User Guide

C6747 BIOS PSP User Guide

01.30.01

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 9
1.1 Introduction 9
1.2 Naming Conventions 10
1.3 Installation Guide 11
1.4 Integration Guide 15
1.5 Power Management 16

2 UART driver 19
2.1 Introduction 19
2.2 Installation 19
2.3 Features 21
2.4 Configurations 22
2.5 Control Commands 24
2.6 Use of UART driver through GIO APIs 25
2.7 Sources that need re-targeting 26
2.8 EDMA3 Dependency 26
2.9 Known Issues 26
2.10 Limitations 26
2.11 Uart Sample applications 26

3 I2C driver 29
3.1 Introduction 29
3.2 Installation 29
3.3 Features 31
3.4 Configurations 31
3.5 Control Commands 34
3.6 Use of I2C driver through Stream APIs 35
3.7 Sources that need re-targeting 36
3.8 EDMA3 Dependency 36
3.9 Known Issues 36
3.10 Limitations 36
3.11 I2c Sample applications 36

4 GPIO driver 40
4.1 Introduction 40
4.2 Installation 40
4.3 Features 42
4.4 Configurations 43
4.5 Gpio Bank Event Numbers 45
4.6 Sources that need re-targeting 45
4.7 Known Issues 45
4.8 Limitations 45
4.9 Gpio Sample application 45

5 LCDC Raster Controller Driver 47
5.1 Introduction 47

5.2 Installation 47
5.3 Features 49
5.4 Configurations 50
5.5 Control Commands 53
5.6 Use of RASTER driver through SIO APIs 54
5.7 Sources that need re-targeting 55
5.8 EDMA3 Dependency 56
5.9 Known Issues 56
5.10 Limitations 56
5.11 Raster Sample Application 56
6 LCDC LIDD Controller Driver 58
6.1 Introduction 58
6.2 Installation 58
6.3 Features 60
6.4 Configurations 60
6.5 Control Commands 62
6.6 Use of LIDD driver through GIO APIs 63
6.7 Sources that need re-targeting 65
6.8 EDMA3 Dependency 65
6.9 Known Issues 65
6.10 Limitations 65
6.11 LIDD Sample Application 65
7 SPI driver 67
7.1 Introduction 67
7.2 Installation 67
7.3 Features 69
7.4 Configurations 70
7.5 Control Commands 73
7.6 Use of SPI driver through GIO APIs 73
7.7 Use of GPIO as chip select 74
7.8 Sources that need re-targeting 76
7.9 Use of GPIO as chip select 76
7.10 EDMA3 Dependency 76
7.11 Known Issues 76
7.12 Limitations 76
7.13 Spi Sample applications 76
8 PSC driver 80
8.1 Introduction 80
8.2 Installation 80
8.3 Features 81
8.4 Use of PSC driver through module APIs 81
8.5 Sources that need re-targeting 81
8.6 EDMA3 Dependency 81
8.7 Known Issues 81
8.8 Limitations 81

9 Mcasp driver 82
9.1 Introduction 82
9.2 Installation 83
9.3 Features 84
9.4 IDLE Time Data Patterns 88
9.5 Explicit control of IO PINS 89
9.6 Clocking McASP 90
9.7 Clock Configuration (EVM6747) 91
9.8 Configurations 91
9.9 IO Request Format 94
9.10 CACHE Control 94
9.11 Control Commands 95
9.12 Use of PSP driver through SIO APIs 96
9.13 Timeline of Frame Sync, High Clock and or Bit Clock generation 97
9.14 Porting Guide 97
9.15 Sources that need re-targeting 98
9.16 EDMA3 Dependency 98
9.17 How to support "NEW" data format 98
9.18 Known Issues 98
9.19 Limitations 98
9.20 Mcasp DIT Sample application 98
9.21 McASP Sample application 99
10 Audio driver 101
10.1 Introduction 101
10.2 Installation 101
10.3 Features 102
10.4 Configurations 103
10.5 Control Commands 104
10.6 Use of Audio driver through SIO APIs 104
10.7 Sources that need re-targeting 105
10.8 EDMA3 Dependency 105
10.9 Known Issues 105
10.10 Limitations 105
10.11 Audio Sample Application 106
10.12 Dependencies 106
11 AIC31 CODEC driver 109
11.1 Introduction 109
11.2 Installation 109
11.3 Features 110
11.4 Configurations 111
11.5 Control Commands 112
11.6 Use of AIC31 driver through SIO APIs 113
11.7 Sources that need re-targeting 114
11.8 EDMA3 Dependency 114
11.9 Known Issues 114
11.10 Limitations 114

12 BLOCK MEDIA driver 115
12.1 Introduction 115
12.2 Installation 115
12.3 Configurations 117
12.4 Block media driver API's 118
12.5 Use of Block media driver for RAW application interface 122
12.6 Use of Block Media driver for File System Interface 124
12.7 Sources that need re-targeting 125
12.8 EDMA3 Dependency 125
12.9 Known Issues 125
12.10 Limitations 125
12.11 Block Media Sample application 126
12.12 Dependencies 126
13 MMCSD driver 129
13.1 Introduction 129
13.2 Installation 129
13.3 Features 130
13.4 Configurations 131
13.5 Control Commands 132
13.6 MMCSD Driver APIs 134
13.7 Sources that need re-targeting 135
13.8 EDMA3 Dependency 135
13.9 Known Issues 135
13.10 Limitations 135
13.11 MMCSD Sample applications 136
14 NAND driver 137
14.1 Introduction 137
14.2 Installation 137
14.3 Features 138
14.4 Configurations 139
14.5 Control Commands 142
14.6 NAND Driver APIs 143
14.7 Sources that need re-targeting 143
14.8 EDMA3 Dependency 143
14.9 Known Issues 143
14.10 Limitations 143
14.11 NAND Sample applications 143

1 Top level Information

1.1 Introduction

This chapter introduces the C6747 BIOS PSP to the user by providing a brief overview of the purpose and construction of the C6747 BIOS PSP**,** along with hardware and software environment specifics in the context of the C6747 BIOS PSP deployment.

1.1.1 Overview

The C6747 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 Programming Interface
CSL	TI Chip Support Library – primitive h/w abstraction.
IP	Intellectual property
ISR	Interrupt Service Routine
OS	Operating System
ID	Installation Directory
MMC	Multi-media Card
SD	Secure Digital
RTFS/ERTFS	File System

1.1.3 References

1	SPRS377A	C6747 SoC reference Guide
2	SPRU616	DSP/BIOS Device Driver Developer's Guide
3	SPRU403	TMS320C6000 DSP/BIOS Application Programming Interface
4	SPRU423	TMS320 DSP/BIOS User’s Guide

1.1.4 Supported Services and features

The C6747 BIOS PSP provides the following:

- Device drivers for UART, I2C, SPI, McASP, PSC, MMCSD, NAND, LCDC Raster,LCDC lidd and EVM specific devices like the Aic31 codec driver.
- Block Media Interface for storage drivers like MMCSD, NAND etc.
- Sample applications that demonstrate use of drivers for UART (loop back & Echo Test), I2C (writes to on board EEPROM), SPI (Serial Flash), McASP (Plays a tone, EVM to EVM communication example), MMCSD and NAND (Read/Write to the storage devices).
- rCSL and Examples for selected peripherals

1.1.5 System Requirements

The following products are required to be installed prior to using the C6747 BIOS PSP:

- EDMA 3 LLD – This package (C6747 BIOS PSP) is compatible with EDMA 3 LLD versioned 01.11.00.03 or above
- DSP-BIOS versioned 5.41.03.17
- CCS 3.3.24 or higher with service release 10
- CCS 4.0.0.16 or higher (optional)
- Code Generation Tools 6.1.9
- XDS 510 USB Emulator (Optional) – EVM has on board emulator
- EVM6747 Board
- ERTFS File System (Optional). This is required if one wants to maintain a filesystem on Storage Media. Same can be downloaded from following link:

  http://software-dl.ti.com/dsps/dsps_registered_sw/sdo_sb/targetcontent//bios_file_system/index.html

1.2 Naming Conventions

The DSP/BIOS 5 PSP drivers in this release were written based on already existing DSP/BIOS 6 PSP drivers. As such, it has been decided to maintain the same DSP/BIOS 6 naming schema for constants and modules in the driver code for consistency.

This means that module names for drivers may not be all upper case, but would have the first letter of the module name capital, followed by lower case letters. For example, the GPIO module is named:

```
Gpio
```

Constants for the Gpio module are all upper case, except that they are preceded by the module name in which they are defined. The module name which precedes is cased as described previously. One example of a Gpio module constant is:

```
Gpio_NUM_BANKS
```

This is slightly different than the normal, all uppercase naming convention found in DSP/BIOS 5, but it was done so in order to lessen confusion for the user.

1.3 Installation Guide

This chapter discusses the C6747 BIOS PSP installation, how and what software and hardware components to be availed in order to complete a successful installation (and un-installation) of the C6747 BIOS PSP.

1.3.1 Installation and Usage Procedure

1.3.1.1 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_Setup.exe) using the self extracting installer
3. Install EDMA-3 LLD Device Driver into preferred drive / folder
4. Ensure that environment variable 'EDMA3LLD_BIOS5_INSTALLDIR' is set to the packages folder of the EDMA3 installation. (e.g. If the EDMA3 LLD Driver is installed into "c:\edma3_lld_xx_yy_zz\" then set the environment variable as follows: EDMA3LLD_BIOS5_INSTALLDIR =c:\edma3_lld_xx_yy_zz\packages)
5. Optionally, if user wants to use RTFS File system install the Files system to preferred location. Ensure that environment variable 'RTFS_INSTALL_DIR' is set to the RTFS installation directory.
6. For building the downloadable images refer to section 1.4
7. Download the executable image of the required application onto your platform using CCS.
8. Run the program

Please see the help on package locations and API information help that is generated from doxygen, found under the docs folder for each driver.

