (available at <ID>\ti\psp\psc\docs)

6.2.1 PSC Component folder

On installation of PSP package for DA830, the PSC module can be found at <ID>\ ti\psp\psc\



As show above the psc folder contains sub-folder, contents of which are described below.

- **psc -** The psc folder is the place holder for the entire PSC module, documents and the build configuration files. PSC is implemented as module under DSP/BIOS™ operating system. PSC can be used as RAW APIs for clock control of the required device. This folder contains the build configuration file (package.bld), the PSC module specification file (Psc.xdc), the module script file (Psc.xs) and the miscellaneous files required for compiling the PSC library.
- **docs –** Holds Psc module's architecture document. Please note that the API reference would be found as a part of cdoc help (<install dir>\packages\docs\DA830\cdoc\ index.html)
- **src –** Place holder for PSC module's source code.

6.2.2 Build Options

PSC module will be built with "whole_program_debug" mode. When built successfully the respective library will be available at <ID>\ti\psp\psc\lib\<ti.psp.psc.a674>.

6.3 Features

This section details the features of PSC and how to use them in detail.

6.3.1 Multi-Instance

The PSC module can operate on all the instance of PSC on DA830. Different instances are specified during module configuration time. Supported instance are 0 and 1 only with instNum (or device ID) 0 and 1 only.

These instances could be operated simultaneously with configurations supported by PSC module.

The device ID could be specified using the `instNum` field of structure Psc_Params. There are two ways in which a new instance of the PSC module can be created.

1. Static creation – static creation of the module is done in the "cfg" file of the application. The creation happens at compile time.
2. Dynamic creation – Dynamic creation of module is done in the application source files and the creation happens at runtime.

6.3.2 Operating on each Instance

PSC can be operated upon as a standalone module. This could be achieved by calling the PSC module APIs directly by the application. The instance creation needs configuration parameters. The configuration parameters are explained in the sections to follow.

6.4 Configurations

Following tables document some of the configurable parameter of PSC. Please refer to Psc.h (Psc.xdc) for complete configurations and explanations.

6.4.1 Module configuration

The following parameters are module wide configurable parameters. Applications can configure these settings according to their specific requirements.

variable	Description
paramCheckEnable	Option to enable/disable the input parameter checking at the function entry point.

6.4.2 Psc_Params

This structure is used for the device configuration of the PSC module. The PSC module expects the application to supply this configuration during the creation of the module instance.

Members	Description
instNum	Hardware instance of the PSC to be created.

6.5 Use of PSC through module APIs

Following sections explain the use of parameters of module calls in the context of PSC module. Any PSC specific requirements are covered below.

6.5.1 Psc_create

This call creates the PSC module instance and returns the handle to the module instance. The PSC module is a support module which provides APIs for clock control of the peripherals. The sample applications (PSC does not have a separate sample application. Users could refer to GPIO sample application for the same) provided shows the creation of an instance statically in CFG file. This instance handle (Program.global.psc0) is accessed in C file by referencing this handle as an external variable (extern Psc_Handle psc0;). This handle should be used to further reference this instance for any PSC module API calls.

Parameter Number	Parameter	Specifics to PSC
1	Psc_Params *	Pointer to the Psc_Params structure required for the instance creation

6.6 Sources that need re-targeting

6.6.1 SoC level changes

When the module has to adapt to SoC level changes the two files Psc.xs (Module Script File) and the SoC script file soc.xs need to be updated with the changes.

6.6.2 EVM level changes

None

6.7 EDMA3 Dependency

The PSC module does not depend on the EDMA3 LLD driver. It does not support any data transfer operations.

6.8 Known Issues

Please refer to the top level release notes that came with this release.

6.9 Limitations

Please refer to the top level release notes that came with this release.

7 Mcasp driver

7.1 Introduction

This document is the reference guide for using the Mcasp device. The user guide explains in detail the features of the device driver and the various options available for the user to configure and use the driver.

DSP/BIOS applications use the driver typically through APIs provided by Stream layer, to transmit and receive serial data. The following sections describe in detail, procedures to use this driver, and configure among others…. It is recommended to go through the sample application to get a feel of initializing and using the Mcasp driver.

7.1.1 Key Features

- Multi-instanceable and re-entrant driver.[7.3.1]
- Each instance can operate as a receiver and/or transmitter.[7.3.2]
- Supports multiple data formats.[7.3.3]
- Can be configured to operate in multi-slot TDM, I2S, DSP and DIT (S/PDIF).[7.3.4]
- Mechanism to transmit desired data (such as NULL tone), when idle.[7.3.5]
- Explicit control of PIN directions for High Clock, Bit Clock and Frame Sync PINS.[7.3.6]

7.1.2 Terms and Abbreviations

API	Application Programmer's Interface
CSL	TI Chip Support Library – primitive h/w abstraction.
IP	Intellectual property
ISR	Interrupt Service Routine
OS	Operating System
S/PDIF	Sony Philips Digital Interface
TDM	Time Division Multiplexing
I2S	Inter-Integrated Sound Format
ID	Installation Directory

7.1.3 References

1 SPRUFM1 DA830 McASP Reference Guide

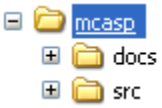
2 TLV320AIC31IRHBRG4_3960631 Stereo Audio Codec Data Manual

7.2 Installation

The Mcasp device driver is a part of PSP package for DA830 platform and would be installed as part of whole package installation. For high level design information please refer to the driver architecture guide that came with this package (available at <ID>\ti\psp\mcasp\docs)

7.2.1 McASP Component folder

On installation of PSP package for DA830, the McASP driver can be found at <ID>\ ti\psp\mcasp\



As show above the McASP folder contains sub-folder, contents of which are described below.

- **Mcasp -** The Mcasp folder is the place holder for the entire mcasp driver, documents and the build configuration files. McASP driver is implemented as IDriver under DSP/BIOS™ operating system. Stream defined APIs could be used to interface to McASP driver. This folder contains the build configuration file(package.bld),the McASP module specification file (Mcasp.xdc),the runtime configuration file (Mcasp.xs) and the miscellaneous files required for compiling the Mcasp library.
- **docs –** Holds Mcasp driver's architecture. Please note that the API reference would be found as a part of cdoc help (<install dir>\packages\docs\cdoc\index.html)
- **src –** Place holder for McASP driver's source code.

7.2.2 Build Options

McASP device driver will be built with "whole_program_debug" mode. When built successfully the respective library will be available at <ID>\ti\psp\mcasp\lib\ < ti.psp.mcasp.a674>

7.3 Features

This section details the features of McASP and how to use them in detail.

7.3.1 Multi-Instance

The McASP driver is multi-instantiable i.e. it can simultaneously support different instances of the driver. It can operate on all the instance of McASP available on DA830. Different instances are specified during driver creation time. Supported instance are 0 through 2 with device ID 0 through 2 respectively.

These instances could be operated simultaneously with configurations supported by McASP driver.

There are two ways in which a new instance of the Mcasp driver can be created.

1. Static creation – static creation of the driver is done in the "cfg" file of the application. The creation happens at compile time.
2. Dynamic creation – Dynamic creation of driver is done in the application source files and the creation of the instance happens at runtime.

The device instance to be used could be specified using the "`instNum`" field of structure Mcasp_Params.

(i.e. Mcasp_Params.`instNum = 0x00`, Mcasp_Params.`instNum = 0x1`, so on…).

Each instance of the driver is independent and can be configured as the user requires.each instance will maintain its own instance state variables which will be used to store the information specific to the particular instance.

7.3.2 Each Instance as Transmitter and / or receiver

Each instance of the driver can be used for either transmit or receive or simultaneous transmit and receive operation. This could be achieved by opening a stream Channel as an INPUT channel and opening a stream Channel as an OUTPUT channel. The type of Channel is specified while creating the channel. (using `Stream_create()` specify "`DriverTypes_OUTPUT`" or "`DriverTypes_INPUT`").

Also the creation of the channel takes multiple parameters (specified by the structure "`Mcasp_ChanParams`"). Refer to section 7.6.6 for the details about the channel parameters and their explanation.

7.3.3 Supported Data Formats

McASP driver expects the data (samples) to be arranged in a specific format when requesting for an IO transfer. These formats are explained under scenario of using 1 serializer or more serializers and also 1 slot or more slots. The required buffer format could be configured at the channel creation time. The sections below capture the details of supported data formats.

McASP Mode	**Single Serializer**	**Multiple Serializer**
Burst Mode / DSP Mode	1SER_1SLOT	MULTISER_1SLOT_SER_INTERLEAVED, MULTISER_1SLOT_SER_NON_INTERLEAVED
Multi-Slots TDM/I2S	1SER_MULTISLOT_NON_INTERLEAVED, 1SER_MULTISLOT_INTERLEAVED,	MULTISER_MULTISLOT_SEMI_INTERLEAVED_1, MULTISER_MULTISLOT_SEMI_INTERLEAVED_2, MULTISER_MULTISLOT_SEMI_INTERLEAVED_3
DIT	1SER_MULTISLOT_NON_INTERLEAVED, 1SER_MULTISLOT_INTERLEAVED,	MULTISER_MULTISLOT_SEMI_INTERLEAVED_1, MULTISER_MULTISLOT_SEMI_INTERLEAVED_2, MULTISER_MULTISLOT_SEMI_INTERLEAVED_3

7.3.3.1 1SER_1SLOT (Burst Mode)

When configured in this mode, it is expected that McASP is configured to use 1 serializer and uses only one slot. The expected data format is as depicted below.

[<Slot0-Sample1>, <Slot0-Sample2>, <Slot0-Sample3>……..<Slot0-SampleN>**]**

Note: Interleaved and Non – interleaved data format is not applicable for this specific format.

The Mcasp expects to transfer a <wordwidth> of data for every event(RX/TX). The total size that would be required to specify during the IO request is = <word width>*<number of samples **N**>.

The sample application that came with this package demonstrates the use of this data format. File audioSample_io.c implements the functions which configure McASP to use this buffer format. This buffer format is used to support DSP/BURST mode where only one serializer is being used.

The key configurations are

- `Mcasp_ChanParams.noOfChannels = 0x01;`
- `Mcasp_ChanParams.noOfSerRequested = 0x01;`
- `Mcasp_ChanParams.indexOfSersRequested[0] = Mcasp_SerializerNum_N;`
- The size of the IO request is computed as <`No of Bytes per Sample`> * < `No of Samples` >. This value should be given as a size parameter of `stream_submit` ().

- Idle Time[7.3.5] data pattern length computation. Minimum length should be **<word width in bytes>** or an integral multiple of computed value. While allocating buffer, allocate <**word width in bytes**>.

7.3.3.2 *MULTISER_1SLOT_SER_INTERLEAVED(BURST MODE)*

When configured in this mode, it is expected that McASP is configured to use multiple serializers and each serializer is using only one slot. The expected data format is as depicted below. Below example uses N number of serializers with each having one slot enabled.

[<Slot 0 serializer 0 sample 1> <Slot 0 serializer 1 sample 1>…<Slot 0 serializer N sample 1>

<Slot 0 serializer 0 sample 2> <Slot 0 serializer 1 sample 2>…<Slot 0 serializer N sample 2>

….

<Slot 0 serializer 0 sample N> <Slot 0 serializer 1 sample N>…<Slot 0 serializer N sample N>]

The key configurations are

- `Mcasp_ChanParams.noOfChannels = 0x01`
- `Mcasp_ChanParams.noOfSerRequested = 0x0N`
- `Mcasp_ChanParams.indexOfSersRequested[0] = SERIALIZER_0,…`
- The size of the IO request is computed as <`No of Bytes per Sample`> * <`No of serializers`> * <`No of samples`>. This value should be given as a size parameter of `stream_submit` ()
- Idle Time[7.3.5] data pattern length computation. Minimum length should be **<word width in bytes> * <no of serializers>** or an integral multiple of computed value. While allocating buffer, allocate <`computed value`>.

7.3.3.3 *MULTISER_1SLOT_SER_NON_INTERLEAVED*

When configured in this mode, it is expected that McASP is configured to use multiple serializers and each serializer is using only one slot. The serializer data is not interleaved. The expected data format is as depicted below. It is assumed that the configuration has 1 slot and N serializers.

[<Slot 0 serializer 0 sample 1> <Slot 0 serializer 0 sample 2>…<Slot 0 serializer 0 sample N>

<Slot 0 serializer 1 sample 1> <Slot 0 serializer 1 sample 2>…<Slot 0 serializer 1 sample N>

….

<Slot 0 serializer N sample 1> <Slot 0 serializer N sample 2>…<Slot 0 serializer N sample N>]

The key configurations are

- `Mcasp_ChanParams.noOfChannels = 0x01`
- `Mcasp_ChanParams.noOfSerRequested = 0x0N`
- `Mcasp_ChanParams.indexOfSersRequested[0] = SERIALIZER_0,…`
- The size of the IO request is computed as <`No of Bytes per Sample`> * <`No of Samples`> * <`No of serializers`>. This value should be given as a size parameter of `stream_submit` ()
- Idle Time[7.3.5] data pattern length computation. Minimum length should be **<word width in bytes> * <no of serializers>** or an integral multiple of computed value. While allocating buffer, allocate <`computed value`>.

7.3.3.4 1SER_MULTISLOT_NON_INTERLEAVED

When configured in this mode, it is expected that McASP is configured to use single serializer and each serializer is using multiple slots. The expected data format is as depicted below.

[<Slot 0 serializer 0 sample 1> <Slot 0 serializer 0 sample 2>…<Slot 0 serializer 0 sample N>

<Slot 1 serializer 0 sample 1> <Slot 1 serializer 1 sample 2>…<Slot 1 serializer N sample N>

….

<Slot N serializer 0 sample N> <Slot N serializer 1 sample N>…<Slot N serializer N sample N>]

The key configurations are

- `Mcasp_ChanParams.noOfChannels = 0x0N`
- `Mcasp_ChanParams.noOfSerRequested = 0x01`
- `Mcasp_ChanParams.indexOfSersRequested[0] = SERIALIZER_0`
- The size of the IO request is computed as <`No of Bytes per Sample`> * < `No of Samples`> * < `No of slots` >. This value should be given as a size parameter of `stream_submit` ()
- Idle Time[7.3.5] data pattern length computation. Minimum length should be **<word width in bytes> * <no of slots>** or an integral multiple of computed value. While allocating buffer, allocate `<computed value>`.

7.3.3.5 1SER_MULTISLOT_INTERLEAVED

When configured in this mode, it is expected that McASP is configured to use single serializer and each serializer is using multiple slots. The expected data format is as depicted below. The Slot data is interleaved as shown below. The below example assumes N slots.

[<Slot 0 serializer 0 sample 1> <Slot 1 serializer 0 sample 1>…<Slot N serializer 0 sample 1>

<Slot 0 serializer 0 sample 2> <Slot 1 serializer 0 sample 2>…<Slot N serializer 0 sample 2>

….

<Slot 0 serializer 0 sample N> <Slot 1 serializer 0 sample N>…<Slot N serializer 0 sample N>]

The key configurations are

- `Mcasp_ChanParams.noOfChannels = 0x0N`
- `Mcasp_ChanParams.noOfSerRequested = 0x01`
- `Mcasp_ChanParams.indexOfSersRequested[0] = SERIALIZER_0`
- The size of the IO request is computed as <`No of Bytes per Sample`> * < `No of Samples`> * < `No of slots` >. This value should be given as a size parameter of `stream_submit` ()
- Idle Time[7.3.5] data pattern length computation. Minimum length should be **<word width in bytes> * <no of slots>** or an integral multiple of computed value. While allocating buffer, allocate `<computed value>`.

7.3.3.6 MULTISER_MULTISLOT_SEMI_INTERLEAVED

This format supports 2 different data arrangments

7.3.3.6.1 MULTISER_MULTISLOT_SEMI_INTERLEAVED_1

When configured to use this format, the data for a given serializers is interleaved whereas the data for the slots are continuous. The following representation specifies the expected data format. The assumption in this example is we have enabled 2 serializer and 3 slots in each serializer.

[< slot 0 serializer 0 sample 1> <slot 0 serializer 1 sample 1> … <slot 2 serializer 0 sample 1> <slot 2 serializer 1 sample 1>
< Slot 0 serializer 0 sample 2> <slot 0 serializer 1 sample 2> … <slot 2 serializer 0 sample 2> <slot 2 serializer 1 sample 2>
…
< slot 0 serializer 0 sample N-1> <slot 0 serializer 1 sample N-1> … <slot 2 serializer 0 sample N-1> <slot 2 serializer 1 sample N-1>
< slot 0 serializer 0 sample N> <slot 0 serializer 1 sample N> … <slot 2 serializer 0 sample N> <slot 2 serializer 1 sample N>]

The key configurations are

- `Mcasp_ChanParams.noOfChannels = 0x03`
- `Mcasp_ChanParams.noOfSerRequested = 0x02`
- The size of the IO request is computed as <`No of Bytes per Sample`> * < `No of Samples` **per Slot**> * <`no of slots`> * < `no of serializers`>. This value should be given as a size parameter of `Stream_submit` ().
- Idle Time[7.3.5] data pattern length computation. Minimum length should be **<number of slots enabled> * <word width in bytes> * <no of serializers>** or an integral multiple of computed value. While allocating memory for the loopJobBuffer allocate the `<computed size>`.