1.3.1.2 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**)
   - EDMA3 LLD Device Driver un-installation
   - CCS & DSP/BIOS Product un-installation
   - Code Generation tools uninstallation

1.3.2 PSP Component Folder

This section details the files and directory structure of the installed C6747 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.

1.3.2.1 Top level PSP Directory structure:

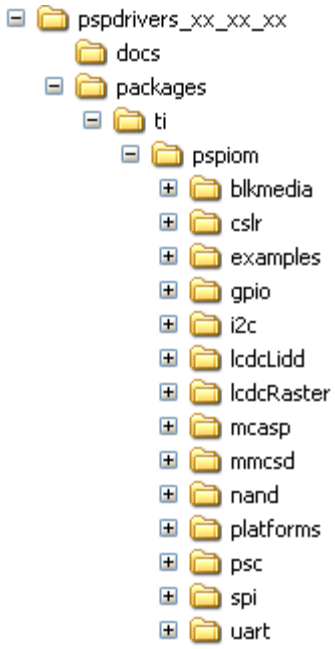


Figure 1: BIOS PSP Top level directory structure

The sections below describe the folder contents.

pspdrivers_

Contains the device drivers and other PSP components. Top level installation directory

docs

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

cslr

Contains the register level chip support for C6747 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/pspiom directory under their individual directories. For example, the UART driver is placed under ti/pspiom/Uart.

1.3.2.2 Driver Directory structure:

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

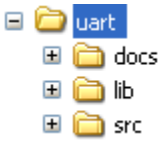


Figure 2: C6747 PSP driver directory structure

docs

Contains peripheral specifically documentation like Architecture documentation, datasheet etc.

lib

Contains generated driver library file(s)

src

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

1.3.2.3 examples Directory structure:

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

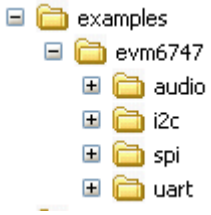
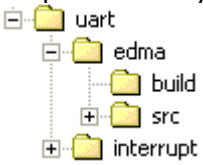


Figure 3: C6747 PSP driver sample application directory structure

evm6747

Contains the EVM/platform specific examples

Each sample example directory is further organized as shown below:



Edma (or Interrupt)

Contains specific files to demonstrate EDMA (or Interrupt) mode of operation

Build

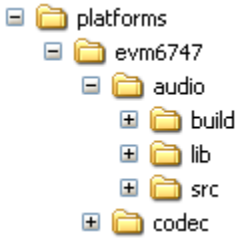
Contains CCS3 project specific files

Src

Contains the example application source files

1.3.2.4 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)

1.4 Integration Guide

This chapter discusses the **C6747 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.

1.4.1 Building the PSP Sample Applications

The PSP package contains separate sample applications for each of the BIOS drivers provided BIOS driver components (except PSC). These sample applications can be built using CCS v3.3 project files or the CCSv4 project files.

1.4.2 BIOS PSP EVM Library Module

1.4.2.1 Description

The sample applications available in the package demonstrate the usage of the BIOS PSP drivers for DSP BIOS 5.41.x on EVM 6747 platform. For successful operation of the applications, some basic initialization (ex., removing the peripheral out of sleep and powering it, 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 EVM6747, 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 evmInit.lib. 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, PSC init for the module, PINMUX settings for the module, any configuration required to be done by using the driver (like GPIO expander) etc. This folder also contains an entry in the configuration (*.tci) 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\pspiom\platforms\evmXXX and contain:

1. Platform specific initialization routines in xxx_evmInit.c
2. Platform specific init configuration files in xxx.tci
3. Platform library project file evmInit.pjt

4. Platform initialization library evmInit.lib

Note: MMCSD and NAND are not IOM based drivers, 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.

1.4.2.2 *Building the EVM library module*

Please build the

<ID>\packages\ti\pspiom\platforms\evm6747\build\ccs3\evmInit.pjt

1.5 Power Management

The drivers implement power management by means of gating respective LPSC modules. This is implemented by enabling the LPSC as long as the driver has requests/packets pending to be completed and disabling the PSC when there are no requests/packets pending to be completed. The device parameter (devParam) "**pscPwrmEnable**" should be set to TRUE, for power management to be enabled. Also the following code should be added to the tcf file to initialize the PSC module count.

```
bios.GBL.CALLUSERINITFXN = 1;
bios.GBL.USERINITFXN = prog.extern("commonInit");
```

Also, if a user wishes not to enable any power management functionality at all in the driver (default behavior), one could do so by supplying the **"pscPwrmEnable"** device/instance parameter as FALSE during device creation. In this case the PSC is enabled once during driver instantiation and disabled once during driver instance deletion.

Note:

1. DSP/BIOS™ based power management support is currently for C6748 and OMAPL138 based platform only. Only PSC power support is enabled for the C6747 platform.
2. A new instance parameter "pllDomain" has been added for power management support. This feature is not applicable for C6747 platform and any value set here does not affect the driver functionality. The legacy users shall need to recompile their applications for the changes in device parameters.
3. DSP/BIOS™ based power management is not available for C6747 and OMAPL137 platforms. It is available only for C6748 and OMAPL138 platforms. Hence, BIOS_PWRM_ENABLE must not be enabled for C6747 and OMAPL137, which would result in compilation error.

1.5.1 Building the BIOS PSP Driver Modules

BIOSPSP drivers and sample application provide support for both CCS3 and CCS4 build environments. The two build setup/project files are located in the build folder of the respective driver/sample application directories. Each of the projects are contained in ccs3 and ccs4 directories in the build folder.

Upon successful installation the BIOSPSP installer creates an environmental variable "BIOS5PSP_INSTALL_DIR" which can be used to refer to the installation directory of BIOSPSP package. This is supposed to provide for CCS3 build environments. CCS4 build environments should use the workspace and macros concept as described below.

- CCS3 build setup

Please build individual drivers using CCS v3.3 pjt files provided.

- CCS4 build setup

The project in the CCS4 build folder needs to be imported via CCS4 into a workspace. Once imported, a workspace specific macro "BIOS5PSP_INSTALL_DIR" is created for the workspace use. This is used to refer to the linked source/configuration files in the project. Since this is a relative path, this resolves into the actual installation directory once imported into the workspace.

If a user has not imported the drivers/sample application, then the install directory macro is not created in the workspace. In such a case the user needs to manually create this macro in the workspace.

Also, user may have to update the versions for DSP/BIOS™, Code generation tools etc for the workspace created.

1.5.2 BIOS drivers sample Application:

UART – The sample application demonstrates the use of the UART driver by performing reading and writing of messages and input characters from and to serial terminal of a host PC. (Tera Term or hyper terminal could be used as a serial terminal on Host PC)

I2C – The sample application demonstrates the use of the I2C driver by reading and writing data to the I2C EEPROM on the EVM

SPI - The sample application demonstrates the use of the SPI driver by writing 64 bytes of known data into serial flash, then reading back the written data and validating it.

McASP/Audio – The sample applications demonstrates the use of the McASP driver by playing an input song/tone via LINE IN. A speaker may be connected to the LINE OUT.

McASP – This sample application demonstrates the use of McASP device for data communication between two devices.

MMCSD – The sample applications demonstrates the uses of the MMCSD driver using the RAW interface by showing the usage of various IOCTLS writes to the media and verify the data written by reading it back. For using the media with File system refer to the sample application provided with the File system package.

NAND – The sample applications demonstrates the uses of the NAND driver using the RAW interface by showing the usage of various IOCTLS writes to the media and verify the data written by reading it back. For using the media with File system refer to the sample application provided with the File system package.

LCDC Raster – The sample application demonstrates the use of the LCDC Raster controller driver by displaying a static RGB stripe image, with a line scrolling on it.

LCDC LIDD – The sample application demonstrates the use of the LCDC LIDD controller driver by displaying a welcome message and doing basic operation in the display.

1.5.3 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_xx_yy_zz\packages\psp\om\cslr\docs\cslr_userguide.doc.

1.5.4 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 BIOS module GIO, to transmit and receive serial data. 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 Uart driver.

2.1.1 Key Features

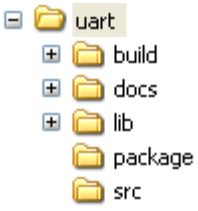
- Multi-instance support and re-entrant driver
- Each instance supports a transmit channel and a receive channel
- Supports Polled, Interrupt and DMA Interrupt Mode of operation

2.2 Installation

The UART device driver is a part of BIOSPSP product for C6747 and would be installed as part of product installation.

2.2.1 UART Component folder

On installation of BIOSPSP package for the C6747, the UART driver can be found at <ID>\ ti\pspiom\uart\



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

- **uart -** The uart folder is the place holder for the entire UART driver. This folder contains Uart.h which is the header file included by the application.
- **build –** contains CCS 3.3 / CCS 4 project file to build Uart library.
- **docs –** Contains doxygen generated API reference.
- **lib –** Contains Uart libraries
- **src –** Contains Uart driver's source code.

2.2.2 Build Options

The Uart library can be built using the CCS v3.3 project file located at <ID>\packages\ti\pspiom\uart\build\C6747\ccs3\uart.pjt. This project file supports the following build configurations.

It can also be compiled using the CCS v4 project files located at <ID>\packages\ti\pspiom\uart\build\C6747\ccs4

IMPORTANT NOTE:

All build configurations require environment variable %EDMA3LLD_BIOS5_INSTALLDIR% to be defined. This variable must point to "<EDMA3_INSTALL_DIR>\packages".

Debug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "-DUart_EDMA_ENABLE" to enable EDMA3 support in Uart driver. It also contains "-i%EDMA3LLD_BIOS5_INSTALLDIR%" to find EDMA3 header files.

iDebug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "-DUart_EDMA_ENABLE" to enable EDMA3 support in Uart driver. It also contains "-i%EDMA3LLD_BIOS5_INSTALLDIR%" to find EDMA3 header files.
- Defines "Uart_DEBUGPRINT_ENABLE to enable Uart driver to LOG debug messages.

Release:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "-DUart_EDMA_ENABLE" to enable EDMA3 support in Uart driver. It also contains "-i%EDMA3LLD_BIOS5_INSTALLDIR%" to find EDMA3 header files.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver

iRelease:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "-DUart_EDMA_ENABLE" to enable EDMA3 support in Uart driver. It also contains "-i%EDMA3LLD_BIOS5_INSTALLDIR%" to find EDMA3 header files.
- Defines "Uart_DEBUGPRINT_ENABLE to enable Uart driver to LOG debug messages.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver

2.2.2.1 Required and Optional Pre-defined symbols

The Uart library must be built with an SOC specific pre-defined symbol.

"-DCHIP_C6747" is used above to build for C6747. Internally this define is used to select a soc specific header file (soc_C6747.h). This header file contains information such as base addresses of uart devices, their event numbers, etc.

The Uart library can also be built with these optional pre-defined symbols.

Use -DUart_EDMA_ENABLE when building library to enable DMA support in Uart driver. If this symbol is not defined edma specific code will get eliminated and the driver can be used only in POLLED or INTERRUPT mode.

Use -DUart_TX_BUFFERING_ENABLE when building to enable TX buffering. This is disabled by default in the CCS 3.3 pjts provided.

Use -DPSP_DISABLE_INPUT_PARAMETER_CHECK when building library to turn OFF parameter checking. This symbol is defined for Release and iRelease profiles by default in the CCS 3.3 pjts provided.

Use -DNDEBUG when building library to turn off runtime asserts. This symbol is defined for Release and iRelease profiles by default in the CCS 3.3 pjts provided.

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 instances of UART on the EVM6747. Different instances may be specified during driver creation time, and instances 0 through 2 with corresponding device IDs 0 through 2 are supported, respectively.

These instances can operate simultaneously with configurations supported by the UART driver. UART instances are created as follows:

1. Static creation – static creation is done in the "tcf" file of the application; this creation happens at build time. The UDEV module (UDEV.create) is used during static configuration. An instance of the UDEV module at static configuration time corresponds to creating and initializing an UART instance
2. Dynamic creation – Dynamic creation of an UART instance is done in the application source files by calling DEV_createDevice(); this creation happens at runtime.

UDEV.create and DEV_createDevice allow user to specify the following:

- iomFxns: Pointer to IOM function table. UART requires this field to be Uart_IOMFXNS.
- initFxn: UART requires that the user call UART_init() as part of this initFxn. Users can also directly hook in UART_init().
- device parameters: UART requires the user to pass an `Uart_Params` struct. This struct must exist in the application source files and it must be initialized very early as part of driver specific initFxn.
- deviceId to identify the UART peripheral. This parameter decides on the instance to which this driver is binding. In case of static driver creation this parameter needs to be modified at TCF/TCI files

For more information on configuring UDEV and Uart, please refer to the Uart sample application (included with this driver release), and the DSP/BIOS API Reference (spru403o.pdf, included in your DSP/BIOS installation).

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 `GIO_create ()specify "IOM_OUTPUT" or "IOM_INPUT"`). The configuration parameters are explained in the sections to follow.

2.3.3 Support for baudrates greater than 115200

The UART driver does not impose a restriction configuring the UART peripheral for baudrates greater than 115200 baud. However, when configuring for higher baudrates, one needs to tweak the parameter rxThreshold and softTxThreshold (detailed below in Uart_Params).

2.4 Configurations

Following tables document some of the configurable parameter of UART. Please refer to Uart.h for complete configurations and explanations. Please refer the sample code as reference to change the default parameter values from the application

2.4.1 Uart_Params

This structure defines the device configurations, expected to supply while instantiating the driver known as **devParams.**

Members	Description
enableCache	Whether the submitted buffers are in cacheable memory.
fifoEnable	Whether the HW FIFO for the device is to enabled
opMode	Whether the UART driver should operate in Polled or Interrupt or DMA Interrupt Mode
loopbackEnabled	If the driver/device works 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	Should Even/Odd parity or No parity should be used
rxThreshold	FIFO data threshold for RX to raise a receive data interrupt
fc	Whether any flow control for data transfer should be used
edmaRxTC/edmaRxTC	EDMA TCs for transmit and receive
hwiNumber	The hardware interrupt number assigned for UART events
polledModeTimeout	The data transfer timeout for polled mode of operation
softTxThreshold	This is a software parameter (not a hardware setting), If this element is not equal to 1, then the number of

	bytes requested to transmit for each IO request must be multiple of this element.
pscPwrmEnable	Boolean flag to enable (TRUE) or disable (FALSE) any power management in the driver
pllDomain	Not used

Note on `softTxThreshold` **and** `rxThreshold`**:-**

In case DMA transfer mode the generation of EDMA sync event from UART to the EDMA peripheral in case of receive depends on the receive FIFO threshold level. Once the reveive FIFO threshold is reached (so many bytes received into the RXFIFO) the sync event to EDMA is generated and the EDMA transfer the bytes from the FIFO to the destination buffer depending on the transfer parameters programmed for this transfer. Similarly, for more flexibility in programming the transmit operation `softTxThreshold` is added as a device parameter above. The UART driver now programs the EDMA in AB sync mode. The B count for the EDMA transfer parameter for receive is programmed equal to the "`rxThreshold`" and the transmit B count is programmed equal to the "`softTxThreshold`". The users can tweak these parameters as required. **However, there is one limitation while setting the** `rxThreshold` **and** `softTxThreshold`**. If these are not equal to one, then the data length should be integral multiples of these values. Else, during receive remainder bytes (< rxThreshold) may not be sufficient to trigger the EDMA event and during transmit the EDMA may not pick up the remainder bytes from the buffer, since remainder bytes are not programmed at all**.

Apart from the instance parameters described above module wide constants declared in Uart.h can be changed e.g Uart_TASKLET_PRIORITY. These constants apply to all Uart instances.

Build options can also be added or removed to add/remove features. e.g –DUart_EDMA_ENABLE.

2.4.2 Uart_ChanParams

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

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, -DUart_EDMA_ENABLE must be defined

2.4.3 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.

2.4.4 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.

2.4.5 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. The driver must also be built with –DUart_EDMA_ENABLE. Also, as part of *chanParams*, the handle to the EDMA driver, hEdma, should be passed by the application.

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`

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 GIO APIs

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

2.6.1 GIO_create

Parameter Number	Parameter	Specifics to PSP
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 tcf or DEV_createDevice()
2	Channel Mode	Should be "IOM_INPUT" when UART requires to received data and "IOM_OUTPUT" when UART requires to transmit
3	Status	Address to place return status from Uart.
4	Channel Params	Pointer to chanParams structure for Uart channel.
5	GIO_Attrs *	Parameters required for the creation of the GIO instance (e.g. channel parameters)

2.6.2 GIO_control

Parameter Number	Parameter	Specifics to PSP
1	GIO_Handle	Handle returned by GIO_create
2	Command	IOCTL command defined by UART driver
3	Arguments	Misc arguments if required by the command

2.6.3 GIO_write/read

Parameter Number	Parameter	Specifics to PSP
1	Channel Handle	Handle returned by GIO_create
2	Pointer to buffer	Should be pointer to the buffer that holds data for transfer or take data in case of receive
3	Pointer to size of	Size of the transaction

	buffer	

2.7 Sources that need re-targeting

2.7.1 **ti/pspiom/cslr/soc_C6747.h (soc specific header file):**

This file contains target (SoC) specific definitions. In most cases, changing the values for the of 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.

2.8 EDMA3 Dependency

UART 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.

2.8.1.1 Used Paramset of EDMA 3

BIOSPSP UART driver uses TWO paramsets of EDMA3 per instance – one for Tx and another for Rx; 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.

2.9 Known Issues

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

2.10 Limitations

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

2.11 Uart Sample applications

2.11.1 **Interrupt mode sample**

2.11.1.1 Description:

This sample demonstrates the use of the Uart driver in interrupt mode.

The Uart driver is configured statically in uartSample.tci file. The initFxn and uartParams used in UDEV.create are globals declared in uartSample.c.

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

bios.ECM.ENABLE = 1;

bios.HWI.instance("HWI_INT8").interruptSelectNumber = 1;

These lines configure the ECM module and map Uart events to CPU interrupts. For example the Uart event number is 38 which falls in ECM group 1. Here ECM group 1 is mapped to HWI_INT8.

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 GIO APIS to create uart channels amd read and write to them.

The user_uart0_init() calls Uart_init() and initializes the Uart_Params structure.

2.11.1.2 Build:

This sample can be built using

<ID>/packages/ti/pspiom/examples/evm6747/uart/interrupt/build/ccs3/uartSample.pjt

IMPORTANT NOTE: uartSample.pjt contains references to %EDMA3LLD_BIOS5_INSTALLDIR% environment variable and links with edma3 libraries. This is required because by default the Uart driver library is built with –DUart_EDMA_ENABLE. The user can remove all references of EDMA3 from uartSample.pjt if he re-builds the Uart library without –DUart_EDMA_ENABLE.

2.11.1.3 Setup:

You need to connect a NULL Model cable from the evm6747 platform to a host PC. On the host an application like HyperTerminal needs to be setup for appropriate COM port, baud rate etc.

2.11.1.4 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. The user could make use of the sample.txt file provided with the package at ti\pspiom\examples\evm6747\uart\<edma/interrupt>. This file contains 1000 characters of data
- This sample application will echo the received characters to the terminal.

2.11.2 Dma mode sample

2.11.2.1 Description:

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

The Uart driver is configured statically in uartSample.tci file. This file can be directly imported into an application's tcf script. The initFxn and uartParams used in UDEV.create are globals declared in uartSample.c.