7.3.3.6.2 MULTISER_MULTISLOT_SEMI_INTERLEAVED_2

When configured to use this format, the data for a given serializers is continous whereas the data for the slots are interleaved. The following representation specifies the expected data format. The assumption in this example is we have enabled N serializer and two slots in each serializer.

[< slot 0 serializer 0 sample 1> <slot 1 serializer 0 sample 1> … <slot 0 serializer 0 sample N> <slot 1 serializer 0 sample N>
< Slot 0 serializer 1 sample 1> <slot 1 serializer 1 sample 1> … <slot 0 serializer 1 sample N> <slot 1 serializer 1 sample N>
…
< Slot 0 serializer N-1 sample 1> <slot 1 serializer N-1 sample 1> … <slot 0 serializer N-1 sample N> <slot 1 serializer N-1 sample N>
< Slot 0 serializer N sample 1> <slot 1 serializer N sample 1> … <slot 0 serializer N sample N> <slot 1 serializer N sample N>]

The key configurations are

- `Mcasp_ChanParams.noOfChannels = 0x02`
- `Mcasp_ChanParams.noOfSerRequested = 0x0N`
- The size of the IO request is computed as <`No of Bytes per Sample`> * < `No of Slots per serializer`> * <`No of slots`> * < `no of samples`>. This value should be given as a size parameter of `Stream_submit` ().
- Idle Time[7.3.5] data pattern length computation. Minimum length should be **<number of slots enabled> * <word width in bytes> * <no of serializers>** or an integral multiple of computed value. While allocating memory for the loopJobBuffer allocate the <`computed size`>.

7.3.4 Operational Modes (multi-slot TDM, I2S, DSP and DIT (S/PDIF)

7.3.4.1 Multi-Slot TDM

To configure McASP to operate with multi-slot, use the `Mcasp_HwSetupData.tx/rx.frSyncCtl`, this variable represents McASPs AFRCTL/AFXCTL. Refer section 7.3.3 for details on the supported data formats. We can program a 2 lot TDM to 32 slot TDM in these registers.

7.3.4.2 I2S

To configure McASP to operate in I2S format, use the `Mcasp_HwSetupData.tx/rx.frSyncCtl` and `Mcasp_HwSetupData.tx/rx.xfmt`. This variable represents McASPs AFRCTL/AFXCTL and XFMT / RFMT registers.Program the register to support 2 slots.

7.3.4.3 DSP

To configure McASP to operate in DSP format, use the `Mcasp_HwSetupData.tx/rx.frSyncCtl` the fields `RMOD/XMOD` should be 0 and `FRWID / FXWID` should be 0. This variable represents McASPs AFRCTL/AFXCTL.

The initialization time configurable parameter **noOfChannels** could be used to specify the no of channels that 32 bit is split into. E.g if 32 bit is to be interpreted as 2 16 bit samples, the noOfChannels should be set to 2.Note that the maximum transfer size supported by the driver is only 32 bit.

7.3.4.4 DIT (S/PDIF)

To change the User Bits and Channel Status Bits that would be embedded by the S/PDIF stream, applications are expected to give the following parameters

- `Mcasp_PktAddrPayload.writeDitParams = TRUE;`
- `Mcasp_PktAddrPayload.chStat = Address of structure of type Mcasp_ChStatusRam.`
- `Mcasp_PktAddrPayload.userData = Address of structure of type Mcasp_UserDataRam.`

Driver would update the User Bits and Channel Status bits immediately. Applications using the driver are in complete control change/update of User Bits and Channel Status bits.

7.3.5 IDLE Time Data Patterns

IDLE Time in the context of McASP could be better explained under the CREATE Time and Run Time. The sections below explain the behavior of Clock, Frame Sync and Data signals.

7.3.5.1 Create Time

On successful creations of Stream channel, the McASP driver starts generating the clock, Frame Sync and data (if configured as source / if configured as sink McASP expects theses signals). The data that would be sent out at this point can be configured using `Mcasp_ChanParams.userLoopJobBuffer` and `Mcasp_ChanParams .userLoopJobLength`. Optionally this could be set NULL and 0x0 respectively, the McASP driver then uses driver's internal buffers and length of these NULL buffers is <wordwidth * no of slots * no of serializers> bytes.

7.3.5.2 Run Time

If the applications could not meet the real time needs of transmission/reception of data, McASP steps in to consume to received the data or transmit a know data pattern.

McASP driver could be configured to send out a know pattern when ever the above situation arises using `Mcasp_ChanParams.userLoopJobBuffer` and `McaspChanParams .userLoopJobLength`. Optionally this could be set NULL and 0x0 respectively, the McASP driver uses driver's internal buffers and length of these NULL buffers is <wordwidth * no of slots * no of serializers> bytes.

7.3.5.3 IDLE Time buffer size

This IDLE Time data patterns could possibly have un-intended effects, if used in-correctly. It is recommended that following method is used to calculate the size of the IDLE time buffers.

Size of Idle Time buffers = <width of slot in bytes> * <no of serializer enabled> * <no of slots enabled>

If the application does not supply the idle time buffers, the McASP driver would use its internal buffer of length <wordwidth * no of slots * no of serializers> bytes when operating in TDM mode and 8 bytes when operating in DIT mode.

CAUTION: If the computed size does not match the logical end of slots, the channels could be swapped. A quick way to check would be to monitor the frame sync and data line/s on scope and send out unique pattern in each slot of the idle time buffer.

7.3.6 Explicit control of IO PINS

McASP drivers provide explicit control on the directions of the following McASP pins.

Signal Pin	Description
`AFSR`	Frame Sync signal for reception. Direction should be explicitly set when channel opened for READ
`AHCLKR`	High Clock signal for reception. Direction should be explicitly set when channel opened for READ
`ACLKR`	Bit Clock signal for reception. Direction should be explicitly set when channel opened for READ
`AFSX`	Frame Sync signal for reception. Direction should be explicitly set when channel opened for WRITE
`AHCLKX`	High Clock signal for reception. Direction should be explicitly set when channel opened for WRITE
`ACLKX`	Bit Clock signal for reception. Direction should be explicitly set when channel opened for WRITE

There could be scenarios where the applications would require the McASP to be configured as MASTER (one generating the Frame Sync, Bit Clock and High Clock) and yet not drive these pins. This feature allows achieving this.

Use `Mcasp_HwSetup.glb.pdir` to set the directions. This variable maps to PDIR register of McASP

7.4 Clocking McASP

The McASP peripheral requires two clocks to operate. The peripheral clock used to drive the peripherals functional, the second clock (also called as auxiliary clock / internal clock source) used to generate the high clock and the bit clocks for the serial data-bit streams.

Alternatively, McASP could be configured to use an external clock source to derive the bit clock for the serial data-bit streams. This external clock would be received via the High Clock Pin. This setup is referred to as External Clock in this document.

7.4.1 Internal Clock

The Auxiliary clock passes thorough a two stage divider to generate bit clock for the serial data stream. Please refer the data manual for McASP, section 2.2.1 Transmit Clock and 2.2.2 Receive Clock. The configurations that would be required are explained in the context of the example below.

Assumption: McASP is configured as output channel and would require to output the High Clock (used as the system clock for the DACs), Bit clock and the frame sync. For these setup following are the key configurations

- **`Mcasp_HwSetup.glb.pdir = 0x1C000000;`** With this we are selecting `AFSX, AHCLKX, CLKX` as out pins and `AFSR, AHCLKR, CLKR as input pins.`
- **`Mcasp_HwSetupData.clk.clkSetupHiClk = 0x000080XX;`** With this we are configuring McASP high clock to be sourced from internal clock (auxiliary clock divided by the divisor specified by bits 0-11 of this register, is interpreted as High Clock)
- **`Mcasp_HwSetupData.clk.clkSetupClk = 0x0000002X;`** With this we are configuring McASP to source bit clock from the output of High clock (High Clock divided by the divisor specified by divisor specified by the bits 0-4 of this value)
- If it's desired that the High Clock, Frame Sync and Bit Clock signal should not be outputted, change the pin functionality as an input pin.

7.4.2 External Clock

7.4.2.1 External Frame Sync & External Bit Clock

McASP could be programmed to source the Frame Sync (for both reception and transmission) from an external source such as DAC/ADC. The condition being that the Bit Clock is also sourced from the same entity, failing which the behavior is un-predictable (i.e. we could see clock failure condition). To configure the McASP to source Bit clock and Frame Sync from an external entity following are the important configurations.

Assuming that McASP is configured to transmit data and High Clock is ignored.(i.e. External entity is generating Frame Sync and Bit clocks only)

- **`Mcasp_HwSetup.glb.pdir = 0x00000000;`** With this we are selecting `AFSX, AHCLKX, CLKX` as input pins and `AFSR, AHCLKR, CLKR` could be ignored if the receive section of McASP is un-used.
- **`Mcasp_HwSetupData.clk.clkSetupHiClk = 0x00000000;`** With this we are configuring McASP Bit clock to be sourced from ACLKX Pin. (Typically, in this scenario we would not want to divide bit clock, we could out of Sync and not meet the needs of the external device)
- **`Mcasp_HwSetupData.clk.clkSetupClk = 0xXXXXXXXX;`** Since we are sourcing the Bit clock from the external AHCLK Pin, this register will not have any effect on the Bit Clock and Frame Sync.

7.4.2.2 External High Clock

McASP could be programmed to source the High Clock from an external entity. Typically if the High Clock is sourced from an external entity, the Bit Clock and

Frame Sync would be generated the McASP. The Bit Clock and the Frame Sync in turn could feed into a serials data consumption unit such as a DAC. The configurations mentioned below are the important configurations that are to configured to use the external High Clock

Assuming that McASP is configured to transmit data and High Clock is sourced from an external entity.

- **`Mcasp_HwSetup.glb.pdir = 0x14000000;`** With this we are selecting `AHCLKX` as input pins, AFSX / ACLKX as output `pins` and `AFSR`, `AHCLKR`, `CLKR` could be ignored if the receive section of McASP is un-used.
- **`Mcasp_HwSetupData.clk.clkSetupHiClk = 0x000000XX;`** With this we are configuring McASP high clock to be sourced from AHCLKX Pin (The output of clock divided by the divisor specified by bits 0-11 of this register, is interpreted as High Clock)
- **`Mcasp_HwSetupData.clk.clkSetupClk = 0x0000002X;`** With this we are configuring McASP to source bit clock from the output of High clock (High Clock divided by the divisor specified by divisor specified by the bits 0-4 of this value)

7.5 Clock Configuration (EVM - DA830)

McASP drivers sample application that came with this release is configured to use external Clock. The configurations are as explained in section 7.4.1. The sample application demonstrates the audio data capturing through the line in and transmits the same data through the line out Pin.

7.6 Configurations

Following tables document some of the configurable parameter of McASP. Please refer to Mcasp.h (Mcasp.xdc) for complete configurations and explanations.

7.6.1 Module configuration

The following parameters are module wide configurable parameters

Variable	Description
`paramCheckEnable`	Option to select if parameter checking at the function entry should be enabled or disabled.
`SWI_PRIORITY`	Option to modify the priority of the SWI thread used for the handling of the interrupts

7.6.2 Mcasp_Params

This structure defines the device configurations, expected to supply while creating the driver. This is provided when driver channels are created (e.g. stream_create).

Members	Description
`instNum`	Instance number of the driver.
`hwiNumber`	Maps HWI event number to the ECM group. Please

	note that no validation is done by the driver.
enablecache	Specifies if the applications supplied buffers required to be FLUSHED/INVALIDATED.
isDataBufferPayloadStructure	Specifies to use to use User Bits, Channel Status bit and flag update DIT params of the IO request.
mcaspHwSetup	Hardware configurations of McASP driver.

7.6.3 Mcasp_HwSetup

Members	Description
Glb	Specifies the device configurations that are common for both the reception and transmission section.
Rx	Specifies the configurations that are specific to the reception section.
Tx	Specifies the configurations that are specific to the transmission section.
Emu	Power down emulation mode control

7.6.4 Mcasp_HwSetupGbl

Members	Description
pfunc	Kept for future use. Driver decides the functionality of the McASP PINS.
pdir	Applications could decide the PIN directions of Frame Sync, High Clock and Bit Clock for both reception and transmission. The directions are determined the driver.
Ctl	Kept for future use. Recommended to be 0x0 for now.
Ditctl	Kept for future use. Recommended to be 0x0 for now.

7.6.4.1 Mcasp_HwSetupData

This structure defines the channel specific configurations for reception section and transmission section.

Members	Description
Mask	The driver applies the value supplied by this register to RMASK/XMASK
Fmt	The driver applies the value supplied by this register to RFMT/XFMT
frSyncCtl	The driver applies the value supplied by this register to AFSRCTL/AFSXCTL
Tdm	The driver applies the value supplied by this register to RTDM/XTDM

intCtl	The driver applies the value supplied by this register to RINTCTL /XINTCTL
Stat	The driver applies the value supplied by this register to RSTAT/XSTAT
evtCtl	The driver applies the value supplied by this register to REVTCTL/XEVTCTL
Clk	Configure the BIT clock, the High clock configuration and Clock failure detection

7.6.5 Mcasp_HwSetupData

Members	Description
clkSetupClk	The driver applies the value supplied by this register to ACLKRCTL/ACLKXCTL
clkSetupHiClk	The driver applies the value supplied by this register to AHCLKRCTL/AHCLKXCTL
clkChk	The driver applies the value supplied by this register to RCLKCHK/XCLKCHK

7.6.6 Mcasp_ChanParams

Applications could use this structure to configure the channel specific configurations.

Members	Description
noOfSerRequested	The number of serializers required to used by the channels.
indexOfSersRequested	Index of the serializer that would be required.
mcaspSetup	The hardware configurations required for the channel specifically. Please refer section PSP_McaspHwSetupData.
channelMode	To operate in DIT/TDM mode
wordWidth	Required wordwidth in the slots.
isDmaDriven	whether the channel is DMA driven.
userLoopJobBuffer	Buffer to be transferred when the loop job is running.
userLoopJobLength	Number of bytes of the userloopjob buffer for each serializer.
edmaHandle	Handle to PSP EDMA LLD driver
gblCbk	callback required when global error occurs and this must be callable from the ISR context
noOfChannels	No of channels of data to be transmitted. Please refer section 7.3.4.3 for details.
dataFormat	The buffer format is specified by the application
enableHwFifo	This parameter is used by the application to specify if the McASP Hw FIFO is to be used or not.
isDataPacked	This variable is used to specify if the data supplied by the buffer is to be packed to the nearest slot width or is it to be

	rounded to the nearest 32,16 bit width.

7.6.7 Mcasp_PktAddrPayload

Application are expected to pass pointer to this structure in `Stream_submit ()` function calls. It is recommends that these packets are allocated on the heap, since the driver would return a pointer to this structure when the IO request is completed/flushed/aborted.

Members	Description
`chStat`	Applicable to DIT mode, should point to a channel status bits associated with S/PDIF stream.
`userData`	Applicable to DIT mode, should point to a user bits associated with S/PDIF stream.
`writeDitParams`	Flag to indicate if the user bits and channel status bits is to be updated/re-configured with the supplied values.
`Addr`	Pointer to data that requires to be transmitted. Please refer section 7.3.3 for details on the supported data formats.

7.7 IO Request Format

While creating the McASP device driver (either through CFG file statically or using the API Mcasp_create) it's required to configure as to how the data buffers would be supplied by the application.

7.7.1 TDM Mode

Application could pass the address of the audio buffer to McASP via the stream_write () API. On completion of transmission/reception the application supplied callback would be called with address of the audio buffer as the parameter. The behavior described above could be configured using the create time configuration

`Mcasp_params.isDataBufferPayloadStructure = FALSE`

If `Mcasp_Params.isDataBufferPayloadStructure` is set to `TRUE` the audio data is expected to be encapsulated in structure `PSP_Mcasp_PktAddrPayload`. The member `writeDitParams` should be set to `FALSE`.

7.7.2 DIT Mode

Applications could use the structure `Mcasp_PktAddrPayload` to pass a pointer to the data buffer and specify User Bits / Channel Status Bits. In DIT mode, this could be specified with configuration M**`casp_Params.isDataBufferPayloadStructure = TRUE`**, the driver would interpret the data buffer passed in function call `Stream_submit ()` as a pointer to structure M`casp_PktAddrPayload` and all its members are populated.

7.8 CACHE Control

MCASP could be configured to FLUSH/INVALIADTE the application supplied buffers while creating the drivers () with configuration parameter M**`casp_Params.enablecache = TRUE/FALSE`**. When set to TRUE for every request the data buffer is FLUSHED/INVALIDATED. One could improve the latency of Stream_submit () call by providing pre-flushed/pre-invalidate data and disabling the cache option.

7.9 Control Commands

Following table describes some of important the control commands, for a comprehensive list please refer the IOCTL defined in `Mcasp.h(Mcasp.xdc)`.

Command	Arguments	Description
Mcasp_IOCTL_CNTRL_AMUTE	Uint32 *	Writes the supplied Uint32 value into AMUTE register of McASP peripheral.
Mcasp_IOCTL_STOP_PORT	None	Stops the transmission/reception. The current IO request in the QUE is completed.
Mcasp_ICOTL_START_PORT	None	Re-Starts the transmission / reception. When there are no pending IO requests, the clocks are stopped and re-started.
Mcasp_IOCTL_CTRL_MODIFY_LOOPJOB	Mcasp_ChanParams *	Used to modify the existing know data pattern. Parameters userLoopJobBuffer and userLoopJobLength are used.
Mcasp_IOCTL_CTRL_MUTE_ON	None	Applicable to Transmit channel only. The current IO request is completed and MUTE Data pattern is sent out
Mcasp_ IOCTL_CTRL_MUTE_OFF	None	Applicable to Transmit channel only which is muted. Configures to play the next pending IO request, else configures to play the LoopJobBuffers.
Mcasp_IOCTL_PAUSE	None	Pause the Mcasp channel operations
Mcasp_IOCTL_RESUME	None	Resume the Mcasp channel operations
Mcasp_IOCTL_CHAN_RESET	None	De-activates the transmission/reception and returns all the queued request with status of the IO request set as FLUSHED/ABORTED
Mcasp_IOCTL_CNTRL_SET_FORMAT_CHAN	Mcasp_HwSetupData *	Re-Configures the channel with new configurations specified. Takes no effect on the pending / current IO request.
PSP_MCASP_CNTRL_GET_FORMAT_CHAN	Mcasp_HwSetupData *	Return the current channel configurations
Mcasp_IOCTL_DEVICE_RESET	None	Icotl command to reset the Mcasp device
Mcasp_ IOCTL_QUERY_MUTE	Uint32 *	Ioctl command to query the current settings of the AMUTE register.

Mcasp_ IOCTL_SET_DIT_MODE	Uint32 *	Icotl command to set the DIT mode of operation
Mcasp_IOCTL_CHAN_TIMEDOUT	None	Ioctl command to handle the channel timeout condition.
Mcasp_IOCTL_ABORT	None	This IOCTL aborts all the pending request of the channel and stops the state machine. The EDMA transfer is also stopped.
Mcasp_IOCTL_SET_DLB_MODE	None	This command is used to set the McASP in to the loopback mode.
Mcasp_IOCTL_CNTRL_SET_GBL_REGS	Mcasp_HwSetup *	Command to set the global control registers

7.10 Use of McASP driver through Stream APIs

Following sections explain the use of parameters of Stream calls in the context of McASP driver. Note that no effort is made to document the use of Stream calls; any McASP specific requirements are covered below.

7.10.1 Stream_create

Parameter Number	Parameter	Specifics to McASP
1	Device Name string	Unique identifier used to identify this driver. Please note the name should be same as specified while creating the driver. (Either through CFG or Mcasp_create ()
2	IO Type	Should be "DriverTypes_INPUT" when McASP requires to received data and "DriverTypes_OUTPUT" when McASP requires to transmit
3	Stream_Params *	Parameters required for the creation of the stream (e.g. channel parameters)
4	Error_Block *	Pointer to the application supplied error block

7.10.2 Stream_control

Parameter Number	Parameter	Specifics to McASP
1	Stream_handle	Handle returned by stream_create
2	Command	IOCTL command defined by Mcasp driver

3	Arguments	Misc arguments if required by the command
4	Error_Block *	Pointer to the Application supplied error block

7.10.3 Stream_write/Read

Parameter Number	Parameter	Specifics to McASP
1	channel Handle	Handle returned by `Stream_create`
3	Pointer to buffer	Should be pointer to variable of type `Mcasp_PktAddrPayload OR Uint32 *` that holds the audio data.
4	Size	Size of the transaction
5	Error_Block *	Pointer to the Application supplied error block

7.10.4 Mcasp_create

Parameter Number	Parameter	Specifics to McASP
1	Mcasp_params *	Pointer to the Mcasp_params structure required for the Driver creation

7.11 Timeline of Frame Sync, High Clock and or Bit Clock generation

The behavior of McASP drivers is better explained under these two sections.

7.11.1 McASP sourcing Frame Sync, High clock and or Bit Clock

On successful creation of McASP device driver, the Frame Sync, Bit Clock and High Clock are started. In EVM designs such as DA830, the High Clock is fed into On board DAC/ADC (Such as AIC31). Applications are expected to create the driver first, (after recommended delay) applications could program the DACs.

7.11.2 McASP sinking Frame Sync, High clock and or Bit Clock

When McASP is sinking the Frame Sync, Bit Clock and or High Clock, applications should ensure that clocks are being fed into McASP before creating the device driver. Failing which the McASP will not pull transmit/reception section out of re-set. Effectively the driver creation would fail (A timeout error will be indicated by the driver).

7.12 Porting Guide

This section describes the major changes that would be required to port the McASP driver from DS/BIOS™ operating system to a different operating system.

The McASP Device Driver is based upon the DSP BIOS IDriver framework. The driver is tightly coupled with the DSP BIOS operating system

7.13 Sources that need re-targeting

7.13.1 SoC level changes

When the driver has to adapt to SoC level changes the two files Mcasp.xs (Module Script File) and the SoC script file soc.xs need to be updated with the changes.

7.13.2 EVM level changes

None

7.14 EDMA3 Dependency

When the MCASP driver is configured in EDMA mode (compile time configuration is needed) MCASP driver relies on EDMA3 LLD driver to move data from/to application buffers to peripheral; Please note that EDMA3 LLD driver would not be part of this release. Please ensure that current PSP release is compliant with version of EDMA3 driver being used from the system requirements section of this document.

7.14.1 Used Paramset of EDMA 3

McASP driver uses TWO spare paramsets of EDMA3 per channel(A total of 4 for TX and RX combined); if there are no paramsets are available the McASP driver creation would fail. These paramsets are used through the life time of McASP driver.

7.15 How to support "NEW" data format

If a custom data format is to be supported, one would require following these steps.

- Add an enumeration in `Mcasp_BufferFormats` defined in `Mcasp.xdc`
- Update the function `mcaspValidateBufferConfig()` implemented in `Mcasp.c` to recognize this new data format.
- Update the function `mcaspGetIndicesSyncType()`implemented in `mcasp_edma.c` to provide the EDMA 3 indices required to configure EDMA3.

7.16 Known Issues

Please refer to the top level release notes that came with this release.

7.17 Limitations

Please refer to the top level release notes that came with this release.

7.18 Mcasp DIT Sample application

7.18.1.1 Description:

This sample application demonstrates the use of the Mcasp driver in DIT mode. Mcasp driver supports only DMA mode of operation. Also note that the Mcasp driver application also supports only transmission in DIT mode (reception is not supported).

The Mcasp driver along with the required component modules are configured statically in mcaspDitSample.cfg file. The required task for the audio play and the memory for the heap are also created here.

The mcaspDitSample.cfg file contains the remaining BIOS configuration like the configuration of the event combiner etc. This helps to map the Mcasp events to the CPU interrupts.

The main () function configures the PINMUX and uses the Psc module to enable the Mcasp peripheral.

The Audio_echo_Task () task exercises the Mcasp driver. It uses Stream APIS to create McASP driver channel and also to perform the IO operations.

7.18.1.2 Build:

This sample can be built using the CCS4 interface.

IMPORTANT NOTE: The mcaspDitSample project contains the references to %EDMA3LLD_BIOS6_INSTALLDIR% environment variable and links with edma3 libraries. This is required because audio driver by default requires that the EDMA be present.

There is also facility for users to compile the project using the command line. The file package.bld takes care of the necessary steps to compile the project from command line.

Please refer to the "Integration Guide" section for more details about building the project.

7.18.1.3 Setup:

You need an "audio board" to be connected to the evmDA830. The DIT OUT port should be connected to the IN port of the "Flying cow" (a DIT data receiver) device. The OUT port of the "Flying cow" should be connected to the Headphones (speakers).

7.18.1.4 Output:

When the sample application is executed, a sine tone should be heard at the speaker continuously.

8 Audio driver

8.1 Introduction

This document is the reference guide for the Audio driver which explains the features and tips to use the driver in an application.

DSP/BIOS applications use the driver typically through APIs provided by Stream layer, to transmit and receive serial data. The following sections describe in detail, procedures to use this driver, configure among others… It is recommended to go through the sample application to get a feel of initializing and using the Audio driver

8.1.1 Key Features

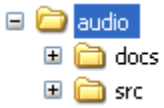
- Multi-instanceable and re-entrant driver
- Each instance can be used to configure a complete receive and transmit section of an audio configuration consisting of an audio device and multiple audio codecs

8.2 Installation

The Audio device driver is a part of PSP package for DA830 platform and would be installed as part of whole package installation. For high level design information please refer to the driver architecture guide that came with this package(available at <ID>\ti\psp\platforms\evmDA830\audio\docs)

8.2.1 Audio Component folder

On installation of PSP package for DA830, the Audio driver can be found at <ID>\ ti\psp\platforms\evmDA830\audio



As show above the audio folder contains sub-folder, contents of which are described below.

- **audio -** The audio folder is the place holder for the entire Audio driver, documents and the build configuration files. Audio driver is implemented as IDriver under DSP/BIOS™ operating system. Stream defined APIs could be used to interface to Audio driver. This folder contains the build configuration file(package.bld),the Audio module specification file (Audio.xdc),the module script file (Audio.xs) and the miscellaneous files required for compiling the Audio library.
- **docs –** Holds Audio driver's architecture. Please note that the API reference would be found as a part of cdoc help (<install dir>\packages\docs\DA830\cdoc\ index.html).
- **src –** Place holder for Audio driver's source code.

8.2.2 Build Options

Audio device driver will be built with "whole_program_debug" mode. When built successfully the respective library will be available at <ID>\ti\psp\platforms\evmDA830\lib\ < ti.psp.platforms.evmDA830.audio.a674>

8.3 Features

This section details the features provided by audio driver and how to use them in detail.

8.3.1 Multi-Instance

The Audio driver can operate on all the instance of Audio configurations available on DA830. Different instances are specified during driver creation time. Supported instance are 0 through 2 with device ID 0 through 2 respectively.

These instances could be operated simultaneously with configurations supported by Audio driver.

The device ID could be specified using the `instNum` field of structure Audio_Params. There are two ways in which a new instance of the Audio driver can be created.

1. Static creation – static creation of the driver is done in the "cfg" file of the application. The creation happens at compile time.
2. Dynamic creation – Dynamic creation of driver is done in the application source files and the creation happens at runtime.

(i.e. Audio_Params.`instNum = 0x0`, Auido_Params.`instNum = 0x1`, so on…)

8.3.2 Each Instance as Transmitter and / or receiver

Each Audio interface driver instance can be used for simultaneous transmit and receive operation. This could be achieved by opening a stream Channel as an INPUT channel and opening a stream Channel as an OUTPUT channel. The type of Channel is specified while creating the channel (using `Stream_create ()`specify `"DriverTypes_OUTPUT" or "DriverTypes_INPUT"`). The configuration parameters are explained in the sections to follow.

8.4 Configurations

Following tables document some of the configurable parameter of Audio. Please refer to Audio.h (Audio.xdc) for complete configurations and explanations.

8.4.1 Module configuration

The following parameters are module wide configurable parameters

Variable	Description
`paramCheckEnable`	Option to select if parameter checking at the function entry should be enabled or disabled.

8.4.2 Audio_Params

This structure defines the device configurations, expected to supply while creating the driver instance. This is provided when driver channels are created (e.g. stream_create).

Members	Description
`instNum`	Instance number of the driver.
`adDevType`	Audio device to be used in the configuration (Mcasp/Mcbsp)

adDevName	Name of the audio device driver in the driver table
acNumCodecs	Number of codecs in the current audio configuration
acDevname	Name of the audio codec device in the driver table

8.4.3 Audio_ChannelConfig

Applications could use this structure to configure the channel specific configurations required by the individual channels.

Members	Description
chanParam	Pointer to the channel structure needed by the audio device. (This structure needs to be identified by the device in use in the current configuration).
acChannelConfig	The structure holding the audio codec driver's channel parameters.

8.5 Use of Audio driver through Stream APIs

Following sections explain the use of parameters of Stream calls in the context of Audio driver. Note that no effort is made to document the use of Stream calls; any Audio specific requirements are covered below.

8.5.1 Stream_create

Parameter Number	Parameter	Specifics to Audio
1	Device Name string	Unique identifier used to identify this driver. Please note the name should be same as specified while creating the driver. (Either through CFG or Audio_create ()
2	IO Type	Should be "DriverTypes_INPUT" when Audio requires to received data and "DriverTypes_OUTPUT" when Audio requires to create a transmit channel.
3	Stream_Params *	Parameters required for the creation of the stream (e.g. channel parameters)
4	Error_Block *	Pointer to the application supplied error block

8.5.2 Stream_control

Parameter Number	Parameter	Specifics to Audio
1	Stream_handle	Handle returned by Stream_create
2	Command	IOCTL command defined by device

		driver to which the command is intented.
3	Audio_ IoctlParam *	Pointer to the structure containing the information about the device to which the command is intended and also the extra information required in case of certain IOCTL commands.
4	Error_Block *	Pointer to the Application supplied error block

8.5.3 Stream_write/read

Parameter Number	Parameter	Specifics to Audio
1	Channel Handle	Handle returned by `Stream_create`
3	Pointer to buffer	Should be pointer to variable of type that holds the data to be transmitted.
4	Size	Size of the transaction
5	Error_Block *	Pointer to the Application supplied error block

8.5.4 Audio_create

Parameter Number	Parameter	Specifics to Audio
1	Audio_params *	Pointer to the Audio_params structure required for the Driver creation

8.6 Control Commands

Following table describes some of important the control commands, for a comprehensive list please refer the IOCTL defined in `Audio.h(Audio.xdc)`.

Command	Arguments	Description
`Audio_IOCTL_SAMPLE_RATE`	`None`	`Configures the sample rate for the entire audio configuration(both the audio device and the audio codec)`

8.7 Sources that need re-targeting

8.7.1 SoC level changes

When the driver has to adapt to SoC level changes the two files Audio.xs (Module Script File) and the SoC script file soc.xs need to be updated with the changes.

8.7.2 EVM level changes

When the platform/EVM changes, the platform.xs file needs to be updated, with information relevant to that platform

8.8 EDMA3 Dependency

The Audio driver does not depend on the EDMA3 LLD driver directly. But, the underlying audio driver might be dependent on the EDMA driver.

8.9 Known Issues

Please refer to the top level release notes that came with this release.

8.10 Limitations

Please refer to the top level release notes that came with this release.

8.11 Audio Sample application

8.11.1.1 Description:

This sample demonstrates the use of the audio driver in DMA mode only.

The Audio driver along with the required component modules are configured statically in audioSample.cfg file. The required task for the audio play and the memory for the heap are also created here.

The audioSample.cfg file contains the remaining BIOS configuration like the configuration of the event combiner etc. This helps to map the Mcasp events to the CPU interrupts.

The main () function configures the PINMUX and uses the Psc module to enable the Mcasp peripheral.

The Audio_echo_task () task exercises the Audio driver. It uses Stream APIS to create audio driver channels and also to perform the IO operations.

8.11.1.2 Build:

This sample can be built using the CCS4 interface.

IMPORTANT NOTE: The audioSample project contains the references to %EDMA3LLD_BIOS6_INSTALLDIR% environment variable and links with edma3 libraries. This is required because audio driver by default requires that the EDMA be present.

There is also facility for users to compile the project using the command line. The file package.bld takes care of the necessary steps to compile the project from command line.

Please refer to the "Integration Guide" section for more details about building the project.

8.11.1.3 Setup:

You need to connect a LINE IN cable to the line-in port available on the EVM. An audio source (like an mp3 player) needs to be connected to the other end of the

LINE IN cable. Connect a speaker to the Line-out port or alternatively you may even connect a headphone to the HPOUT port available on the EVM.

Note: The input audio signal levels to the codec Line-In port should be within the limits specified by the codec data sheet. If the signal is not within the limits specified by the codec datasheet the clipping of the input audio signal will occur and the same will be observed in the form of noise(or disturbance) in the output audio signal.

8.11.1.4 Output:

When the sample is run and the audio is input to the line-in cable, the audio will be in loop back and output through the connected speaker or headphones

9 AIC31 CODEC driver

9.1 Introduction

This document is the reference guide for the Aic31 codec device driver which explains the features and tips to use the driver.