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

```
bios.ECM.ENABLE = 1;
bios.HWI.instance("HWI_INT8").interruptSelectNumber = 1;
```

These lines configure the ECM module and map Uart events to CPU interrupts. For example the Uart event number is 38 which falls in ECM group 1. Here ECM group 1 is mapped to HWI_INT8.

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 GIO APIS to create uart channels and reads and writes to them.

The user_uart0_init() calls Uart_init() and initializes the Uart_Params structure. It also calls edma3init() which initializes the EDMA3 driver and sets up hEdma.

2.11.2.2 Build:

This sample can be built using

<ID>/packages/ti/pspiom/examples/evm6747/uart/edma/build/ccs3/uartSample.pjt

IMPORTANT NOTE: uartSample.pjt assumes that the Uart driver library is built with –DUart_EDMA_ENABLE.

2.11.2.3 Setup:

You need to connect a NULL Modem cable from the evm6747 platform to a host PC. On the host an application like HyperTerminal needs to be setup for appropriate COM port, baud rate etc.

2.11.2.4 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. The user could make use of the sample.txt file provided with the package at ti\pspiom\examples\evm6747\uart\<edma/interrupt>. This file contains 1000 characters of data
- This sample application will echo the received characters to the terminal.

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 the GIO layer, in order to transmit and receive serial data. The following sections describe in detail the necessary procedures to configure and use this driver, as well as other additional information. 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

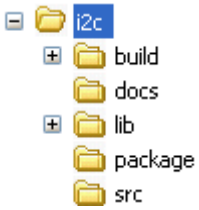
- Multi instantiable and re-entrant driver
- Each instance can operate as an receiver and/or transmitter
- Supports Polled, Interrupt and DMA Interrupt Mode of operation

3.2 Installation

The I2c device driver is a part of the PSP package for the C6747 and is 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\pspiom\i2c\docs)

3.2.1 I2C Component folder

On installation of PSP package for the C6747, the I2C driver can be found at <ID>\ ti\pspiom\i2c\



As show above, the i2c folder contains several sub-folders, the 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. This folder contains I2c.h, which is the header file included by the application.
- **build -** contains CCS 3.3 / CCS 4 project files to build the I2c library.
- **docs –** Contains doxygen generated API reference.
- **src –** Contains the I2C driver's source code.

3.2.2 Build Options

The I2c library can be built using the CCS v3.3 project file located at <ID>\packages\ti\pspiom\i2c\build\C6747\ccs3\i2c.pjt. This project file supports the following build configurations.

IMPORTANT NOTE:

All build configurations require environment variable %EDMA3LLD_BIOS5_INSTALLDIR% to be defined. This variable must point to "<EDMA3_INSTALL_DIR>\packages".

Debug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "-DI2c_EDMA_ENABLE" to enable EDMA3 support in I2c driver. It also contains "-i%EDMA3LLD_BIOS5_INSTALLDIR%" to find EDMA3 header files.

iDebug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "-DI2c_EDMA_ENABLE" to enable EDMA3 support in I2c driver. It also contains "-i%EDMA3LLD_BIOS5_INSTALLDIR%" to find EDMA3 header files.
- Defines "I2c_DEBUGPRINT_ENABLE to enable I2c driver to LOG debug messages.

Release:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "-DI2c_EDMA_ENABLE" to enable EDMA3 support in I2c driver. It also contains "-i%EDMA3LLD_BIOS5_INSTALLDIR%" to find EDMA3 header files.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver

iRelease:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "-DI2c_EDMA_ENABLE" to enable EDMA3 support in I2c driver. It also contains "-i%EDMA3LLD_BIOS5_INSTALLDIR%" to find EDMA3 header files.
- Defines "I2c_DEBUGPRINT_ENABLE to enable I2c driver to LOG debug messages.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver

3.2.2.1 Required and Optional Pre-defined symbols

The I2c library must be built with a soc specific pre-defined symbol.

"-DCHIP_C6747" is used above to build for C6747. Internally this define is used to select a soc specific header file (soc_C6747.h). This header file contains information such as base addresses of I2C devices, their event numbers, etc.

The I2c library can also be built with these optional pre-defined symbols.

Use –DI2c_EDMA_ENABLE when building library to enable DMA support in I2c driver. If this symbol is not defined edma specific code will get eliminated and the driver can be used only in POLLED or INTERRUPT mode.

Use -DPSP_DISABLE_INPUT_PARAMETER_CHECK when building library to turn OFF parameter checking. This symbol is defined for Release and iRelease profiles by default in the CCS 3.3 pjts provided.

Use -DNDEBUG when building library to turn off runtime asserts. This symbol is defined for Release and iRelease profiles by default in the CCS 3.3 pjts provided.

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 instances of I2C on the EVM6747. Different instances may be specified during driver creation time, and instances 0 through 2 with corresponding device IDs 0 through 2 are supported, respectively.

These instances can operate simultaneously with configurations supported by the I2C driver. I2C instances are created as follows:

1. Static creation – static creation is done in the "tcf" file of the application; this creation happens at build time. The UDEV module (UDEV.create) is used during static configuration. An instance of the UDEV module at static configuration time corresponds to creating and initializing an I2C instance
2. Dynamic creation – Dynamic creation of an I2C instance is done in the application source files by calling DEV_createDevice(); this creation happens at runtime.

UDEV.create and DEV_createDevice allow user to specify the following:

- iomFxns: Pointer to IOM function table. I2C requires this field to be I2c_IOMFXNS.
- initFxn: I2C requires that the user call I2c_init() as part of this initFxn. Users can also directly hook in I2c_init().
- device parameters: I2C requires the user to pass an `I2c_Params` struct. This struct must exist in the application source files and it must be initialized very early as part of driver specific initFxn.
- deviceId to identify the I2C peripheral.

For more information on configuring UDEV and I2c, please refer to the I2c sample application (included with this driver release), and the DSP/BIOS API Reference (spru403o.pdf, included in your DSP/BIOS installation).

3.3.2 Each Instance as Transmitter and/or receiver

I2C driver can be simultaneously operated as a transmitter and receiver. This could be achieved by opening a GIO Channel as an INPUT channel and opening another GIO Channel as an OUTPUT channel. The type of Channel is specified while creating the channel (using `GIO_create()` and specifying `"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 for complete configurations and explanations.

3.4.1 I2c_Params

This structure defines the device configurations, expected to supply while instantiating the driver known as `devParams`.

Members	Description
enableCache	Whether or not the submitted buffers are in cacheable memory.
opMode	Whether the I2C driver should operate in Polled or Interrupt or DMA Interrupt Mode
ownAddr	The slave address of the device application is addressing
loopbackEnabled	Enable or Disable digital loop back mode
numBits	The number of data bits
busFreq	The frequency at which the clock (SCL) is operating
addressing	Whether 7 bit addressing or extended (10-bit) addressing mode is used
edma3EventQueue	The EDMA event queue the application will use in DMA Interrupt mode of operation mode
hwiNumber	The hardware interrupt number assigned for I2C events
polledModeTimeout	The data transfer timeout for polled mode of operation
pscPwrmEnable	Boolean flag to enable (TRUE) or disable (FALSE) any power management in the driver
pllDomain	Not used

Note: I2C address does not allow addressing "self". That is any requests with slave address as own address is not permitted, and such submit requests raise an error.

Apart from the instance parameters described above module wide constants declared in I2c.h can be changed e.g `I2c_peripheralClkFreq`. These constants apply to all I2c instances.

Build options can also be added or removed to add/remove features. e.g – DI2c_EDMA_ENABLE.

3.4.2 I2c_ChanParams

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

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, the necessary define switch – DI2c_EDMA_ENABLE should be thrown, as described in section 3.2.2 "Build Options".
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

3.4.3 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.

3.4.4 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.

3.4.5 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. Also, as part of chanParams, the handle to the EDMA driver, hEdma, should be passed by the application.

Note that -DI2c_EDMA_ENABLE define 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.

3.4.6 Slave mode

This version of I2C driver supports slave mode and to use this driver in I2C slave mode

a) masterOrSlave flag in chanparams to select slave mode.

b) use I2c_MASTER flag in the DataParam->flags during the IO submits

Please note the following

- Only one channel is allowed to be open in Slave mode.
- I2C driver does not support slave mode of operation in polled mode. Only interrupt and DMA interrupt mode of operation are supported. The slave mode of operation is tested successfully 100,200 and 400 kHz I2C clock frequency.
- (a) I2C slave application need to take care of the data (application level) protocol on when and what to receive and send by/from slave side. (b)This driver provides a generic bus communication path for slave. (c) Application protocol also needs to consider the latency caused by software slave implementation. (d) The driver does not support "0" no of byte transfer and the slave driver would not function properly if master issues a STOP condition immediately after a START condition.

- In receive mode, the current IOP is completed when an SCD is detected. However, when the receive buffer has exhausted, the receiver sends an NACK to the master. This is done to prevent the call to driver from the application from blocking indefinitely.
- In transmit mode, the current IOP is completed when the transmit buffer is exhausted, or an SCD is detected (generated) on the bus. However, when the transmit buffer has exhausted, though the IOP is completed, dummy bytes are transferred. This is done to prevent the call to the driver from the application from blocking indefinitely.

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 GIO_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 is 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`.

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 GIO calls in the context of PSP driver. Note that no effort is made to document the use of GIO calls; any PSP specific requirements are covered below.

3.6.1 GIO_create

Parameter Number	Parameter	Specifics to PSP
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 TCF or DEV_createDevice ()
2	Channel Mode	Should be "IOM_INPUT" when I2C requires to received data and "IOM_OUTPUT" when I2C requires to transmit
3	GIO_Attrs *	Parameters required for the creation of the GIO instance (e.g. channel parameters)

3.6.2 GIO_control

Parameter Number	Parameter	Specifics to PSP
1	GIO_handle	Handle returned by GIO_create
2	Command	IOCTL command defined by I2C driver
3	Arguments	Misc arguments if required by the command

3.6.3 GIO_write/read

Parameter Number	Parameter	Specifics to PSP
1	Channel Handle	Handle returned by GIO_create
2	Pointer to buffer	Should be pointer to variable of type PSP_I2c_PktAddrPayload OR Uint32 * that holds the audio data.