DSP/BIOS applications use the driver typically through APIs provided by Stream layer, to configure the Transmit and receive sections. The following sections describe in detail, procedures to use this driver, configure among others… It is recommended to go through the provided "audio" sample application to get a feel of initializing and using the Aic31 driver.

9.1.1 Key Features

- Multi-instanceable and re-entrant driver
- Independent configuration of transmit and receive sections
- Interfaces to control the codec specific features like sample rate etc
- Appropriate interfaces to configure the initial values of gain, sample rate etc

9.2 Installation

The Aic31 device driver is a part of PSP package for DA830 platform and would be installed as part of whole package installation. For high level design information please refer to the driver architecture guide that came with this package (available at <ID>\ti\psp\platform\evmDA830\codec\docs)

9.2.1 Codec Component folder

On installation of PSP package for DA830, the codec driver can be found at <ID>\ ti\psp\platforms\evmDA830\codec

As show above the Codec folder contains sub-folder, contents of which are described below.

- **codec -** The codec folder is the place holder for the entire codec driver, documents and the build configuration files. Codec driver is implemented as IDriver under DSP/BIOS™ operating system. Stream defined APIs could be used to interface to codec driver. This folder contains the build configuration file(package.bld),the Codec module specification file (Aic31.xdc),the module script file (Aic31.xs) and the miscellaneous files required for compiling the Aic31 codec driver library.
- **docs –** Holds Aic31 driver's architecture. Please note that the API reference would be found as a part of cdoc help (<install dir>\packages\docs\DA830\cdoc\ index.html).
- **Aic31_src –** Place holder for Aic31 driver's source code.

9.2.2 Build Options

Aic31 codec device driver will be built with "whole_program_debug" mode. When built successfully the respective library will be available at

<ID>\ti\psp\platforms\evmDA830\codec\lib\
<ti.psp.platform.evmDA830.Aic31.a674>

9.3 Features

This section details the features of Aic31 codec driver and how to use them in detail.

9.3.1 Multi-Instance

The Aic31 codec driver can operate on all the instances of Aic31 on DA830 board. Different instances are specified during driver creation time. Supported instance currently are 0 with instance id 0.

These instances could be operated simultaneously with configurations supported by AIc31 driver.

The device ID could be specified using the `instNum` field of structure Aic31_Params. There are two ways in which a new instance of the Aic31 driver can be created.

1. Static creation – static creation of the driver is done in the "cfg" file of the application. The creation happens at compile time.
2. Dynamic creation – Dynamic creation of driver is done in the application source files and the creation happens at runtime.

(i.e. Aic31_Params.`instNum = 0x0`)

9.3.2 Independent configuration of transmit and receive sections

Aic31 driver can be used to configure the transmitter and receiver section of the Aic31 codec independently. Each of the sections can be configured independently by opening a stream Channel as an INPUT channel and opening a stream Channel as an OUTPUT channel. The type of Channel is specified while creating the channel (using `Stream_create ()specify "DriverTypes_OUTPUT" or "DriverTypes_INPUT"`). The configuration parameters are explained in the sections to follow.

9.3.3 Interfaces to control the codec

The Aic31 driver provides the interface to control the specific features of the codec through a well defined set of IOCTL commands.The IOCTL commands supported are listed in the section 9.5

9.4 Configurations

Following tables document some of the configurable parameter of AIC31. Please refer to Aic31.h (Aic31.xdc) for complete configurations and explanations.

9.4.1 Module configuration

The following parameters are module wide configurable parameters

Variable	Description
`paramCheckEnable`	Option to select if parameter checking at the function entry should be enabled or disabled.
`i2cTimeout`	Timeout to be used by the I2c driver when configuring the codec

9.4.2 Aic31_Params

This structure defines the device configurations, expected to supply while creating the driver. This is provided when driver channels are created (e.g. stream_create).

Members	Description
acType	Type of the codec
instNum	Instance number of the codec to use.
acControlBusType	Control bus to be used by the AIC for configuring of the codec(I2C/SPI)
acOpMode	Operational mode of the codec(Master/slave)
acSerialDataType	Data transfer format(DSP/TDM/I2c etc)
acSlotWidth	Slot width of the data
acDataPath	Mode to configure the codec.
isRxTxClockIndependent	is the clocks for the RX and TX sections independent

9.4.3 Aic31_ChannelConfig

Applications could use this structure to configure the channel specific configurations.

Members	Description
samplingRate	Audio data sampling rate to be used
chanGain	The initial gain to be programmed(To be specified in percent)
bitClockFreq	Bit clock frequency to be used
numSlots	Number of slots for the audio data

9.4.4 Codec Configuring

The codec usually is configured using an I2C bus or a SPI bus. Hence the codec internally uses an I2c or SPI driver to configure the codec. The codec uses only the interrupt mode of the driver to configure the codecs. It also uses a call back function to synchronize each access done to/with the control bus.

Note: The current codec driver supports only the I2c bus to configure the codec.

9.5 Control Commands

Following table describes some of important the control commands, for a comprehensive list please refer the ICOTL defined in `ICodec.h (ICodec.xdc)`.

Command	Arguments	Description
Aic31_AC_IOCTL_MUTE_ON	None	Configures the mute for the codec
Aic31_AC_IOCTL_MUTE_OFF	None	Disables the
Aic31_AC_IOCTL_SET_VOLUME	UInt32 *	Set the required volume for the codec
Aic31_AC_IOCTL_SET_LOOPBACK	None	Not supported

Aic31_AC_IOCTL_SET_SAMPLERATE	UInt32 *	Gets the current sample rate for the audio codec
Aic31_AC_IOCTL_REG_WRITE	Aic31_RegData *	Writes to the specified register
Aic31_AC_IOCTL_REG_READ	Aic31_RegData *	Reads from the specified register
Aic31_AC_IOCTL_REG_WRITE_MULTIPLE	Aic31_RegData *	Writes to the specified number of registers
Aic31_AC_IOCTL_REG_READ_MULTIPLE	Aic31_RegData *	Reads from the specified number of registers
Aic31_AC_IOCTL_SELECT_OUTPUT_SOURCE	Aic31_OutputDest	The output audio port can be specified(HPOUT,LINOUT or both)
AC_IOCTL_SELECT_INPUT_SOURCE	Aic31_InputDest	The input audio port selection can be specified(MIC IN or LINE IN)

9.6 Use of AIC31 driver through Stream APIs

Following sections explain the use of parameters of Stream calls in the context of AIC31 driver. Note that no effort is made to document the use of Stream calls; any AIC31 specific requirements are covered below.

9.6.1 Stream_create

Parameter Number	Parameter	Specifics to AIC31
1	Device Name string	Unique identifier used to identify this driver. Please note the name should be same as specified while creating the driver. (Either through CFG or Aic31_create ()
2	IO Type	Should be "DriverTypes_INPUT" when AIC31 requires to configure the receive section and "DriverTypes_OUTPUT" when AIC31 requires to configure transmit section.
3	Stream_Params *	Parameters required for the creation of the stream (e.g. channel parameters)
4	Error_Block *	Pointer to the application supplied error block

9.6.2 Stream_control

Parameter Number	Parameter	Specifics to AIC31
1	Stream_handle	Handle to the channel returned during the "Stream_create"
2	Command	IOCTL command defined by Aic31 driver
3	Arguments	Misc arguments if required by the command
4	Error_Block *	Pointer to the Application supplied error block

9.6.3 Aic31_create

Parameter Number	Parameter	Specifics to AIC31
1	Aic31_params *	Pointer to the Aic31_params structure required for the Driver creation

9.7 Sources that need re-targeting

9.7.1 SoC level changes

When the driver has to adapt to SoC level changes the two files Aic31.xs (Module Script File) and the SoC script file soc.xs need to be updated with the changes.

9.7.2 EVM level changes

When the platform/EVM changes, then the platform.xs needs to updated with the relevant information like the codec I2C slave address etc.

9.8 EDMA3 Dependency

Aic31 driver does not use the EDMA mode of transfer. It does not handle any kind of data transfer requests from the application.

9.9 Known Issues

Please refer to the top level release notes that came with this release.

9.10 Limitations

Please refer to the top level release notes that came with this release.

10 LCDC Raster Controller Driver

10.1 Introduction

This document is the reference guide for the device driver which explains the features and tips to use them.

DSP/BIOS applications use the driver typically through APIs provided by Stream layer, to transmit and receive serial data. The following sections describe in detail, procedures to use this driver, configure among others…. It is recommended to go through the sample application to get a feel of initializing and using the LCDC Raster driver

10.1.1 Key Features

- Multi-instance able, asynchronous and re-entrant driver
- Each instance operates as a raster controller instance of the LCDC
- Supports multiple frame sizes – only limited by the hardware

10.1.2 References

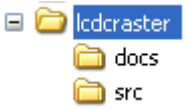
1 SPRUFM0 DA830 LCDC User's Guide

10.2 Installation

The LCDC Raster device driver is a part of PSP package for DA830 platform and would be installed as part of whole package installation. For high level design information please refer to the driver architecture guide that came with this package (available at <ID>\ti\psp\lcdcrasater\docs)

10.2.1 LCDC Raster Component folder

On installation of PSP package for DA830, the LCDC Raster Controller driver can be found at <ID>\ ti\psp\lcdcraster\



As show above the LCDC Raster contains sub-folder, contents of which are described below.

- **lcdcraster -** The lcdcraster folder is the place holder for the entire lcdcraster driver, documents and the build configuration files. LCDC Raster driver is implemented as IDriver under DSP/BIOS™ operating system. Stream defined APIs could be used to interface to LCDC Raster driver. This folder contains the build configuration file (package.bld),the LCDC Raster module specification file (Raster.xdc),the module configuration file (Raster.xs) and the miscellaneous files required for compiling the LCDC Raster library.
- **docs –** Holds LCDC Raster driver's architecture. Please note that the API reference would be found as a part of cdoc help (<ID>\packages\docs\DA830\cdoc\index.html)
- **src –** Place holder for LCDC Raster driver's source code.

10.2.2 Build Options

LCDC Raster device driver will be built with "whole_program_debug" mode. When built successfully the respective library will be available at <ID>\ti\psp\lcdcraster\lib\< ti.psp.lcdcraster.a674>

10.3 Features

This section details the features of LCDC Raster (henceforth also referred to as Raster) and how to use them in detail.

10.3.1 Multi-Instance

The Raster driver can operate on all the instance of LCDC Raster Controller on DA830. Different instances are specified during driver creation time. Supported instance are 0 only with device ID 0 only.

This instance could be operated with configurations supported by Raster driver.

The device ID could be specified using the `instNum` field of structure Raster_Params. There are two ways in which a new instance of the Raster driver can be created.

3. Static creation – static creation of the driver is done in the "cfg" file of the application. The creation happens at compile time.
4. Dynamic creation – Dynamic creation of driver is done in the application source files and the creation happens at runtime.

(i.e. Raster_Params.`instNum = 0x0`)

10.3.2 I/O using raster driver

The Raster driver can operate only in output mode. This is because, the LCDC Raster controller can only output image data onto the Raster LCD displays, using the concept of frame buffers. There is nothing to be read. Hence, the driver only supports a "write" channel creation.

10.4 Configurations

Following tables document some of the configurable parameter of LCDC raster device. Please refer to Raster.h (Raster.xdc) for complete configurations and explanations.

10.4.1 Module configuration

The following parameters are module wide configurable parameters

Variable	Description
`paramCheckEnable`	Option to select if parameter checking at the function entry should be enabled or disabled.
`captureEventStatistics`	Option

10.4.2 Device Parameters

This structure defines the device configurations, expected to supply while instantiating the driver (formerly known as devparams)

Raster_Params

Serial Number	Parameter	Description
1	instNum	The hardware instance number
2	devConf	The device configuration provided as a DeviceConf structure

10.4.2.1 DeviceConf

This structure defines the LCDC device setting configuration.

Serial Number	Parameter	Description
1	clkFreqHz	The output pixel clock frequency desired to be set
2	opMode	Mode of operation
3	hwiNum	The HWI event number assigned to the group the LCDC CPU event belongs to
4	dma	Configuration for the DMA controller internal to LCDC. This is provided as a DmaConfig structure

Note: The only mode of operation supported by the LCDC raster driver is DMAINTERRUPT mode. This utilizes the independent DMA controller that the LCDC controller is provided with. This DMA is different from the EDMA peripheral of the DA830. This DMA takes care of transferring the data in terms of frame buffer from external RAM to the display. This DMA can be configured as noted above in via *DeviceConf* structure and as described below via *DmaConfig* structure. For further details refer to DA830 LCDC User's Guide.

10.4.2.2 Internal DMA Configuration

This structure defines the parameters to configure the DMA operation, internal to the LCDC controller.

DmaConfig

Serial Number	Parameter	Description
1	fbMode	The device should operate in single frame buffer mode or double frame buffer mode (ping-pong mode)
2	burstSize	The chunks of 4-bytes in which the DMA should transfer the data
3	bigEndian	The operation is big endian mode or little India mode

4	eofInt	To enable End Of Frame interrupts

Note: The driver supports only little endian mode of operation. Hence big-Endian should be set to false.

10.4.3 Channel Parameters

The channel parameters configure the raster controller operation and are described below.

ChanParams

Serial Number	Parameter	Description
1	Controller	The controller type to be configured. This should be configured as a raster controller
2	chanConf	The Raster controller configuration, given as RasterConf
3	heapHandle	The heap handle to be used if the driver was to allocate the frame buffer memory on application's behalf

Note:

The allocation of memory for the frame buffer is purely on application's behalf. This happens, when the application asks the driver to allocate memory for the frame buffers it requires, via IOCTL calls. In such cases, dynamic allocation happens from heap. The heap from which the allocation should be made, should be defined by the application. For this, the application should create a heap instance and pass the handle to this heap via heapHandle. In case the heapHandle is NULL and the application requests for allocation, then the driver tries to allocate the frame buffer from the default heap of the system. Please note that the size of this heap should be sufficiently large to accommodate memory for all the buffers. However, the application may choose not to allocate the frame buffers via driver and instead just pass the buffers it has populated to the driver. The driver shall simple processes these buffers and no dynamic allocation happens in the driver.

10.4.3.1 Raster controller configuration

RasterConf

Serial Number	Parameter	Description
1	outputFormat	Right aligned or left aligned, TFT or STN data format
2	intface	The physical data interface with the display
3	panel	Whether STN or TFT type of

		panel. For raster It should be TFT
4	display	If monochrome or colour display is interfaced
5	bitsPP	The number of bits per pixel
6	fbContent	If the frame buffer contains frame data, pallete, or both
7	dataOrder	The order of data is arranged is 'LSB to MSB' or 'MSB to LSB'
8	nibbleMode	If the nibble mode should be enabled. This is true for bits per pixel less than 8 bits
9	subPanel	The configuration required for sub-panel, when enabled
10	timing2	The configuration required for SYNC signals and their polarity control
11	fifoDmaDelay	The delay after which the raster should generate DMA request to the internal DMA controller
12	intMask	Interrupts which need to be enabled
13	hFP	Horizontal front porch length in terms of number of pixel clock cycles
14	hBP	Horizontal back porch length in terms of number of pixel clock cycles
15	hSPW	Horizontal sync pulse width in terms of number of pixel clock cycles
16	pPL	Number of pixels per line
18	vFP	vertical front porch length in terms of number of line clock cycles
19	vBP	vertical back porch length in terms of number of line clock cycles
20	vSPW	vertical sync pulse width in terms of number of line

		clock cycles
21	lPP	Number of lines per panel

Note:

The raster configuration must be carefully provided depending upon the image size and resolution (vertical and horizontal width, bits-per-pixel and size), display panel specification (horizontal and vertical pulse width parameters). The fields of interest would be output format for different BPP modes, lines per panel and pixels per lines in case of different display panels.

10.5 Control Commands

Following some of the important control commands for the raster controller driver

Command	Arguments	Description
Raster_IOCTL_GET_DEVICE_CONF	Pointer to DeviceConf structure	To get the current device configuration
Raster_IOCTL_GET_RASTER_CONF	Pointer to RasterConf structure	To get the current raster configuration
Raster_IOCTL_GET_RASTER_SUBPANEL_CONF	Pointer to SubPanel structure	To get the current raster sub panel configuration
Raster_IOCTL_SET_RASTER_SUBPANEL_EN	Pointer to boolean variable	If boolean is true then enables subpanel, else disables subpanel
Raster_IOCTL_SET_RASTER_SUBPANEL_POS	Pointer to SubpanelPos enum variable	To configure the position of the raster subpanel
Raster_IOCTL_SET_RASTER_SUBPANEL_LPPT	Pointer to interger variable	To configure the number of lines to be refreshed in the subPanel
Raster_IOCTL_SET_RASTER_SUBPANEL_DATA	Pointer to interger variable	To configure the default pixel data outside the subPanel
Raster_IOCTL_GET_DMA_CONF	Pointer to DmaConfig structure	To get the current DMA configuration setting
Raster_IOCTL_SET_DMA_FB_MODE	Pointer to DmaFb enum variable	To set the frame buffer mode for the
Raster_IOCTL_SET_DMA_BURST_SIZE	Pointer to the DmaBurstSize enum	To set the DMA burst size
Raster_IOCTL_SET_DMA_EOF_INT	Pointer to Boolean variable	To enable/disable the end-of-frame interrupt
Raster_IOCTL_ADD_RASTER_EVENT	Pointer to Integer variable containing the interrupt mask	To enable a specific event interrupt enable
Raster_IOCTL_REM_RASTER_EVENT	Pointer to integet variable containing interrupt mask	To disable a specific event interrupt disable

Raster_IOCTL_GET_EVENT_STAT	Pointer to EvenStat structure	To get the current event statistics
Raster_IOCTL_CLEAR_EVENT_STAT	None	Clears the current event statistics
Raster_IOCTL_RASTER_ENABLE	None	To enable the raster controller
Raster_IOCTL_RASTER_DISABLE	None	To disable the raster controller
Raster_IOCTL_GET_DEVICE_VERSION	Pointer to Interger variable	To get the current version of the controller
Raster_IOCTL_ALLOC_FB	Pointer to a frame buffer pointer	To allocate a frame buffer on application's behalf
Raster_IOCTL_FREE_FB	Pointer to a frame buffer	To de-allocate a frame buffer in application's behalf

10.6 Use of RASTER driver through Stream APIs

10.6.1 Stream_create

Parameter Number	Parameter	Specifics to Raster
1	Device Name string	Unique identifier used to identify this driver. Please note the name should be same as specified while creating the driver. (Either through CFG or Raster_create `()`
2	IO Type	Should be "`DriverTypes_OUTPUT`"
3	Stream_Params *	Parameters required for the creation of the stream (e.g. channel parameters)
4	Error_Block *	Pointer to the application supplied error block

10.6.2 Stream_control

Parameter	Parameter	Specifics to Raster

Number		
1	Stream_handle	Handle returned by `Stream_create`
2	Command	IOCTL command defined by RASTER driver
3	Arguments	Misc arguments if required by the command
4	Error_Block *	Pointer to the Application supplied error block

10.6.3 Stream_issue/reclaim

Parameter Number	Parameter	Specifics to Raster
1	Channel Handle	Handle returned by `Stream_create`
3	Pointer to buffer	Should be pointer to framebuffer of type
4	Size	Size of the transaction
5	Error_Block *	Pointer to the Application supplied error block

10.6.4 Raster_create

Parameter Number	Parameter	Specifics to Raster
1	Raster_params *	Pointer to the Raster_params structure required for the Driver creation

10.7 Sources that need re-targeting

10.7.1 SoC level changes

When the driver has to adapt to SoC level changes the two files Raster.xs (Module Script File) and the SoC script file soc.xs need to be updated with the changes.

10.7.2 EVM level changes

None

10.8 EDMA3 Dependency

The raster controller driver does not rely on the EDMA LLD driver. The raster controller interacts with an independent DMA controller provided to it and does not use any EDMA3 paramsets.

10.9 Known Issues

Please refer to the top level release notes that came with this release.

10.10 Limitations

- The LCDC controller on DA830 has two modes of operation. One is the Raster mode and the other is the LIDD mode. However, only one mode can be operation can be chosen at a time. Following this constraint, the drivers for these two modes have been separated out and the each mode has a different driver/module , namely Raster and Lidd. Only one driver should be used at a time.

For other limitations, please refer to the top level release notes that came with this release.

10.11 Raster Sample Application

10.11.1.1 Description:

This sample demonstrates the use of the LCDC Raster.

The LCDC Raster driver along with the required component modules are configured statically in rasterSample.cfg file. It also instantiates the I2C driver to configure the I2C GPIO expander on UI board, to configure it to select routing of signals the raster display.

The rasterSample.cfg file contains the remaining BIOS configuration like the configuration of the event combiner etc. This helps to map the LCDC events to the CPU interrupts.

The main () function configures the PINMUX and uses the Psc module to enable the LCDC peripheral. It creates a task *'rasterSampleTask()'* to run the sample application.

The rasterSampleTask() task exercises the Raster driver. It also, utilizes the I2C driver to read/write to the I2C GPIO expander on the UI board to route the LCDC signals to the display.

It uses Stream APIS to create I2C and LCDC Raster driver channels and also to perform the IO operations.

10.11.1.2 Build:

This sample can be built using the CCS4 interface.

IMPORTANT NOTE: The I2C driver contains EDMA references. and hence, user should ensure that the EDMA package path is properly taken care of in the project.

There is also facility for users to compile the project using the command line. The file package.bld takes care of the necessary steps to compile the project from command line.

Please refer to the "Integration Guide" section for more details about building the project.

10.11.1.3 Setup:

The sample does not need any special setup apart from plugging in the DA830 User Interface module.

10.11.1.4 Output:

When the sample is run a baby image with a scrolling line on the image is displayed on the raster display

11 LCDC LIDD Controller Driver

11.1 Introduction

This document is the reference guide for the device driver which explains the features and tips to use them.

DSP/BIOS applications use the driver typically through APIs provided by Stream layer, to transmit and receive serial data. The following sections describe in detail, procedures to use this driver, configure among others…. It is recommended to go through the sample application to get a feel of initializing and using the LIDD driver

11.1.1 Key Features

- Multi-instance able, asynchronous and re-entrant driver
- Each instance operates as a LIDD controller instance of the LCDC
- Supports multiple display types

11.1.2 References

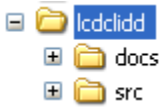
1 SPRUFM0 DA830 LCDC User's Guide

11.2 Installation

The LCDC LIDD device driver is a part of PSP package for DA830 platform and would be installed as part of whole package installation. For high level design information please refer to the driver architecture guide that came with this package (available at <ID>\ti\psp\lcdclidd\docs)

11.2.1 LCDC LIDD Component folder

On installation of PSP package for DA830, the LCDC LIDD Controller driver can be found at <ID>\ ti\psp\lcdclidd\



As show above the LIDD folder contains sub-folder, contents of which are described below.

- **lcdclidd -** The lcdclidd folder is the place holder for the entire lcdclidd driver, documents and the build configuration files. LCDC LIDD driver is implemented as IDriver under DSP/BIOS™ operating system. Stream defined APIs could be used to interface to LCDC Raster driver. This folder contains the build configuration file (package.bld),the LCDC LIDD module specification file (Lidd.xdc),the module configuration file (Lidd.xs) and the miscellaneous files required for compiling the LCDC LIDD library.

- **docs –** Holds LCDC LIDD driver's architecture. Please note that the API reference would be found as a part of cdoc help (<ID>\packages\docs\DA830\cdoc\index.html)
- **src –** Place holder for LCDC LIDD driver's source code.

11.2.2 Build Options

LCDC LIDD device driver will be built with "whole_program_debug" mode. When built successfully the respective library will be available at <ID>\ti\psp\lcdclidd\lib\< ti.psp.lcdclidd.a674>

11.3 Features

This section details the features of LCDC LIDD (henceforth also referred to as LIDD) and how to use them in detail.

11.3.1 Multi-Instance

The LIDD driver can operate on all the instance of LCDC LIDD Controller on DA830. Different instances are specified during driver creation time. Supported instance are 0 only with device ID 0 only.

This instance could be operated with configurations supported by Raster driver.

The device ID could be specified using the `instNum` field of structure Lidd_Params. There are two ways in which a new instance of the LIDD driver can be created.

5. Static creation – static creation of the driver is done in the "cfg" file of the application. The creation happens at compile time.
6. Dynamic creation – Dynamic creation of driver is done in the application source files and the creation happens at runtime.

(i.e. Lidd_Params.`instNum = 0x0`)

11.3.2 I/O using LIDD driver

The LIDD driver can operate only in output mode. This is because, the LCDC LIDD controller can only output data onto the passive LCD displays. There is nothing to be read. Hence, the driver only supports a "write" channel creation.

11.4 Configurations

Following tables document some of the configurable parameter of LCDC raster device. Please refer to Lidd.h (Lidd.xdc) for complete configurations and explanations.

11.4.1 Module configuration

The following parameters are module wide configurable parameters

Variable	Description
`configDelay`	
`paramCheckEnable`	Option to select if parameter checking at the function entry should be enabled or disabled.
`NUM_CHAR_PER_LINE`	Option to configure the number of characters per line.

NUM_LINE_PER_PANEL	Option to configure the number of Lines per panel

11.4.2 Device Parameters

This structure defines the device configurations, expected to supply while instantiating the driver (formerly known as devparams)

Raster_Params

Serial Number	Parameter	Description
1	instNum	The hardware instance number
2	devConf	The device configuration provided as a DeviceConf structure

11.4.2.1 DeviceConf

This structure defines the LCDC device setting configuration.

Serial Number	Parameter	Description
1	displayType	Type of display interfaced
2	clkFreqHz	MCLK frequency desired
3	hwiNum	The HWI event number assigned to the group the LCDC CPU event belongs to
4	funcSet	Function configuration for character LCD display interface
5	addressArray	Array of line start addresses for each line incase of character LCD

Note: Currently maximum of four line display is supported. The user needs to fill in the addresses for all the lines even if using less than 4 lines. In this case, the user can fill zero for the address for lines not used.

11.4.3 Channel Parameters

The channel parameters configure the raster controller operation and are described below.

ChanParams

Serial Number	Parameter	Description
1	Controller	The controller type to be

		configured. This should be configured as a LIDD controller
2	chanConf	The LIDD controller configuration, given as DisplayConf

11.4.3.1 Display Configuration configuration

DisplayConf

Serial Number	Parameter	Description
1	cs0Timing	Strobe signal timong configuration for device connected on CS0 chip select
2	cs1Timing	Strobe signal timing configuration for device connected on the CS1 chip select

11.5 Control Commands

Following some of the important control commands for the raster controller driver

Command	Arguments	Description
IOCTL_CLEAR_SCREEN	Pointer to ioctlCmdArg type variable.	To clear the display screen, connected on chipSelect specified by the ioctlCmdArg
IOCTL_CURSOR_HOME	Pointer to ioctlCmdArg type variable.	To set the cursor to home position, for the display connected on the chipsel specified by the ioctlCmdArg
IOCTL_SET_CURSOR_POSITION	Pointer to CursorPosition structure	To set the cursor to a particular position in the display
IOCTL_SET_DISPLAY_ON	Pointer to ioctlCmdArg type variable.	To turn the display on for the chipsel specified by the ioctlCmdArg
IOCTL_SET_DISPLAY_OFF	Pointer to ioctlCmdArg type variable.	To turn the display off for, the chipsel specified by the ioctlCmdArg

IOCTL_SET_BLINK_ON	Pointer to ioctlCmdArg type variable.	To turn the cursor blink on for display, on the chipsel specified by the ioctlCmdArg
IOCTL_SET_BLINK_OFF	Pointer to ioctlCmdArg type variable.	To turn the cursor blink off for display, on the chipsel specified by the ioctlCmdArg
IOCTL_SET_CURSOR_ON	Pointer to ioctlCmdArg type variable.	To show the cursor for display, on the chipsel specified by the ioctlCmdArg
IOCTL_SET_CURSOR_OFF	Pointer to ioctlCmdArg type variable.	To not show the cursor for display, on the chipsel specified by the ioctlCmdArg
IOCTL_SET_DISPLAY_SHIFT_ON	Pointer to ioctlCmdArg type variable.	To turn the display shift on for display, on the chipsel specified by the ioctlCmdArg
IOCTL_SET_DISPLAY_SHIFT_OFF	Pointer to ioctlCmdArg type variable.	To turn the display shift off for display, on the chipsel specified by the ioctlCmdArg
IOCTL_CURSOR_MOVE_LEFT	Pointer to ioctlCmdArg type variable.variable containing the interrupt mask	To move the cursor left display, on the chipsel specified by the ioctlCmdArg
IOCTL_CURSOR_MOVE_RIGHT	Pointer to ioctlCmdArg type variable.variable containing the interrupt mask	To move the cursor right display, on the chipsel specified by the ioctlCmdArg
IOCTL_DISPLAY_MOVE_LEFT	Pointer to ioctlCmdArg type variable.variable containing the interrupt mask	To move the display left, on the chipsel specified by the ioctlCmdArg
IOCTL_DISPLAY_MOVE_RIGHT	Pointer to ioctlCmdArg type variable.variable containing the interrupt mask	To move the display right, on the chipsel specified by the ioctlCmdArg
IOCTL_COMMAND_REG_WRITE	Pointer to Integer type variable	A generic IOCTL to write a command word to the Character display

11.6 Use of LIDD driver through Stream APIs

11.6.1 Stream_create

Parameter Number	Parameter	Specifics to Lidd
1	Device Name string	Unique identifier used to identify this driver. Please note the name should be same as specified while creating the driver. (Either through CFG or Lidd_create `()`
2	IO Type	Should be "`DriverTypes_OUTPUT`"
3	Stream_Params *	Parameters required for the creation of the stream (e.g. channel parameters)
4	Error_Block *	Pointer to the application supplied error block

11.6.2 Stream_control

Parameter Number	Parameter	Specifics to Lidd
1	Stream_handle	Handle returned by `Stream_create`
2	Command	IOCTL command defined by LIDD driver
3	Arguments	Misc arguments if required by the command
4	Error_Block *	Pointer to the Application supplied error block

11.6.3 Stream_issue/reclaim

Parameter Number	Parameter	Specifics to Raster
1	Channel Handle	Handle returned by `Stream_create`
3	Pointer to buffer	Should be pointer to a buffer of data
4	Size	Size of the transaction
5	Error_Block *	Pointer to the Application supplied

		error block

11.6.4 **Lidd_create**

Parameter Number	Parameter	Specifics to Raster
1	Lidd_params *	Pointer to the Lidd_params structure required for the Driver creation

11.7 Sources that need re-targeting

11.7.1 **SoC level changes**

When the driver has to adapt to SoC level changes the two files Lidd.xs (Module Script File) and the SoC script file soc.xs need to be updated with the changes.

11.7.2 **EVM level changes**

None

11.8 EDMA3 Dependency

The LIDD controller driver does not rely on the EDMA LLD driver. The raster controller interacts with an independent DMA controller provided to it and does not use any EDMA3 paramsets.

11.9 Known Issues

Please refer to the top level release notes that came with this release.

11.10 Limitations

- The LCDC controller on DA830 has two modes of operation. One is the Raster mode and the other is the LIDD mode. However, only one mode can be operation can be chosen at a time. Following this constraint, the drivers for these two modes have been separated out and the each mode has a different driver/module, namely Raster and Lidd. Only one driver should be used at a time.

For other limitations, please refer to the top level release notes that came with this release.

11.11 LIDD Sample Application

11.11.1.1 Description

This sample demonstrates the use of the LCDC LIDD driver.

The LCDC LIDD driver along with the required component modules are configured statically in liddSample.cfg file. It also instantiates the I2C driver to configure the I2C GPIO expander on UI board, to configure it to select routing of signals the raster display.

The liddSample.cfg file contains the remaining BIOS configuration like the configuration of the event combiner etc. This helps to map the LCDC events to the CPU interrupts.

The main () function configures the PINMUX and uses the Psc module to enable the LCDC peripheral. It creates a task *'liddSampleTask()'* to run the sample application.

The liddSampleTask() task exercises the LIDD driver. It also, utilizes the I2C driver to read/write to the I2C GPIO expander on the UI board to route the LCDC signals to the display.

It uses Stream APIS to create I2C and LCDC LIDD driver channels and also to perform the IO operations.

11.11.1.2 Build:

This sample can be built using the CCS4 interface.

IMPORTANT NOTE: The I2C driver contains EDMA references, and hence, user should ensure that the EDMA package path is properly taken care of in the project.

There is also facility for users to compile the project using the command line. The file package.bld takes care of the necessary steps to compile the project from command line.

Please refer to the "Integration Guide" section for more details about building the project.

11.11.1.3 Setup:

- The Raster display should be removed from the DA830 Interface Module (UI board)
- The HDM24216-H 24x2 character display should be plugged into J2 on the UI board.
- The R55 potentiometer should be adjusted to provide sufficient voltage (4.5-4.7V). To verify ensure this see that first line of display shows 24 squares glowing brightly.

11.11.1.4 Output:

When the sample is run a Welcome scrolling message is displayed on the character display module and the sample application performs some operations on the same.

12 BLOCK MEDIA driver

12.1 Introduction

This section is the reference guide for the Block media device driver which explains the features and tips to use them.

DSP/BIOS applications use the block media driver through the PSP APIs provided by Block media package. The following sections describe in detail, procedures to use this driver and configure it. It is recommended to go through the sample application of storage drivers to get familiar with initializing and using the Block media driver.

Please note that the Block Media driver can interface with the ERTFS file system calls.

12.1.1 Key Features

- Provides both Sync access for File system as well as for Raw/Sector level access (for eg. USB MSC Class).
- Provides interfaces for Mass Storage Class clients like USB, NAND to talk to Storage Block devices in a uniform way.
- Provides support for big block sector sizes.
- Supports cache alignment on unaligned buffers from application.
- Provides Write Protect support, Removable media support.