3	Size	Size of the transaction

3.7 Sources that need re-targeting

3.7.1 ti/pspiom/cslr/soc_C6747.h (soc specific header file):

This file contains target (SoC) specific definitions. In most cases, changing the values for the of 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.

3.8 EDMA3 Dependency

I2C 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.

3.8.1 Used Paramset of EDMA 3

PSP driver uses TWO paramsets of EDMA3 per instance, one for Tx and another for Rx; if there are no paramsets available the PSP driver creation would fail. These paramsets are used through the lifetime of PSP 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 Limitations

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

3.11 I2c Sample applications

3.11.1 Interrupt mode sample

3.11.1.1 Description:

This sample demonstrates the use of the I2c driver in interrupt mode.

This example uses the I2c bus to write an array of data to the CAT24WC256 EEPROM memory of the evm6747. 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 GIO modules, in combination. The I2c driver is used to configure and set up the I2c bus, and the GPIO module APIs are used to perform the actual reads and writes to the EEPROM memory, via the I2c bus.

The I2c driver is configured both statically in the i2cSample.tci and i2cSample.tcf files, as well as at run time in the i2cSample_main.c and i2cSample_io.c files.

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

```
bios.ECM.ENABLE = 1;
bios.HWI.instance("HWI_INT8").interruptSelectNumber  = 1;
```

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.tci file, which uses the UDEV module to configure the user defined init function "user_i2c_init", and also hook in the I2c instance parameters (i2cParams).

At run time, this results in the I2c user defined init function to be called before the main() function. This function in turn calls the actual I2c_init() function (a requirement if a user defined init function is used), and then sets up the user's I2c instance parameters via "i2cParams".

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

3.11.1.2 Build:

This sample can be built using

<ID>/packages/ti/pspiom/examples/evm6747/i2c/interrupt/build/ccs3/i2cSample.pjt

IMPORTANT NOTE: i2cSample.pjt contains references to %EDMA3LLD_BIOS5_INSTALLDIR% environment variable and links with edma3 libraries. This is required because by default the I2c driver library is built with –DI2c_EDMA_ENABLE. The user can remove all references of EDMA3 from i2cSample.pjt if he re-builds the I2c library without –DI2c_EDMA_ENABLE.

3.11.1.3 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.

3.11.1.4 Output:

When the sample runs, it will output the following:

```
I2C  :Start of I2C sample application
```

```
GIO_create(outHandle) returned status = 0

GIO_create(inHandle) returned status = 0

I2C CAT24WC256 EEPROM write/read test started

I2C CAT24WC256 EEPROM Read/write test passed

I2C  :End of I2C sample application


!!! PSP HrtBt

!!! PSP HrtBt

.............
```

3.11.2 DMA Interrupt mode sample

3.11.2.1 Description:

This sample demonstrates the use of the I2c driver in EDMA mode. In EDMA mode, the I2c driver uses DMA for data transfers, instead of the CPU.

This example uses the I2c bus to write an array of data to the CAT24WC256 EEPROM memory of the evm6747. 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 GIO modules, in combination. The I2c driver is used to configure and set up the I2c bus, and the GIO module APIs are used to perform the actual reads and writes to the EEPROM memory, via the I2c bus.

The I2c driver is configured both statically in the i2cSample.tci and i2cSample.tcf files, as well as at run time in the i2cSample_main.c and i2cSample_io.c files.

The i2cSample.tcf file contains important BIOS configuration settings, which are required in order for the I2c operations to work properly. The most important lines in this file which the user would need in their application are:

```
bios.ECM.ENABLE = 1;
bios.HWI.instance("HWI_INT7").interruptSelectNumber = 0;
bios.HWI.instance("HWI_INT8").interruptSelectNumber = 1;
bios.HWI.instance("HWI_INT9").interruptSelectNumber = 2;
bios.HWI.instance("HWI_INT10").interruptSelectNumber = 3;
```

The above configuration settings are needed to correctly set up the ECM module and map the EDMA events to CPU interrupts. Since the CPU is not used in I2c transfers in EDMA mode, these ECM groups must be mapped to the EDMA events as shown.

Further I2c static configuration is done in the i2cSample.tci file, which uses the UDEV module to configure the user defined init function "user_i2c_init", and also hook in the I2c instance parameters (i2cParams).

At run time, this results in the I2c user defined init function to be called before the main() function. This function in turn calls the actual I2c_init() function (a requirement if a user defined init function is used), and then sets up the user's I2c instance parameters via "i2cParams".

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

3.11.2.2 Build:

This sample can be built using

<ID>/packages/ti/pspiom/examples/evm6747/i2c/edma/build/ccs3/i2cSample.pjt

IMPORTANT NOTE: i2cSample.pjt assumes that the I2c driver library is built with –DI2c_EDMA_ENABLE.

3.11.2.3 Setup:

No special setup is needed to run the I2c example

3.11.2.4 Output:

When the sample runs, it will output the following:

```
EDMA3 : edma3init() passed

I2C  :Start of I2C sample application

GIO_create(outHandle) returned status = 0

GIO_create(inHandle) returned status = 0

I2C CAT24WC256 EEPROM write/read test started

I2C CAT24WC256 EEPROM Read/write test passed

I2C  :End of I2C sample application


!!! PSP HrtBt

!!! PSP HrtBt

.............
```

4 GPIO driver

4.1 Introduction

This section is the reference guide for the GPIO device driver which explains the features and tips on how to use it.

DSP/BIOS applications use the driver typically through APIs provided by the GPIO driver itself, in order to communicate with the GPIO hardware (the GPIO driver does not follow the DSP/BIOS IOM model). The GPIO driver provides a set of basic APIs which may be used to read or write to the GPIO pins or banks, configure/register interrupts and corresponding interrupt service routines, configure rising or falling edge triggers and more.

This driver does not support any data transfer protocol; the user is expected to write that protocol as a wrapper around the GPIO APIs provided, if needed.

The following sections describe in detail the necessary procedures to configure and use this driver, as well as other additional information. It is recommended to go through the sample application to get a feel of initializing and using the GPIO driver.

4.1.1 Key Features

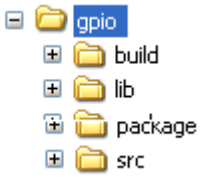
- Setting GPIO pin directions
- Marking pins or banks as available for use
- Enabling and Disabling of bank interrupts
- Registering interrupt handlers for a pin or bank interrupt
- Getting or setting a group of pins to a value

4.2 Installation

The Gpio device driver is a part of the PSP package for the C6747 and is 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\pspiom\gpio\docs)

4.2.1 Gpio Component folder

Upon installation of the PSP package for the C6747, the Gpio driver can be found at <ID>\ ti\pspiom\gpio\



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

- **gpio -** The gpio folder is the place holder for the entire Gpio driver source and the build configuration files. This folder contains Gpio.h, which is the header file included by the application.
- **build -** contains CCS 3.3 / CCS 4 project files to build the Gpio library**.**
- **src –** Contains the Gpio driver's source code.

4.2.2 Build Options

The Gpio library can be built using the CCS v3.3 project file located at <ID>\packages\ti\pspiom\gpio\build\C6747\ccs3\gpio.pjt. This project file supports the following build configurations.

Debug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.

iDebug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "Gpio_DEBUGPRINT_ENABLE to enable Gpio driver to LOG debug messages.

Release:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver

iRelease:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "Gpio_DEBUGPRINT_ENABLE to enable Gpio driver to LOG debug messages.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver

4.2.2.1 Required and Optional Pre-defined symbols

The Gpio library must be built with a soc specific pre-defined symbol.

"-DCHIP_C6747" is used above to build for C6747. Internally this define is used to select a soc specific header file (soc_C6747.h). This header file contains information such as base addresses of Gpio devices, their event numbers, etc.

If this define is missing, the following compile error will be thrown:

"No chip type defined! (Must use -DCHIP_C6747 or -DCHIP_C6747)"

The Gpio library can also be built with these optional pre-defined symbols.

Use -DPSP_DISABLE_INPUT_PARAMETER_CHECK when building library to turn OFF parameter checking. This symbol is defined for Release and iRelease profiles by default in the CCS 3.3 pjts provided.

Use -DNDEBUG when building library to turn off runtime asserts. This symbol is defined for Release and iRelease profiles by default in the CCS 3.3 pjts provided.

4.3 Features

This section details the features of Gpio and how to use it in detail.

4.3.1 Single-Instance Usage

The Gpio driver can operate on all the Gpio banks and pins on the EVM6747. Only one Gpio driver instance is currently supported by the Gpio driver module. Through this instance, the user may specify bank and pin parameter settings as desired. This single Gpio instance uses device ID 0.

Once configured and set up properly, the user may perform operations on the Gpio banks and pins using the Gpio APIs provided by the Gpio module.

The Gpio driver is not an IOM driver, and therefore it is not necessary to make any static configuration settings for UDEV, as is needed in the other drivers (e.g. Uart). However, it is necessary to configure the HWI interrupt select numbers properly in the BIOS configuration.

The following steps provide an overview of how to use the Gpio driver; it is recommended that the user follow the Gpio example in tandem with these steps. The first step must be done in the BIOS configuration file; all steps that follow must be done in C code:

1. In the *.tcf file, set up HWI interrupt source numbers:

```
bios.HWI.instance("HWI_INT7").interruptSelectNumber  = 0;
bios.HWI.instance("HWI_INT8").interruptSelectNumber  = 1;
bios.HWI.instance("HWI_INT9").interruptSelectNumber  = 2;
bios.HWI.instance("HWI_INT10").interruptSelectNumber = 3;
```

2. In the C file, declare a `Gpio_Handle` variable:

```
Gpio_Handle gpioHandle;
```

   `gpioHandle` will be used later in the program to reference the Gpio instance that exists as part of the driver.