12.2 Installation

The Block media device driver is a part of PSP product for DA830 and would be installed as part of whole package installation. For high level design information please refer to the driver architecture guide that came with this package (available at <ID>\ti\psp\blkmedia\docs)

12.2.1 Block Media Component folder

On installation of PSP package for the DA830, the Block media driver can be found at <ID>\ ti\psp\blkmedia\

As shown above, the block media folder contains several sub-folders, the contents of which are described below:

- **blkmedia -** The blkmedia folder is the place holder for the entire BLOCK MEDIA driver and the build configuration files. Block media driver is implemented as module under DSP/BIOS™ operating system. PSP based APIs could be used to interface to Block media driver. This folder contains `psp_blkdev.h` which is the header file included by the application. This folder contains the build configuration file (package.bld) and the miscellaneous files required for compiling the Block media library.
- **docs –** Holds Block media driver's architecture document.(Contains doxygen generated API reference documnent)
- **src –** Contains Block media driver's source code.

12.2.2 Build Options

The block media driver is _NOT_ an XDC module. Hence, it does not support the "whole_program_debug" profile option. The driver is built in debug and release modes separately in lib\Debug and lib\Release directories with individual compiler options. The release mode libraries do not contain the parameter checks, while the debug mode libraries contain the parameter checks, which are controlled via PSP_DISABLE_INPUT_PARAMETER_CHECK options.

12.3 Configurations

Following tables document some of the configurable parameter of BLOCK MEDIA. Please refer to `psp_blkdev.h` for complete configurations and explanations.

12.3.1 Configuration defines

The following configuration defines are provided:

Members	Default Values	Description
PSP_BUFF_ALIGNMENT	Enabled	This macro enables the buffer alignment mechanism in BLOCK MEDIA. If application passes unaligned buffer for read/write from storage media, then block media aligns this buffer to cache line length and passes it to storage driver. Please note that if the underlying storage driver uses EDMA mode of operation then the buffer passed to the storage driver should be cache aligned.
PSP_BUFFER_IO_SIZE	0x100000 bytes	Buffer size for IO access. This buffer is used when File System is used.
PSP_BUFFER_ASYNC_SIZE	0x7D000 bytes	Buffer size for RAW access. This buffer is used when RAW mode of media driver is used.
PSP_BLK_EDMA_MEMCPY_IO	Enabled	For buffer alignment, to enable EDMA copy for IO mode this macro must be defined. If this is undefined then BLKMEDIA will use the memcpy. This is used when alignment is required during access from file system.
PSP_BLK_EDMA_MEMCPY_ASYNC	Disabled	For buffer alignment, to enable EDMA copy for RAW mode this macro must be defined. If this is undefined then BLKMEDIA will use the memcpy. Currently the driver uses memcpy for RAW mode. This is used when alignment is required during access from RAW application.
PSP_BLK_DEV_MAXDEV	PSP_BLK_DRV_MAX = 2	Number of Instances of storage drives supported. Currently set to PSP_BLK_DRV_MAX (MMC0 and NAND) which is an enum having details of how many storage drivers are there.

12.3.2 Run time configuration

Applications could use following parameters to configure block media driver at run time. These parameters are provided when the block media driver is initialized.

Parameters	Description
hEdma	The handle to the EDMA driver.
edmaEventQ	EDMA Event Queue number to be used for Block Media.
taskPrio	Block media task priority. The priority should be greater than any other storage task priority. The value should be in supported rage of OS.
taskSize	Stack size for Block Media task. Minimum 4Kbytes.

12.3.3 Block Device IOCTL structure

Applications could use this structure for populating different ioctls (e.g. PSP_blkmediaDevIoctl)

Members	Description
Cmd	IOCTL command defined by Block media or storage driver.
pData	Pointer to misc arguments if required by the command. Data type information is defined in the IOCTL.
pData1	Second data arg., if required

12.3.4 Block Driver IOCTL structure

Applications could use this structure for raw operation of block media (e.g. PSP_blkmediaDrvIoctl)

Members	Description
Cmd	IOCTL command defined by Block media for RAW usage (e.g. PSP_BlkDrvIoctl_t).
pData	Pointer to misc arguments if required by the command. Data type information is defined in the IOCTL.
pData1	Second data arg., if required

12.4 Block media driver API's

Following sections explain the use of parameters for functions of Block media driver. The Block Media driver provides isolation so that either File System or RAW application owns a particular block device. The API's are broadly divided in to four sections:

12.4.1 Init/De-init API's

12.4.1.1 *PSP_blkmediaDrvInit* - This function initializes the block media driver, take the resources, initialize the data structure and create a block media task for storage driver registration. This function also takes EDMA channel for alignment if the option is selected. Block media needs to be initialized before any initialization to storage driver (if block media is used to access the storage driver). This function also initializes the file system (if supported).

Parameter Number	Parameter	Specifics to Block Media
1	hEdma	EDMA driver handle.
2	edmaEventQ	EDMA Event Queue number to be used for Block Media
3	taskPrio	Block media task priority. The priority should be greater than any other storage task priority. The value should be in supported rage of OS.
4	taskSize	Stack size for Block Media task. Minimum 4Kbytes.

12.4.1.2 *PSP_blkmediaDrvDeInit* - This function de-initialize the Block Media Driver. This function de-allocates any resources taken during init and deletes the task created during init. The function also frees the EDMA channel allocated during init. This function also de-init the file system (if supported).

Parameter Number	Parameter	Specifics to Block Media
1	Void	None

Note: These API are required irrespective of sample application usage (MMCSD or NAND). These API's are required to initialize and de-initialize the block media. These API's should be called only once during the system.

12.4.2 API's for storage media

12.4.2.1 *PSP_blkmediaDrvRegister* - This function registers the storage driver with Block Media Driver. Storage driver will call this function during initialization of the device with a function pointer which can be called as soon as device is detected to get the read write and ioctl pointers of the device. The same parameter is set to NULL during de-init of a storage device.

Parameter Number	Parameter	Specifics to Block Media
1	driverId	Id of the Storage Driver
2	pRegInfo	Structure containing the device register/un-register function. The function passed here will be used later to get the read write and ioctl pointers of the storage

		device.

12.4.2.2 *PSP_blkmediaCallback* - Block Driver Callback interface. This function is used for propagating events from the underlying storage drivers to the block driver, independent of the device context (Ex. Device insertion/removal, media write protected).

Parameter Number	Parameter	Specifics to Block Media
1	driverId	Id of the Storage Driver
2	pRegInfo	Storage Driver Device Event information.

Note: These API are used by storage media driver and not by applications.

12.4.3 **API's for File System**

12.4.3.1 *PSP_blkmediaDevIoctl* - Handle the BLK IOCTL commands when device is active. This IOCTL can be used to set device operation mode, get device sector size, get size of storage device etc. See supported IOCTL commands in PSP_BlkDevIoctl_t and are explained below.

Parameter Number	Parameter	Specifics to Block Media
1	driverId	Id of the Storage Driver
2	pIoctl	IOCTL info structure

Note: This API is used by Application using File System(if supported).

12.4.3.2 Control Commands - Following table describes some of important the control commands in PSP_BlkDevIoctl_t, for a comprehensive list please refer the IOCTL defined in *psp_blkdev.h*

Command	Arguments	Description
PSP_BLK_GETSECTMAX	Uint32*	Get the Max Sector information from the underlying storage driver.
PSP_BLK_GETBLKSIZE	Uint32*	Get the Block Size of one Sector on the storage media.
PSP_BLK_SETPWRMODE	None	Set the Power mode for the device. Currently this IOCTL is not supported in any driver.
PSP_BLK_SETOPMODE	PSP_BlkOpMode *	Set the Operating Mode for the storage device.(Valid if supported by the underlying media driver)
PSP_BLK_GETOPMODE	PSP_BlkOpMode *	Get the Operating Mode of the storage device
PSP_BLK_DEVRESET	None	Reset the block device. Currently this IOCTL is not supported in any driver.

PSP_BLK_GETWPSTAT	Bool*	Get the storage media write protect status.
PSP_BLK_GETREMSTAT	Bool*	Is the storage device removable or not.
PSP_BLK_SETEVENTQ	PSP_Mmcsd_Edma_EventQueue*	Set Event queue of EDMA channel for storage media.
PSP_BLK_IOCTL_MAX	None	This IOCTL is added to the any specific media ioctl to use the media specific ioctls.

12.4.4 API's for Non File system application

12.4.4.1 *PSP_blkmediaAppRegister* - The Media Driver clients like Mass Storage drivers shall use this function to register a storage driver as RAW application for a Block media device.

Parameter Number	Parameter	Specifics to Block Media
1	AppCb	Address of the callback function of application which will be called after every read and write.
2	pIntOps	Block Interface driver structure with member DevOps having read write and ioctl function pointers. PSP_BlkDevOps_t structure will contain address of a read write and ioctl function after returning from this function. This will be use by application for read, write and ioctl functions of storage device.
3	pHandle	Block Driver Device Handle for the storage device. This will be the first arg of read, write and ioctl functions called by the application.

12.4.4.2 *PSP_blkmediaAppUnRegister* - Media Driver clients like Mass Storage drivers shall use this function to un-register from a Block device.

Parameter Number	Parameter	Specifics to Block Media
1	handle	Block Media Device handle.

12.4.4.3 *PSP_blkmediaDrvIoctl* - Handle the BLK IOCTL commands when device is active. This IOCTL can be used to set a storage device for RAW access, get which device is currently set for RAW access, set init completion callback for the storage device etc. See supported IOCTL commands in PSP_BlkDrvIoctl_t.

Parameter Number	Parameter	Specifics to Block Media
1	pDevName	Address of variable which contains Device Name

2	pIoctl	IOCTL info structure.

12.4.4.4 Control Commands - Following table describes some of important the control commands, for a comprehensive list please refer the IOCTL defined in *psp_blkdev.h*

Command	Arguments	Description
PSP_BLK_DRV_SETRAWDEV	PSP_BlkDrvId_t *	Set a device for RAW access.
PSP_BLK_DRV_GETRAWDEV	PSP_BlkDrvId_t *	Get which device is currently set for raw access.
PSP_BLK_DRV_SET_INIT_COMP_CALLBACK	Uint32 *	Sets the init completion call back function for storage device. This needs to be used only by storage drivers and not applications.

Note**:** These API are required when application wants to use the storage driver for RAW access.

12.5 Use of Block media driver for RAW application interface

The section discusses in detail about RAW application interface. The Block Media Driver provides the interfaces to access the registered block device in RAW mode. The section discusses in detail about how to interface a with block media for RAW application interface. The block media driver must be initialized before using any API of Block media.

12.5.1 Set Driver as RAW access

To set any storage device for RAW mode, application must call PSP_blkmediaDrvIoctl() function with PSP_BLK_DRV_SETRAWDEV as a command. Application has to pass the address of variable of type PSP_BlkDrvId_t, which contains the Driver id of the device as first parameter and PSP_BlkDrvIoctlInfo_t structure variable as second parameter. Driver id is enumerated in psp_blkdev.h.

Every time, before registering device for RAW access, application must inform driver about which device, application wants to set as a RAW device using PSP_blkmediaDrvIoctl() function as explained above, otherwise PSP_blkmediaAppRegister() function will fail.

For example to configure MMC as a RAW device, application needs to call following function:

```
PSP_BlkDrvIoctlInfo_t drvIoctlInfo;
PSP_BlkDrvId_t driverDev = PSP_BLK_DRV_MMC0;
drvIoctlInfo.Cmd = PSP_BLK_DRV_SETRAWDEV;
drvIoctlInfo.pData = (Void*)&driverDev;
PSP_blkmediaDrvIoctl((Void*)&device, &drvIoctlInfo);
```

12.5.2 Get RAW device

Block driver provides one more IOCTL to know which device is set as RAW Device. Application has to call PSP_blkmediaDrvIoctl() function with PSP_BLK_DRV_GETRAWDEV IOCTL command. For example

```
PSP_BlkDrvIoctlInfo_t drvIoctlInfo;
PSP_BlkDrvId_t device;
```

```
drvIoctlInfo.Cmd = PSP_BLK_DRV_GETRAWDEV;
drvIoctlInfo.pData = (Void*)&driverDev;
PSP_blkmediaDrvIoctl((Void*)&device, &drvIoctlInfo);
```

12.5.3 Register RAW Client

To register any storage device (NAND, MMCSD) as a RAW device, application needs to call PSP_blkmediaAppRegister() function by passing,

1. Address of callback function which will be called after every read and write function call.
2. Address of variable of PSP_BlkDevOps_t type structure, which will hold read, write and IOCTL function pointers.
3. Address of variable (Handle) of type void*. Block Media returns the handle of storage device in this parameter.

Application can now read, write and control device using the function pointers and (Handle) which was returned from PSP_blkmediaAppRegister() function.

For example to register MMC driver as a RAW device, application needs to call following function:

```
PSP_BlkDevOps_t pDevOps1;

PSP_BlkDevOps_t* pDevOps = &pDevOps1;

Ptr handle;

PSP_blkmediaAppRegister(&blkMmcsdTestCallBack, &pDevOps, &handle);
```

12.5.4 Read/Write

For writing and reading from the storage device, application has to call read/write function pointer, using variable PSP_BlkDevOps_t structure which was returned by PSP_blkmediaAppRegister(). Application has to pass

1. Variable (Handle) of type void* as a first argument, which was returned from PSP_blkmediaAppRegister() function.
2. Address of variable of structure PSP_BlkDevRes_t (to get error value).
3. Address of data buffer. (To or from data needs to be read or written).
4. Location of sector (Sector number) where data is required to be written.
5. Number of sectors to be written. (Size of data (bytes)/sector size (byte)).

For example, to read/write 1024 bytes from 0th sector number of MMC device which has been registered as a RAW device, application needs to call following function:

```
PSP_BlkDevRes_t MMCSD_TestInfo;

Uint8  srcmmcsdBuf[1024];

Uint8  dstmmcsdBuf[1024];

pDevOps->Blk_Write(handle, (Ptr)&MMCSD_TestInfo,  srcmmcsdBuf, 0, 2);
```

```
pDevOps->Blk_Read(handle, (Ptr)&MMCSD_TestInfo, dstmmcsdBuf, 0, 2);
```

12.5.5 IOCTL

For writing and reading from the storage device, application has to call ioctl function pointer, using variable PSP_BlkDevOps_t structure which was returned by PSP_blkmediaAppRegister(). Application has to pass

1. Variable (Handle) of type void* as a first argument, which was returned from PSP_blkmediaAppRegister() function.
2. Address of variable of structure PSP_BlkDevRes_t (to get error value).
3. Address of variable of structure PSP_BlkDevIoctlInfo_t containing the ioctl information.
4. Address of a bool variable.

For example, to get block size from the storage device which has been registered as a RAW device, application needs to call following function:

```
PSP_BlkDevRes_t MMCSD_TestInfo;

PSP_BlkDevIoctlInfo_t   ioctlInfo;

Uint32                  blockSize;

Bool                    isComplete;

ioctlInfo.Cmd = PSP_BLK_GETBLKSIZE;

ioctlInfo.pData = (Void*)&blockSize;

pDevOps->Blk_Ioctl(handle, (Ptr)&MMCSD_TestInfo, &ioctlInfo,
&isComplete);
```

12.5.6 Unregister RAW device

To un-register a device, Block media driver provides PSP_blkmediaAppUnRegister() function. Application needs to pass variable (Handle) which was returned in PSP_blkmediaAppRegister() function.

For example to un-register a device which has been registered as a RAW device, application needs to call following function:

```
PSP_blkmediaAppUnRegister(Handle);
```

12.6 Use of Block Media driver for File System Interface

Block media driver is an interface layer between ERTFS and low level device driver for storage. Block media provides adoption of storage driver to ERTFSOnce the block media driver is initialized then the application can call any of the ERTFS API. Following is the special case for interfacing with block media for ioctls:

12.6.1 IOCTL

To use any IOCTL functions of the block media or storage device user can use following method

For using ioctl from the storage device, application has to call PSP_blkmediaDevIoctl () function. Application has to pass

1. Variable of type PSP_BlkDrvId_t as the first argument.
2. Address of variable of structure PSP_BlkDevIoctlInfo_t containing the ioctl information.

For example, to get block size from the storage device application needs to call following function:

```
PSP_BlkDevIoctlInfo_t   ioctlInfo;

Uint32                  blockSize;
ioctlInfo.Cmd = PSP_BLK_GETBLKSIZE;
ioctlInfo.pData = (Void*)&blockSize;

PSP_blkmediaDevIoctl(PSP_BLK_DRV_MMC0, &ioctlInfo);
```

12.7 Sources that need re-targeting

12.7.1 ti/psp/cslr/soc_DA830.h (soc specific header file):

This file contains target (SoC) specific definitions. In most cases, changing the values for the SoC specific details done here should suffice. However, if there are major changes in the hardware instance then the driver file may be needed to change.

12.7.2 EVM level changes

None

12.8 EDMA3 Dependency

Block media driver relies on EDMA3 LLD driver to move data from/to application buffers to storage buffer for unaligned application buffers; typically EDMA3 driver is PSP deliverable unless mentioned otherwise. Please refer to the release notes that came with this release. Please ensure that current PSP release is compliant with version of EDMA3 driver being used.

12.8.1.1 Used Paramset of EDMA 3

PSP driver uses TWO paramsets of EDMA3; if there are no paramsets are available the PSP driver creation would fail. These paramsets are used through the life time of PSP driver. No link paramsets are used.

12.9 Known Issues

Please refer to the top level release notes that came with this release.

12.10 Limitations

Please refer to the top level release notes that came with this release.

12.11 Block Media Sample application

The sample application can include block media library in two ways, depending upon the use case:

1. For file System

   /* USE block media Driver */

```
var Blkmedia    = xdc.loadPackage("ti.psp.blkmedia");
var Settings = xdc.useModule("ti.psp.blkmedia.Settings");

/*Use File system mode library for block media*/
Settings.mode = Settings.FILE_SYSTEM;
```

2. For Raw mode

```
/* USE block media Driver */
var Blkmedia    = xdc.loadPackage("ti.psp.blkmedia");
var Settings = xdc.useModule("ti.psp.blkmedia.Settings");

/*Use RAW mode library for block media*/
   Settings.mode = Settings.RAW;
```

NOTE: The default settings are done for File system

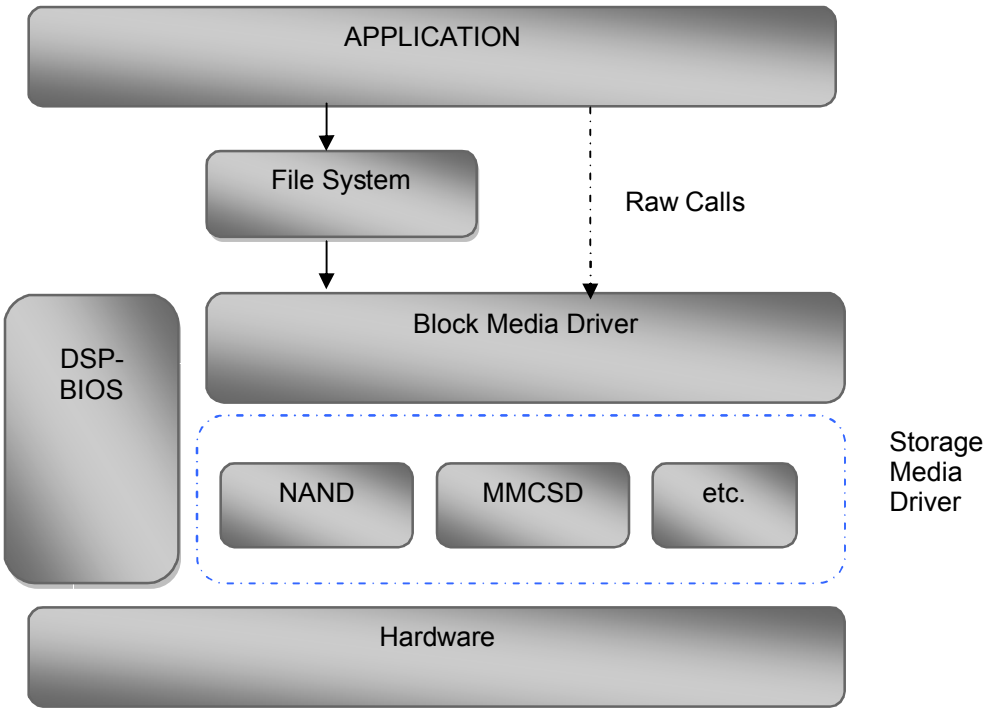
Please refer to the sample application section of NAND and MMCSD for details on interfacing block media for RAW interface.

12.12 Dependencies

The storage sample application is dependent on the following drivers

a. Block media driver
b. Storage driver (MMCSD or NAND).
c. File system(In case file system calls are used). The library for block media driver with ERTFS support is build with version ERTFS engineering version 1.10.00.02.

The block diagram below depicts the dependencies between the different drivers in the sample application. The application interact with the block media driver interface through RAW PSP block media calls or File system related calls (open, read, write etc.). The block media interface internally interacts with the registered storage media driver and finally the call comes to that particular storage media driver. The storage media drivers internally use the operation mode configured to transfer the data from the actual media device. The application needs to configure and initialize the block media first and then the storage drivers in the required modes for operation.



12.12.1 **Block media Driver**

Block Media Driver module lies below the application and file system layer. The Block Media Driver transfers calls from application/file system to the lower layer storage drivers registered. The Block media driver is synchronous driver. Block media driver is designed as a monolithic block of code in a single file as it is just a generic abstraction layer between storage media drivers and File system/applications. Storage driver gets themselves registered to the block media driver so that application can use their services seamlessly.

12.12.2 **Storage Driver**

The Storage drivers are used for data storage to various devices e.g. multimedia card (MMC)/secure digital (SD) card or NAND devices. Storage driver lies below the Block Media module. The Block Media Driver transfers calls from application/file system to the MMCSD driver which is registered to block media. The storage driver actually read/write the data to the card.

The storage device driver is partitioned and its functionality can be enacted by three key roles defined here under:

- Interfacing with the generic block media layer
- Implementing the protocol part of the driver
- Providing services to perform primitive access necessary to control/configure/examine status, of the underlying h/w device.

12.12.3 **File System**

File system can be used if it is required to have a FAT file system on the storage media. File system provided by RTFS, can be used to read and write data to a storage device. Please refer to RTFS user guide for more details. The registration of a storage driver to the file system is take care by the Block media driver. Currently File system is not supported with this release.

12.12.4 Application

The Application can interact with the Storage driver either through file system or through the RAW Calls.

13 MMCSD driver

13.1 Introduction

This section is the reference guide for the MMCSD device driver which explains the features and tips to use them.

DSP/BIOS applications use the mmcsd driver through the PSP APIs provided by MMCSD package. The following sections describe in detail, procedures to use this driver and configure it. It is recommended to go through the sample application to get familiar with initializing and using the mmcsd driver.

13.1.1 Key Features

- Re-entrant safe driver
- Provides Async IO mechanism
- Configurable to operate in Polled and DMA mode
- Supports hot removal and insertion of MMC/SD card
- Supports variety of SD and MMC cards

13.2 Installation

The MMCSD device driver is a part of PSP product for DA830 and would be installed as part of whole package installation. For high level design information please refer to the driver architecture guide that came with this package (available at <ID>\ti\psp\mmcsd\docs).

13.2.1 MMCSD Component folder

On installation of PSP package for the DA830, the MMCSD driver can be found at <ID>\ ti\psp\mmcsd\

As shown above, the mmcsd folder contains several sub-folders, the contents of which are described below:

- **mmcsd -** The mmcsd folder is the place holder for the entire MMCSD driver. and the build configuration files. MMCSD driver is implemented as module under DSP/BIOS™ operating system. PSP based APIs could be used to interface to MMCSD driver. This folder contains `psp_mmcsd.h` which is the header file included by the application. This folder contains the build configuration file (package.bld) and the miscellaneous files required for compiling the MMCSD library.
- **docs –** Holds MMCSD driver's architecture document.(Contains the doxygen generated API reference document)
- **src –** Contains MMCSD driver's source code.

13.2.2 Build Options

The MMCSD driver is _NOT_ an XDC module. Hence, it does not support the "whole_program_debug" profile option. The driver is built in debug and release modes separately in lib\Debug and lib\Release directories with individual compiler options. The release mode libraries do not contain the parameter checks, while the debug mode libraries contain the parameter checks, which are controlled via PSP_DISABLE_INPUT_PARAMETER_CHECK options.

13.3 Features

This section details the features of MMCSD and how to use them in detail.

13.3.1 Multi-Instance

The MMCSD driver can operate on the instance 0 of MMCSD on the EVM DA830.

13.3.2 Notes for Usage of Driver

- PSP_blkmediaDevIoctl() could be used to invoke IOCTL calls on the Block Media layer. Some IOCTLs are standard and need to be implemented by the underlying media layer, and these IOCTL numbers are defined in psp_blkdev.h. These IOCTLs are routed appropriately to the underlying media layer as applicable. However, some IOCTL commands may be specific for underlying media layer. In such cases the IOCTL command that is to be passed to PSP_blkmediaDevIoctl() is (PSP_BLK_IOCTL_MAX + specific command number of the underlying media layer). For example, PSP_BLK_GETOPMODE is a standard command and will return the operating mode of the underlying media layer that is queried in the IOCTL call. However, reading the registers from the MCMSD card is a specific operation on MMCSD. This IOCTL number is defined in `psp_mmcsd.h`. The command number for this should be passed as (PSP_MMCSD_IOCTL_GET_CARDREGS + PSP_BLK_IOCTL_MAX).
- Interrupt based card detection of card insertion on SD/MMC is not supported in the driver. This should be taken care by application. Please refer to the sample application for an implementation of the same using GPIO. If the application would not want interrupt based card detection of card insertion and still check the insertion of MMCSD card then it could be polled for this via PSP_mmcsdCheckCard(). There is also IOCTL which checks for presence of MMC/SD cards but this IOCTL will not work through block media layer unless underlying device is registered with block media layer, since the block media layer passes any device specific IOCTL calls to the underlying media layer.
-
- The driver, exposed to the applications, can be used either using file system mode or block media mode. **Block media mode should be considered as RAW mode** for the system. Please refer to the block media documentation for block media API's

13.4 Configurations

Following tables document some of the configurable parameter of MMCSD. Please refer to `psp_mmcsd.h` for complete configurations and explanations.

13.4.1 Run time configuration

Applications could use following parameters to configure mmcsd driver at run time. These parameters are provided when the mmcsd driver is initialized.

Parameters	Description
`moduleFreq`	MMCSD Controller clock frequency.
`instanceId`	MMCSD instance id.
`config`	MMCSD configuration pointer of type PSP_MmcsdConfig.

13.4.2 PSP_MmcsdConfig

Applications could use this structure to configure the mmcsd. This is provided when mmcsd is initialized.

Parameters	Description

opMode	MMCSD driver operating mode of type PSP_MmcsdOpMode. Only Polled and EDMA mode is supported.
hEdma	Edma Handle pointer.
eventQ	EDMA Event Queue of type PSP_MmcsdEdmaEventQueue.
hwiNumber	Hardware event number for mmcsd.

13.4.3 Polled Mode

The configurations required for polled mode of operation are:

Init configuration *opMode* should be set to PSP_MMCSD_OPMODE_POLLED. Additionally the EDMA handle parameter for the data transfer operation can be passed as NULL.

13.4.4 DMA Mode

The configurations required for DMA Interrupt mode of operation are:

Init configuration *opMode* should be set to PSP_MMCSD_OPMODE_DMAINTERRUPT. Additionally the *hwiNumber* assigned by the application for the MMCSD CPU events group should be passed, so that the driver can enable proper interrupts. Also the handle to the EDMA driver, *hEdma*, should be passed by the application. The Event Queue, *eventQ*, parameter can be set to PSP_MMCSD_EDMA3_EVENTQ_0 or PSP_MMCSD_EDMA3_EVENTQ_1.

13.5 Control Commands

Following table describes some of important the control commands, for a comprehensive list please refer the IOCTL defined in psp_mmcsd.h

Command	Arguments	Description
PSP_MMCSD_IOCTL_START	NONE	Used in RAW mode
PSP_MMCSD_IOCTL_GET_CARDREGS	PSP_MmcsdCardRegs *	Pointer to an PSP_MmcsdCardRegs variable, that would used by the driver to return back the different card register values
PSP_MMCSD_IOCTL_GET_BLOCKSIZE	Uint32*	Pointer to Uint32 variable, that would used by the driver to return back number of bytes per sector of MMC/SD device
PSP_MMCSD_IOCTL_CHECK_CARD	PSP_MmcsdCardType *	Pointer to PSP_MmcsdCardType variable, that would used by the driver to return back which card is present (MMC or SD)
PSP_MMCSD_IOCTL_GET_OPMODE	PSP_MmcsdOpMode *	Pointer to PSP_MmcsdOpMode variable that would be used by the driver to return back the operating mode of the MMCSD device.
PSP_MMCSD_IOCTL_SET_CALLBACK	PSP_MmcsdAppCallback *	Pointer to PSP_MmcsdAppCallback variable that would be used by the driver to set callback function which will be called after every read/write. This

		will be already used by Block Media so application should not use this, unless it is used for RAW mode of operation without using block media and file system.
PSP_MMCSD_IOCTL_SET_HWEVENT_NOTIFICATION	PSP_MmcsdHwEventNotification *	Pointer to PSP_MmcsdHwEventNotification variable that would use by the driver to set callback function which will be called for media insertion or removal, to notify upper layer about hardware events. This will be already used by Block Media so application should not use this, unless it is used for RAW mode of operation without using block media and file system
PSP_MMCSD_IOCTL_GET_HWEVENT_NOTIFICATION	PSP_MmcsdHwEventNotification *	Pointer to PSP_MmcsdHwEventNotification variable that would be used by the driver to return back callback function which will be called for media insertion or removal, to notify upper layer about hardware events.
PSP_MMCSD_IOCTL_GET_CARD_SIZE	Uint32 *	Pointer to Uint32 variable that would be used by the driver to return size of MMC/SD card in bytes for all cards except for High capacity card. In the case of High capacity SD card , it is returned in KBytes and using IOCTL PSP_MMCSD_IOCTL_CHECK_HIGH_CAPACITY_CARD, it could be found whether it is high capacity or not.
PSP_MMCSD_IOCTL_SET_TEMPORARY_WP	Bool *	Pointer to Bool variable, that would used by the driver to set temporary write protect state of MMC/SD card
PSP_MMCSD_IOCTL_GET_TEMPORARY_WP	Bool *	Pointer to Bool variable, that would used to get temporary write protect state of MMC/SD card
PSP_MMCSD_IOCTL_SET_PERMANENT_WP	Bool *	Pointer to Bool variable, that would used by the driver to set permanent write protect state of MMC/SD card
PSP_MMCSD_IOCTL_GET_PERMANENT_WP	Bool *	Pointer to Bool variable, that would used by the driver to get

		permanent write protect state of MMC/SD card
PSP_MMCSD_IOCTL_CHECK_HIGH_CAPACITY_CARD	Bool *	Pointer to Bool variable, that would used by the driver to check if the card is high capacity card or not. This IOCTL will return true in if it is high capacity card else false.
PSP_MMCSD_IOCTL_GET_TOTAL_SECTORS	Uint32 *	Pointer to Uint32 variable, that would used by the driver to return size of MMC/SD card in sectors
PSP_MMCSD_IOCTL_SET_EVENTQ	PSP_MmcsdEdmaEventQueue *	Pointer to PSP_MmcsdEdmaEventQueue variable, that would used by the driver to set event queue of EDMA channel
PSP_MMCSD_IOCTL_SET_CARD_FREQUENCY	PSP_CardFrequency *	Pointer to PSP_CardFrequency variable that would be used by the driver to set the frequency of card at which it is supposed to operate.
PSP_MMCSD_IOCTL_GET_CARD_VENDOR	Uint32 *	Pointer to Uint32 variable, that would used by the driver to return back the vendor id of MMC/SD
PSP_MMCSD_IOCTL_GET_CONTROLLER_REG	Uint32 * and Uint32 *	Pointer to Uint32 variable as first parameter which pass register address offset and another Uint32 pointer variable, the place holder to get value at that register offset.
PSP_MMCSD_IOCTL_SET_CONTROLLER_REG	Uint32 * and Uint32 *	Pointer to Uint32 variable as first parameter which pass register address offset and another Uint32 pointer variable, the value needs to be written at that register offset.

13.6 MMCSD Driver APIs

Following sections explain the use of parameters of MMCSD calls in the context of PSP driver. Only PSP specific requirements are covered below.

13.6.1 PSP_mmcsdDrvInit

Parameter Number	Parameter	Specifics to PSP
1	moduleFreq	MMCSD controller clock frequency

2	instanceId	MMCSD instance id number
3	config	MMCSD config parameter of type PSP_MmcsdConfig *

13.6.2 **PSP_mmcsdDrvDelnit**

Parameter Number	Parameter	Specifics to PSP
1	instanceId	MMCSD instance id number

13.6.3 **PSP_mmcsdCheckCard**

Parameter Number	Parameter	Specifics to PSP
1	cardType	MMCSD Card variable to be updated by this function. It is of type PSP_MmcsdCardType *
2	instanceId	MMCSD instance id number

13.7 Sources that need re-targeting

13.7.1 **ti/psp/cslr/soc_DA830.h (soc specific header file):**

This file contains target (SoC) specific definitions. In most cases, changing the values for the SoC specific details done here should suffice. However, if there are major changes in the hardware instance then the driver file may be needed to change.