3. Create a struct of type `Gpio_Params`:

```
Gpio_Params params = Gpio_PARAMS;
```

   setting its value to `Gpio_PARAMS` initializes it to the default parameter values.

4. Use the params struct created in the previous step to configure pins and banks as needed. For example:

```
/* set instance number to be 0 */
params.instNum = 0;
```

```
/* specify the bank we want to use as unavailable */
params.BankParams[2].inUse = Gpio_InUse_No;

/* specify the HWI associated with this bank */
params.BankParams[2].hwiNum  = 9;

/* specify the pin we want to use within this bank as
unavailable */
params.BankParams[2].PinConfInfo[5].inUse = Gpio_InUse_No;
```

5. Call `Gpio_open()` to get a handle to the Gpio instance:

```
gpioHandle = Gpio_open(&params);
```

6. Wake up the Gpio module (refer to section 7.4 "Use of PSC driver through module APIs" for more information):

```
status = Psc_ModuleClkCtrl(Psc_DevId_1, GPIO_LPSC_NUM, TRUE);
```

7. Make calls to Gpio APIs as desired, using `gpioHandle`. For example:

```
status = Gpio_setRisingEdgeTrigger(gpioHandle, 5);
/*
 *  make other Gpio API calls here, such as registering an
 *  interrupt handler for a particular bank, etc.
 */
```

8. Close the instance handle (optional)

```
Gpio_close(gpioHandle);
```

For more information on configuring and using Gpio, please refer to the Gpio sample application, and the doxygen documentation for Gpio (included with this driver release).

4.4 Configurations

Following tables document some of the configurable parameters of Gpio. Please refer to the doxygen documentation or Gpio.h for complete configurations and explanations.

4.4.1 Gpio_Params

This structure is used to define the user's desired configuration settings for the Gpio instance. It contains the instance number and the array of bank configuration settings for the Gpio instance. The user is expected to supply an instance of this struct when calling `Gpio_open()`.

Members	Description
instNum	The Gpio instance to configure. Currently must be 0.
BankParams[]	An array which represents the configuration settings for the array of Gpio banks existing on the device.

4.4.2 Gpio_BankConfig

Structure representing the configuration settings for a particular bank in the Gpio instance. The `Gpio_Params` structure contains an array of type `Gpio_BankConfig`, through which the user can update to configure bank settings.

Members	Description
`PinConfInfo[]`	Array which represents the configuration settings for the set of pins for this bank.
`hwiNum`	The hardware interrupt number that is assigned to the event associated with this bank.
`inUse`	Used to specify the availability of this bank. Default is Gpio_InUse_Yes (available).

4.4.3 Gpio_PinConfig

Structure representing the settings for an individual pin. The `Gpio_Params` structure contains an array of type `Gpio_BankConfig`, and each of those elements in turn contains an array of type `Gpio_PinConfig`. Through this indirection, the user can configure pin settings for a particular bank. (Please refer to the example code or section 5.3.1 step 4 in this document to see how this works).

Members	Description
`inUse`	Used to specify the availability of this pin. Default is Gpio_InUse_Yes (available).
`hwiNum`	The hardware interrupt number that is assigned to the event associated with this pin.

4.4.4 Gpio_InUse (enumeration type)

This enumeration is used frequently within the `Gpio_Params` and related configuration structs. Its enumeration values are used when specifying whether or not a bank or pin is available for use.

`Gpio_InUse_Yes` – specifies that the bank or pin *is available* to be used.

`Gpio_InUse_No` – specifies that the bank or pin *is not available* for use.

4.5 Gpio Bank Event Numbers

The following event numbers are configured for the 8 Gpio banks on the EVM6747. This table should be used when configuring the HWI interrupt select numbers and HWI number for a given bank that the user wishes to use. Please refer to the ECM module in the *DSP/BIOS 5.xx Application Programming Interface (API) Reference Guide*:

Bank Number	Event Number
0	65
1	41
2	49
3	52
4	54
5	59
6	62
7	72

4.6 Sources that need re-targeting

4.6.1 ti/pspiom/cslr/soc_C6747.h (soc specific header file):

This file contains target (SoC) specific definitions. In most cases, changing the values for the of 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.

4.7 Known Issues

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

4.8 Limitations

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

4.9 Gpio Sample application

4.9.1 Description:

This sample demonstrates the use of the Gpio driver.

The sample does this by executing in a while loop, awaiting input on GPIO pin GPIO0_8 (Gpio bank 0, pin 8, configured as an input pin), which is hooked up to switch SW3-1. This GPIO pin is configured to trigger an interrupt at the rising edge. When the switch SW3-1 is toggled, the ISR handler for this interrupt, `GPIO_input_isr()` is called. This ISR sets the global variable `status` to 1, which is the condition variable for the while loop described previously. The while loop then breaks and and causes led DS1 to blink. This led is configured to be connected to the GPIO0_12 (Gpio bank 0, pin 12, configured as output pin).

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

```
bios.ECM.ENABLE = 1;
bios.HWI.instance("HWI_INT9").interruptSelectNumber  = 2;
```

The above configuration settings are needed to correctly set up the ECM module and map the Gpio bank 0 event number to the correct CPU interrupt number. For example, the Gpio event number for bank 0 is 65, which falls under ECM group 2. Here ECM group 2 is mapped to HWI_INT9, and this is the HWI number used when configuring gpioParams at runtime.

At run time, this results in the Gpio user defined ISR function `GPIO_input_isr()` to be called once the switch SW3-1 is toggled.

When the user configures their own application, they should hook up an HWI, ISR, and ECM in a similar manner.

4.9.1.1 Build:

This sample can be built using

<ID>/packages/ti/pspiom/examples/evm6747/gpio/build/ccs3/gpioSample.pjt

4.9.1.2 Setup:

No special setup is needed to run the Gpio example

4.9.1.3 Output:

When the sample runs, it will output similar to the following. Note that the user must toggle switch SW3-1 in order to see the final two lines of the program output:

```
0  Waiting for GPIO Interrupt
1  Waiting for user to toggle SW3-1
2  Waiting for user to toggle SW3-1
3  Waiting for user to toggle SW3-1
4  Waiting for user to toggle SW3-1
5  Waiting for user to toggle SW3-1
6  Waiting for user to toggle SW3-1
7  Waiting for user to toggle SW3-1
8  Waiting for user to toggle SW3-1
9  Waiting for user to toggle SW3-1
 GPIO Interrupt occured !
11  End of GPIO sample application!
```

5 LCDC Raster Controller Driver

5.1 Introduction

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

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

5.1.1 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.

5.1.2 References

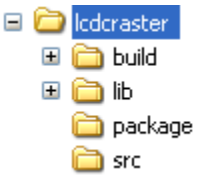
1	SPRUFM0	TMS320C6747 DSP LCD Controller User's Guide

5.2 Installation

The LCDC Raster device driver is a part of PSP package for C6747 platform and is installed as part of whole package installation.

5.2.1 LCDC Raster Component folder

On installation of PSP package for the C6747, the LCDC Raster Controller driver can be found at <ID>\ ti\pspiom\lcdcraster\



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

- **lcdcraster -** The lcdcraster folder is the place holder for the entire lcdcraster driver source and the build configuration files. LCDC Raster driver is implemented as an IOM driver under DSP/BIOS™ operating system. SIO defined APIs can be used to interface to LCDC Raster driver. This folder contains the build configuration file (package.bld), the LCDC Raster header file that's included by the application (Raster.h).
- **build -** contains CCS 3.3 / CCS 4 project files to build the LCDC Raster library.
- **lib –** contains the LCDC Raster libraries.
- **src –** Place holder for LCDC Raster driver's source code.

5.2.2 Build Options

The LCDC Raster library can be built using the CCS v3.3 project file located at <ID>\packages\ti\pspiom\lcdcraster\build\C6747\ccs3\lcdcraster.pjt. This project file supports the following build configurations.

Debug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.

iDebug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "Raster_DEBUGPRINT_ENABLE to enable Raster driver to LOG debug messages.

Release:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver

iRelease:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver
- Defines "Raster_DEBUGPRINT_ENABLE to enable Raster driver to LOG debug messages.

5.2.2.1 Required and Optional Pre-defined symbols

The LCDC Raster library must be built with a soc specific pre-defined symbol.

"-DCHIP_C6747" is used above to build for C6747. Internally this define is used to select a soc specific header file (soc_C6747.h). This header file contains information such as base addresses of LCDC devices, their interrupt numbers, etc.

If this define is missing, the following compile error will be thrown:

"No chip type defined! (Must use -DCHIP_C6747 or -DCHIP_C6747)"

The LCDC Raster library can also be built with these optional pre-defined symbols.

Use -DPSP_DISABLE_INPUT_PARAMETER_CHECK when building library to turn OFF parameter checking. This symbol is defined for Release and iRelease profiles by default in the CCS 3.3 pjts provided.

Use -DNDEBUG when building library to turn off runtime asserts. This symbol is defined for Release and iRelease profiles by default in the CCS 3.3 pjts provided.

5.3 Features

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

5.3.1 Multi-Instance Usage

The Raster driver can be used to operate the LCDC Controller in Raster mode on the C6747. Currently, only one driver instance for LCDC Raster is supported during driver creation time for the C6747. This is because there is only one LCDC Raster controller on the hardware. However, the driver is written in such a way as to support multiple instances for when new SOCs are added which do have multiple controllers. A LCDC Raster driver instance for the C6747 should use a single instance with device ID 0.

A LCDC Raster instance can be operated with configurations supported by Raster driver. The device ID can be specified using the `deviceId` field of a UDEV instance (however, only deviceId = 0 is supported for the C6747).