13.7.2 **EVM level changes**

None

13.8 EDMA3 Dependency

MMCSD driver relies on EDMA3 LLD driver to move data from/to application buffers to peripheral; typically EDMA3 driver is PSP deliverable unless mentioned otherwise. Please refer to the release notes that came with this release. Please ensure that current PSP release is compliant with version of EDMA3 driver being used.

13.9 Known Issues

Please refer to the top level release notes that came with this release.

13.10 Limitations

Please refer to the top level release notes that came with this release.

13.11 MMCSD Sample applications

13.11.1 **Dma mode sample**

13.11.1.1 Description:

This sample demonstrates the use of the MMCSD driver in DMA mode.

The MMCSD driver needs I2C for selecting MMCSD pins if UI card is connected and GPIO for Card detection. This is take care by the code in platforms/evmDA830. The project file includes mmcsd.cfg which brings in all the required packages needed for the same. This file can be directly imported into an application's mmcsdSample.cfg file.

The mmcsdSample.cfg file contains the important BIOS configuration settings, which are required in order for the MMCSD driver to work properly. The most important lines in this file are as follows.

```
ECM.eventGroupHwiNum[0] = 7;
```

These lines configure the ECM module and map mmcsd events to CPU interrupts. For example the Mmcsd event number is 15 which fall in ECM group 0. Here ECM group 0 is mapped to HWI_INT7.

Once initialization has completed, the main() function runs. Following this, the user defined task pspBiosSampleApp() task exercises the mmcsd driver. The configureMmcsd function inside the platform file takes care of configuring the PINMUXes of MMCSD and GPIO (used for interrupt based detection of card insertion). and select MMCSD through I2C expander, if UI card is connected and also take care of GPIO based card detection. The init function is mmcsdStorageInit calls the edma3init, block media init and then the mmcsd init, which initializes the mmcsd driver. The edma3init() initializes the EDMA3 driver and sets up hEdma.

Please note that mmcsdStorageInit and mmcsdStorageDeinit functions provided by the platform layer are for the ease fo sample application writer. If the application wants to addresss multiple media, then these APIS should not be used as block media and edma initialization is required only once throughout the system.

The sample application uses interrupt based detection of card insertion and write protect status via GPIO. The macro Mmcsd_GPIO_CDWP_ENABLE is by default enable in the sample application.

13.11.1.2 Build:

This sample can be built using

<ID>/psp/examples/evmDA830/mmcsd

IMPORTANT NOTE: .cdtbuild contains references to %EDMA3LLD_BIOS6_INSTALLDIR% environment variable and links with edma3 libraries. This is required because by default the MMCSD driver library is built with EDMA ENABLE.

There is also another XDC based project file available for users familiar with XDC build

<ID>/psp/examples/evmDA830/mmcsd/package.bld

This project file includes mmcsdSample.cfg which brings in all the required packages.

This project requires setup of XDCPATH environment variable. The XDCPATH must contain the following -

<EDMA3_INSTALL_DIR>/packages; <PSPDRIVERS_INSTALL_DIR>/packages;

13.11.1.3 Setup:

You need to put a MMC or SD card in the MMCSD slot.

13.11.1.4 Output:

When the sample application runs, it will demonstrate the usage of MMCSD in RAW mode. The applications show the usage of various MMCSD and block media IOCTL and then do the read/write operation on some sectors of the MMC or SD card. The output can be seen on the console window.

14 NAND driver

14.1 Introduction

This section is the reference guide for the NAND device driver which explains the features and tips to use them.

DSP/BIOS applications use the driver typically through PSP APIs provided by NAND package. The following sections describe in detail, procedures to use this driver and configure it.

14.1.1 Key Features

- Supports 512-byte page and 2048-byte page NAND devices
- Supports 8-bit and 16-bit NAND devices
- Error correction using 4-bit ECC mechanism
- Supports wear-leveling and bad-block management functionalities
- Supports protecting a portion of the NAND flash from application access

14.2 Installation

The NAND device driver is a part of PSP product for DA830 and would be installed as part of whole package installation. For high level design information please refer to the driver architecture guide that came with this package (available at <ID>\ti\psp\nand\docs).

14.2.1 NAND Component folder

On installation of PSP package for the DA830, the NAND driver can be found at <ID>\ ti\psp\nand\

As shown above, the nand folder contains several sub-folders, the contents of which are described below:

- **nand -** The nand folder is the place holder for the entire NAND driver and the build configuration files. NAND driver is implemented as module under DSP/BIOS™ operating system. PSP based APIs could be used to interface to NAND driver. This folder contains `psp_nand.h` which is the header file included by the application. This folder contains the build configuration file (package.bld) and the miscellaneous files required for compiling the NAND library.
- **docs –** Holds NAND driver's architecture. (Contains doxygen generated API reference document)
- **src –** Contains Nand driver's source code.

14.2.2 Build Options

The NAND driver is _NOT_ an XDC module. Hence, it does not support the "whole_program_debug" profile option. The driver is built in debug and release modes separately in lib\Debug and lib\Release directories with individual compiler options. The release mode libraries do not contain the parameter checks, while the debug mode libraries contain the parameter checks, which are controlled via PSP_DISABLE_INPUT_PARAMETER_CHECK options.

14.3 Features

This section details the features of NAND and how to use them in detail.

14.3.1 Multi-Instance

The NAND driver can operate on 0 instance of EMIFA on the EVMDA830.

14.3.2 Supports 512-byte page and 2048-byte page NAND devices

NAND driver supports both 512-byte page and 2048-byte page devices. The driver learns about the page size of the device by looking up the device ID and manufacturer ID in the NAND device organization lookup table. Sector write and read operations are then performed for the entire length of the sector without requiring additional configurations.

14.3.3 Supports 8-bit and 16-bit NAND devices

NAND driver supports both 8-bit and 16-bit NAND devices. The driver learns about the bus width of the device by looking up the device ID and manufacturer ID in the NAND device organization lookup table. The driver configures the external memory interface module for the appropriate data bus width.

CAUTION: Driver has not been validated / tested with ONFi compliant NAND devices.

14.3.4 Error correction using 4-bit ECC

NAND driver supports error correction using 4-bit ECC algorithm. The driver uses the external memory interface module for 4-bit ECC parity generation and error correction. The parity generated during the sector write operation is copied in the spare area of the page. During sector reads, the parity stored in the spare area is read back for the error detection and correction operation.

ECC hardware used is capable of correcting a maximum of 32 bits errors, provided that these errors occur in 4 bytes for every 512 bytes of data and these 4 bytes need not be contiguous. If these 32 bits errors (or less than 32 bits but greater than 4 bits) span across 5 bytes of data in 512 byte data boundary the bit errors cannot be corrected.

14.3.5 Supports wear-leveling and bad-block management functionalities

NAND driver supports block wear-leveling and bad block management functionalities. These functionalities are transparent to the application, that is, the applications need not be aware of the wear leveling and bad block management activities performed by the driver.

14.3.6 Supports protecting a portion of the NAND flash from application access

NAND driver supports protecting a portion the NAND flash from application access. The protected portion of the NAND flash starts from the second block of the NAND device to an application specified block number. The application can specify the number of blocks to be protected during the driver initialization. All the protected blocks are excluded from the read-write operations.

14.4 Configurations

This section describes the NAND driver data types, data structures, and configurable parameters of NAND driver. NAND Media could be accessed through File system or sector level (by passing file system). Following tables document some of the configurable parameter of NAND. Please refer to `psp_nand.h` for complete configurations and explanations.

14.4.1 Configuration defines

The following configuration defines are provided:

Members	Default Values	Description
PSP_NAND_RESERVED_BLOCKS	24u	Number of blocks that would be reserved by NAND driver and would be used as a replacement block for a detected BAD block. These blocks will not be visible to applications.
PSP_NAND_MAX_PAGES_IN_BLOCK	128u	Specifies maximum number of pages that would be support by driver in a given block.
PSP_NAND_MAX_CACHE_LINES	8u	Configure maximum number of CACHE lines that NAND driver could use. Please refer the architecture document that came with this release for details.
PSP_NAND_MAX_PAGE_SIZE	2048u	Specifies the maximum size of a page that would be support by NAND driver.
PSP_NAND_FTL_MAX_LOG_BLOCKS	4096u	Maximum number of logical blocks that can be managed by FTL module. The value of this constant can be changed as per the requirement. For example, if the driver is used with a NAND device that has only 2048 blocks, then this constant can be set to 2048.
PSP_NAND_FTL_MAX_PHY_BLOCKS	4096u	Maximum number of physical blocks that can be managed by FTL module. The value of this constant can be changed as per the requirement. For example, if the driver is used with a NAND device that has only2048 blocks, then this constant can be set to 2048.

14.4.2 Nand Driver Data types

14.4.2.1 PSP_nandType - The PSP_nandType enumerated data type specifies the types of NAND devices supported by the NAND driver. Following table lists the values of the data type.

Type	Description
PSP_NT_NAND	Device type is NAND device
PSP_NT_ONENAND	Device type is OneNAND device(Not supported)
PSP_NT_INVALID	Device type is unknown

14.4.2.2 PSP_NandOpMode - The PSP_NandOpMode enumerated data type specifies the mode of operation in which the nand driver will be used. Following table lists the values of the data type.

Type	Description
PSP_NAND_OPMODE_POLLED	Polled mode of operation
PSP_NAND_OPMODE_INTERRUPT	Interrupt mode of operation(Not supported)
PSP_NAND_OPMODE_DMAINTERRUPT	DMA mode of operation

14.4.3 Nand Driver Data Structures

14.4.3.1 PSP_nandDeviceInfo - The PSP_nandDeviceInfo data structure specifies the device organization of the NAND device. Following table lists the elements of this data structure.

Members	Description
vendorId	Vendor/Manufacturer/Maker ID of NAND device
deviceId	Device ID of the NAND device
pageSize	Size of each page
pagesPerBlock	Number of pages per block
numBlocks	Number of blocks in the NAND device
spareAreaSize	Size of spare area of each page
dataBusWidth	Data bus width of the NAND device

14.4.3.2 PSP_nandDeviceTiming - The PSP_nandDeviceTiming data structure specifies the timing characteristics of the NAND device. Following table lists the elements of this data structure.

Members	Description
vendorId	Vendor/Manufacturer/Maker ID of NAND device
deviceId	Device ID of the NAND device
writeSetup	Write setup time in ns
writeStrobe	Write strobe time in ns
writeHold	Write hold time in ns
readSetup	Read setup time in ns
readStrobe	Read strobe time in ns
readHold	Read hold time in ns
turnAround	Turnaround time in ns

14.4.3.3 PSP_nandConfig - The PSP_nandConfig data structure specifies parameters for initializing and configuring the NAND driver. Following table lists the elements of this data structure.

Members	Description
inputClkFreq	EMIF input clock frequency for calculating the timing values for the EMIF
nandType	Type of NAND flash. (NAND or OneNAND)
opMode	Data transfer mode used by the NAND driver. Supported data transfer modes are polled and EDMA mode
eraseAtInit	If TRUE, enables erase of the complete NAND flash during initialization

protectedBlocks	Number of protected blocks that are not mapped as logically available storage area
hEdma	EDMA driver handle use in EDMA operating mode
edmaEvtQ	EDMA event queue number to be used in EDMA data transfer mode
nandDevInfo	NAND Device organization information
nandDevTiming	NAND device timing information

14.4.4 Polled Mode

The configurations required for polled mode of operation are:

Init configuration *opMode* should be set to PSP_NAND_OPMODE_POLLED. The EDMA handle can be NULL in this mode of operation.

14.4.5 DMA Interrupt Mode

The configurations required for DMA Interrupt mode of operation are:

Init configuration *opMode* should be set to PSP_NAND_OPMODE_DMAINTERRUPT. Also the handle to the EDMA driver, hEdma, and the event queue number should be passed by the application.

14.5 Control Commands

The PSP_nandIoctlCmd enumerated data type specifies the IOCTL commands supported by the NAND driver. When using NAND driver via File system or using RAW mode of operation via Block Media driver, use block media API PSP_blkmediaDevIoctl() to send control commands to NAND driver. Note that the command should be one of the enumerations PSP_nandIoctlCmd added with PSP_BLK_IOCTL_MAX. Following table describes some of important the control commands, for a comprehensive list please refer the IOCTL defined in psp_nand.h. Following table lists the values of the data type:

Command	Arguments	Description
PSP_NAND_IOCTL_GET_NAND_SIZE	Uint32 *	Determine the usable number of logical sectors in the device
PSP_NAND_IOCTL_GET_SECTOR_SIZE	Uint32 *	Determine the page size of the device
PSP_NAND_IOCTL_SET_EVENTQ	Uint32 *	Set the EDMA event queue for EDMA mode data transfer
PSP_NAND_IOCTL_ERASE_BLOCK	Uint32 *	Erase a logical block
PSP_NAND_IOCTL_GET_OPMODE	Uint32 *	Returns the current operation mode of NAND driver.
PSP_NAND_IOCTL_GET_DEVICE_INFO	PSP_nandDeviceInfo *	Returns the device details.

14.6 NAND Driver APIs

Following sections explain the use of parameters of NAND calls in the context of PSP driver. Only PSP specific requirements are covered below.

14.6.1 PSP_nandDrvInit

Parameter Number	Parameter	Specifics to PSP
1	config	Configuration parameters of type PSP_nandConfig * is passed.

14.6.2 PSP_nandDrvDeInit

Parameter Number	Parameter	Specifics to PSP
1	Void	None

14.7 Sources that need re-targeting

14.7.1 ti/psp/cslr/soc_DA830.h (soc specific header file):

This file contains target (SoC) specific definitions. In most cases, changing the values for the SoC specific details done here should suffice. However, if there are major changes in the hardware instance then the driver file may be needed to change.

14.8 EDMA3 Dependency

NAND driver relies on EDMA3 LLD driver to move data from/to application buffers to peripheral; typically EDMA3 driver is PSP deliverable unless mentioned otherwise. Please refer to the release notes that came with this release. Please ensure that current PSP release is compliant with version of EDMA3 driver being used.

14.8.1.1 Used Paramset of EDMA 3

PSP driver uses one paramsets of EDMA3; if there are no paramsets are available the PSP driver creation would fail. These paramsets are used through the life time of PSP driver.

14.9 Known Issues

Please refer to the top level release notes that came with this release.

14.10 Limitations

Please refer to the top level release notes that came with this release.

14.11 NAND Sample applications

14.11.1 Dma mode sample

14.11.1.1 Description:

This sample demonstrates the use of the Nand driver in DMA mode.

The NAND driver needs I2C for selecting NAND on the UI card. This is take care by the nand specific code in platforms/evmDA830. The project file includes nand.cfg which brings in all the required packages needed for the same. This file can be directly imported into an application's nandSample.cfg file.

The nandSample.cfg file contains the important BIOS configuration settings, which are required in order for the NAND driver to work properly. The most important lines in this file are as follows.

```
ECM.eventGroupHwiNum[0] = 7;
ECM.eventGroupHwiNum[1] = 8;
ECM.eventGroupHwiNum[2] = 9;
ECM.eventGroupHwiNum[3] = 10;
```

These lines configure the ECM module and map events to CPU interrupts

Once initialization has completed, the main() function runs. Following this, the user defined task pspBiosSampleApp() task exercises the nand driver. The configureNand function inside the platform file takes care of configuring the pin mux (for nand and I2C) and select NAND through I2C expander, if UI card is connected. Please refer to the platforms section in this guide for more details.

The init function is nandStorageInit calls the edma3init, block media init and then the nand init, which initializes the nand driver. The edma3init() initializes the EDMA3 driver and sets up hEdma.

Please note that nandStorageInit and nandStorageDeinit functions provided by the platform layer are for the ease of sample application writer. If the application wants to addresss multiple media, then these APIS should not be used as block media and edma initialization are required only once throughout the system.

14.11.1.2 Build:

This sample can be built using

<ID>/psp/examples/evmDA830/nand

IMPORTANT NOTE: .cdtbuild contains references to %EDMA3LLD_BIOS6_INSTALLDIR% environment variable and links with edma3 libraries. This is required because by default the NAND driver library is built with EDMA ENABLE.

There is also another XDC based project file available for users familiar with XDC build

<ID>/psp/examples/evmDA830/nand/package.bld

This project file includes nandSample.cfg which brings in all the required packages.

This project requires setup of XDCPATH environment variable. The XDCPATH must contain the following -

<EDMA3_INSTALL_DIR>/packages; <PSPDRIVERS_INSTALL_DIR>/packages;

14.11.1.3 Setup:

You need to connect a UI daughter card having NAND to the evmDA830 platform.

14.11.1.4 Output:

When the sample application runs, it will demonstrate the usage of NAND in RAW mode. The applications show the usage of various NAND and block media IOCTL and then do the read/write operation on some sectors of the NAND device. The output can be seen on the console window