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

1. Static creation – static creation is done in the "tcf" file of the application; this creation happens at build time. It's necessary to configure LCDC Raster using two modules:
   a. The UDEV module (UDEV.create) is used during static configuration. An instance of the UDEV module at static configuration time corresponds to creating and initializing an LCDC Raster instance.
   b. It is also necessary to create an instance of the class driver DIO. This DIO instance is needed in order to write to the LCDC Raster controller using the SIO module at run time. It's necessary to hook the UDEV instance that was created into this DIO instance via the DIO instance property `deviceName`. Additionally, a Raster_ChanParams struct (which must be defined in the application's C code) must be set using the DIO instance property `chanParams`.
2. Dynamic creation – Dynamic creation of an LCDC Raster instance is done in the application source files by calling DEV_createDevice(); this creation happens at runtime. However, it is still necessary to configure the DIO instance statically, as described in part 1.b above.

UDEV.create and DEV_createDevice allow user to specify the following:

- iomFxns: Pointer to IOM function table. Raster requires this field to be Raster_IOMFXNS.
- initFxn: LCDC Raster requires that the user call Raster_init() as part of this initFxn. Users can also directly hook in Raster_init().
- device parameters: LCDC Raster requires the user to pass an `Raster_Params` struct. This struct must exist in the application source files and it must be initialized very early as part of driver specific initFxn.
- deviceId to identify the LCDC Raster peripheral.

For more information on configuring UDEV, DIO and LCDC Raster, please refer to the LCDC Raster sample application (included with this driver release), and the DSP/BIOS API Reference (spru403o.pdf, included in your DSP/BIOS installation).

5.3.2 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.

5.4 Configurations

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

5.4.1 Device Parameters

This structure defines the device instance configuration, which should be supplied while instantiating the driver.

Raster_Params

Serial Number	Parameter	Description
1	devConf	The device configuration provided as a Raster_DeviceConf structure

5.4.1.1 Raster_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 Raster_DmaConfig structure
5	`pscPwrmEnable`	Boolean flag to enable (TRUE) or disable (FALSE) any power management in the driver

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 C6747. 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 *Raster_DeviceConf* structure and as described below via *Raster_DmaConfig* structure. For further details refer to TMS320C6747 DSP LCD Controller User's Guide .

5.4.1.2 Internal DMA Configuration

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

Raster_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 endian mode
4	eofInt	To enable End Of Frame interrupts

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

5.4.2 Channel Parameters

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

Raster_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 Raster_RasterConf
3	segId	The MEM segment ID to be used if the driver is 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 the heap. The heap from which the allocation is made should be defined by the application. In result, the application should create a heap using the DSP/BIOS MEM manager, and pass the segment ID for this heap via `segId`. In case the `segId` is NULL and the application requests for allocation, then the driver tries to allocate the frame buffer from the default heap of the system. 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 in this case no dynamic allocation happens in the driver.

5.4.2.1 Raster controller configuration

Raster_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

5.5 Control Commands

The following are some of the important control commands for the raster controller driver:

Command	Arguments	Description
Raster_IOCTL_GET_DEVICE_CONF	Pointer to Raster_DeviceConf structure	To get the current device configuration
Raster_IOCTL_GET_RASTER_CONF	Pointer to Raster_RasterConf structure	To get the current raster configuration
Raster_IOCTL_GET_RASTER_SUBPANEL_CONF	Pointer to Raster_RasterSubpanel structure	To get the current raster sub panel configuration
Raster_IOCTL_SET_RASTER_SUBPANEL_EN	Pointer to Void	If boolean is true then enables subpanel, else disables subpanel
Raster_IOCTL_SET_RASTER_SUBPANEL_POS	Pointer to Void	To configure the position of the raster subpanel
Raster_IOCTL_SET_RASTER_SUBPANEL_LPPT	Pointer to Void	To configure the number of lines to be refreshed in the subPanel
Raster_IOCTL_SET_RASTER_SUBPANEL_DATA	Pointer to Void	To configure the default pixel data outside the subPanel
Raster_IOCTL_GET_DMA_CONF	Pointer to Raster_DmaConfig structure	To get the current DMA configuration setting
Raster_IOCTL_SET_DMA_FB_MODE	Pointer to Void	To set the frame buffer mode for the

Raster_IOCTL_SET_DMA_BURST_SIZE	Pointer to Void	To set the DMA burst size
Raster_IOCTL_SET_DMA_EOF_INT	Pointer to Void	To enable/disable the end-of-frame interrupt
Raster_IOCTL_ADD_RASTER_EVENT	Pointer to Uint32 variable containing the interrupt mask	To enable a specific event interrupt enable
Raster_IOCTL_REM_RASTER_EVENT	Pointer to Uint32 variable containing interrupt mask	To disable a specific event interrupt disable
Raster_IOCTL_GET_EVENT_STAT	Pointer to Raster_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 Uint32 variable	To get the current version of the controller
Raster_IOCTL_ALLOC_FB	Pointer to a Raster_FrameBuffer	To allocate a frame buffer on application's behalf
Raster_IOCTL_FREE_FB	Pointer to a Raster_FrameBuffer	To de-allocate a frame buffer in application's behalf

5.6 Use of RASTER driver through SIO APIs

5.6.1 SIO_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 DIO instance in the "tcf" file.
2	IO mode	Should be "SIO_OUTPUT"
3	size_t buffersize	Size of stream buffer.

4	SIO_Attrs *attrs	Pointer to the parameters structure. Should set: • `attrs.model = SIO_ISSUERECLAIM;`

5.6.2 SIO_ctrl

Parameter Number	Parameter	Specifics to Raster
1	SIO_Handle stream	Handle returned by `SIO_create`
2	Uns cmd	IOCTL command defined by LCDC Raster driver
3	Arg arg	Misc arguments if required by the command

5.6.3 SIO_issue

Parameter Number	Parameter	Specifics to Raster
1	SIO_Handle stream	Handle returned by `SIO_create`
3	Pointer to buffer	Should be pointer to framebuffer of type
4	Size	Size of the transaction in MADUs
5	Arg arg	User argument

5.6.4 SIO_reclaim

Parameter Number	Parameter	Specifics to Raster
1	SIO_Handle stream	Handle returned by `SIO_create`
3	Pointer to buffer	pointer to buffer
4	Size	Size of the transaction
5	Arg *arg	Pointer to user argument

5.7 Sources that need re-targeting

5.7.1 ti/pspiom/cslr/soc_C6747.h (soc specific header file):

This file contains target (SoC) specific definitions. In most cases, changing the values for the of 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.

5.8 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 parameter sets.

5.9 Known Issues

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

5.10 Limitations

- The LCDC controller on C6747 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.

5.11 Raster Sample Application

5.11.1.1 Description:

This sample demonstrates the use of the LCDC Raster driver.

The rasterSample.tcf 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. It also creates a task for the function *'rasterSampleTask()'*, which runs the sample application.

In particular, the rasterSample.tcf file contains the following important BIOS configuration settings, which are required in order for the I2c and LCDC Raster operations to work properly. The most important lines in this file are:

```
bios.ECM.ENABLE = 1;
bios.HWI.instance("HWI_INT7").interruptSelectNumber  = 0;
bios.HWI.instance("HWI_INT8").interruptSelectNumber  = 1;
bios.HWI.instance("HWI_INT9").interruptSelectNumber  = 2;
bios.HWI.instance("HWI_INT10").interruptSelectNumber = 3;
```

The above configuration settings are needed to correctly set up the ECM module and map the I2c and LCDC Raster events to the correct CPU interrupts. For example the Lcdc event number is 73, which falls under ECM group 2. Here ECM group 2 is mapped to HWI_INT9, and this is the HWI number used when configuring lcdcParams at runtime (explained further below). For more information on the I2C event mappings, please refer to section 4.11 I2c Sample applications.

Further LCDC Raster static configuration is done in the rasterSample.tci file and raster.tci file. The rasterSample.tci file uses the UDEV module to configure the user defined init function "userRasterInit", and also hook in the LCDC instance parameters (rasterParams). Additionally, the DIO module is used to connect this UDEV instance and specify the channel parameters (chanParams); this DIO instance will be needed

to write to the LCDC Raster controller using the SIO module at run time. In the raster.tci file, the I2C driver is configured, in order for the example to make use of the I2C GPIO expander on the UI board. It is configured to select the routing signals of the raster display.

The configuration of the user init function done in the rasterSample.tci file results in this user defined init function (`userRasterInit`) to be called before the main() function. This function in turn calls the actual `Raster_init()` function (a requirement if a user defined init function is used), and then sets up the user's LCDC Raster instance parameters via "rasterParams".

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

The rasterSampleTask() task exercises the LCDC 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 GIO APIS for I2c communication (please refer to the section on I2C driver and example), and SIO APIs for the LCDC Raster driver channels and also to perform the IO operations.

Please note that, when the raster channel is closed, the driver disables the raster. However, the raster display panel may not go "black" owing to the property of the display. If the user needs such a feature then one may issue an all black image.

5.11.1.2 Build:

This sample can be built using

<ID>/pspiom/examples/evm6747/lcdcraster/build/ccs3/rasterSample.pjt

IMPORTANT NOTE: rasterSample.pjt contains references to %EDMA3LLD_BIOS5_INSTALLDIR% environment variable and links with edma3 libraries. This is required because by default the I2c driver library is built with –DI2c_EDMA_ENABLE. The user can remove all references of EDMA3 from rasterSample.pjt if he re-builds the I2c library without –DI2c_EDMA_ENABLE.

5.11.1.3 Setup:

The sample does not need any special setup apart from plugging in the C6747 User Interface module.

5.11.1.4 Output:

When the sample is run an RGB stripe image with a scrolling line on the image is shown on the raster display.

6 LCDC LIDD Controller Driver

6.1 Introduction

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

DSP/BIOS applications use the driver typically through APIs provided by the GIO layer, to transmit and receive serial data. The following sections describe in detail the necessary procedures to configure and use this driver, as well as other additional information. It is recommended to go through the sample application to get a feel of initializing and using the LCDC LIDD driver.

6.1.1 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

6.1.2 References

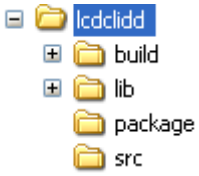
1	SPRUFM0	TMS320C6747 DSP LCD Controller User's Guide

6.2 Installation

The LCDC LIDD device driver is a part of PSP package for C6747 platform and is installed as part of whole package installation.

6.2.1 LCDC LIDD Component folder

On installation of PSP package for C6747, the LCDC LIDD Controller driver can be found at <ID>\ ti\pspiom\lcdclidd\



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

- **lcdclidd -** The lcdclidd folder is the place holder for the entire lcdclidd driver source and the build configuration files. LCDC LIDD driver is implemented as an IOM driver under DSP/BIOS™ operating system. GIO defined APIs can be used to interface to LCDC LIDD driver. This folder contains the build configuration file (package.bld), the LCDC LIDD header file that's included by the application (Lidd.h).
- **build -** contains CCS 3.3 / CCS 4 project files to build the LCDC LIDD library.
- **lib –** contains the LCDC LIDD libraries.
- **src –** Place holder for LCDC LIDD driver's source code.

6.2.2 Build Options

The LCDC LIDD device driver can be built using the CCS v3.3 project file located at <ID>\packages\ti\pspiom\lcdclidd\build\C6747\ccs3\lcdclidd.pjt. This project file supports the following build configurations.

Debug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.

iDebug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "Lcdc_DEBUGPRINT_ENABLE to enable LIDD driver to LOG debug messages.

Release:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver

iRelease:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver
- Defines "Lcdc_DEBUGPRINT_ENABLE to enable LIDD driver to LOG debug messages.

6.2.2.1 Required and Optional Pre-defined symbols

The LCDC LIDD library must be built with a soc specific pre-defined symbol.

"-DCHIP_C6747" is used above to build for C6747. Internally this define is used to select a soc specific header file (soc_C6747.h). This header file contains information such as base addresses of LCDC devices, their interrupt numbers, etc.

If this define is missing, the following compile error will be thrown:

"No chip type defined! (Must use -DCHIP_C6747 or -DCHIP_C6747)"

The LCDC LIDD library can also be built with these optional pre-defined symbols.

Use -DPSP_DISABLE_INPUT_PARAMETER_CHECK when building library to turn OFF parameter checking. This symbol is defined for Release and iRelease profiles by default in the CCS 3.3 pjts provided.

Use -DNDEBUG when building library to turn off runtime asserts. This symbol is defined for Release and iRelease profiles by default in the CCS 3.3 pjts provided.

6.3 Features

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

6.3.1 Multi-Instance Usage

The LIDD driver can be used to operate the LCDC Controller in LIDD mode on the C6747. Currently, only one driver instance for LIDD is supported during driver creation time for the C6747. This is because there is only one LCDC LIDD on the hardware. However, the driver is written in such a way as to support multiple instances for when new SOCs are added which do have multiple controllers. A LCDC LIDD driver instance for the C6747 should use a single instance with device ID 0.

This instance can be operated with configurations supported by The LIDD driver. The device ID can be specified using the `deviceId` field of a UDEV instance (however, only deviceId = 0 is supported).

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

1. Static creation – static creation is done in the "tcf" file of the application; this creation happens at build time. It's necessary to configure LCDC LIDD using the UDEV module (UDEV.create). An instance of the UDEV module at static configuration time corresponds to creating and initializing an LCDC LIDD instance.
2. Dynamic creation – Dynamic creation of an LCDC LIDD instance is done in the application source files by calling DEV_createDevice(); this creation happens at runtime.

UDEV.create and DEV_createDevice allow user to specify the following:

- iomFxns: Pointer to IOM function table. LIDD requires this field to be Lidd_IOMFXNS.
- initFxn: LCDC LIDD requires that the user call Lidd_init() as part of this initFxn. Users can also directly hook in Lidd_init().
- device parameters: LCDC LIDD requires the user to pass an `Lidd_Params` struct. This struct must exist in the application source files and it must be initialized very early as part of driver specific initFxn.
- deviceId to identify the LCDC LIDD peripheral.

For more information on configuring UDEV, DIO and LCDC LIDD, please refer to the LCDC LIDD sample application (included with this driver release), and the DSP/BIOS API Reference (spru403o.pdf, included in your DSP/BIOS installation).

6.3.2 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.

6.4 Configurations

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

6.4.1 Device Parameters

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

Lidd_Params

Serial Number	Parameter	Description
1	devConf	The device configuration provided as a Lidd_DeviceConf structure

6.4.1.1 Lidd_DeviceConf

This structure defines the LCDC device setting configuration.

Serial Number	Parameter	Description
1	clkFreqHz	MCLK frequency desired
2	hwiNum	The HWI event number assigned to the group the LCDC CPU event belongs to
3	numLines	The number of lines in the display.
3	numCharPerLine	The number of characters on each line in the display.
4	addressArray	Array of line start addresses for each line incase of character LCD
5	`pscPwrmEnable`	Boolean flag to enable (TRUE) or disable (FALSE) any power management in the driver

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.

6.4.2 Channel Parameters

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

Lidd_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 Lidd_DisplayConf

6.4.2.1 Display Configuration configuration

Lidd_DisplayConf

Serial Number	Parameter	Description
1	displayType	The type of display interfaced.
2	cs0Timing	Strobe signal timong configuration for device connected on CS0 chip select
3	cs1Timing	Strobe signal timing configuration for device connected on the CS1 chip select
4	chipSel	The chip select number on which the display device is connected and this channel is referring to

6.5 Control Commands

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

Command	Arguments	Description
Lidd_IOCTL_CLEAR_SCREEN	Pointer to ioctlCmdArg type variable.	To clear the display screen, connected on chipSelect specified by the ioctlCmdArg
Lidd_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
Lidd_IOCTL_SET_CURSOR_POSITION	Pointer to CursorPosition structure	To set the cursor to a particular position in the display
Lidd_IOCTL_SET_DISPLAY_ON	Pointer to ioctlCmdArg type variable.	To turn the display on for the chipsel specified by the ioctlCmdArg
Lidd_IOCTL_SET_DISPLAY_OFF	Pointer to ioctlCmdArg type variable.	To turn the display off for, the chipsel specified by the ioctlCmdArg
Lidd_IOCTL_SET_BLINK_ON	Pointer to ioctlCmdArg	To turn the cursor blink

	type variable.	on for display, on the chipsel specified by the ioctlCmdArg
Lidd_IOCTL_SET_BLINK_OFF	Pointer to ioctlCmdArg type variable.	To turn the cursor blink off for display, on the chipsel specified by the ioctlCmdArg
Lidd_IOCTL_SET_CURSOR_ON	Pointer to ioctlCmdArg type variable.	To show the cursor for display, on the chipsel specified by the ioctlCmdArg
Lidd_IOCTL_SET_CURSOR_OFF	Pointer to ioctlCmdArg type variable.	To not show the cursor for display, on the chipsel specified by the ioctlCmdArg
Lidd_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
Lidd_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
Lidd_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
Lidd_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
Lidd_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
Lidd_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
Lidd_IOCTL_COMMAND_REG_WRITE	Pointer to Integer type variable	A generic IOCTL to write a command word to the Character display

6.6 Use of LIDD driver through GIO APIs

6.6.1 GIO_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 tcf or DEV_createDevice()
2	Channel Mode	Should be "IOM_INPUT" when UART requires to received data and "IOM_OUTPUT" when UART requires to transmit
3	Status	Address to place return status from Uart.
4	Channel Params	Pointer to chanParams structure for Uart channel.
5	GIO_Attrs *	Parameters required for the creation of the GIO instance (e.g. channel parameters)

6.6.2 GIO_control

Parameter Number	Parameter	Specifics to PSP
1	GIO_Handle	Handle returned by GIO_create
2	Command	IOCTL command defined by UART driver
3	Arguments	Misc arguments if required by the command

6.6.3 GIO_write

Parameter Number	Parameter	Specifics to Raster
1	Channel Handle	Handle returned by GIO_create
2	Pointer to buffer	Should be pointer to variable of type PSP_Uart_PktAddrPayload OR Uint32 * that holds the audio data.
3	Pointer to size of buffer	Size of the transaction

6.7 Sources that need re-targeting

6.7.1 **ti/pspiom/cslr/soc_C6747.h (soc specific header file):**

This file contains target (SoC) specific definitions. In most cases, changing the values for the of 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.

6.8 EDMA3 Dependency

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

6.9 Known Issues

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

6.10 Limitations

- The LCDC controller on C6747 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.

6.11 LIDD Sample Application

6.11.1.1 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.

6.11.1.2 Build:

This sample can be built using the CCS 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.

6.11.1.3 Setup:

- The Raster display should be removed from the C6747 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.

6.11.1.4 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.

7 SPI driver

7.1 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 the GIO layer, in order to transmit and receive serial data. The following sections describe in detail the necessary procedures to configure and use this driver, as well as other additional information. It is recommended to go through the sample application to get a feel of initializing and using the Spi driver.

7.1.1 Key Features

- Multi-instanceable and re-entrant driver
- Each instance can operate as an receiver and or transmitter
- Supports Polled, Interrupt and DMA Interrupt Mode of operation
- Supports using the GPIOs (External to SPI) to be used as additional chipselects.

7.2 Installation

The Spi device driver is a part of PSP package for the C6747 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\pspiom\spi\docs).

7.2.1 SPI Component folder

On installation of PSP package for the C6747, the SPI driver can be found at <ID>\ ti\psp\spi\

As show above the spi folder contains several sub-folders, the 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. This folder contains Spi.h, which is the header file included by the application.
- **build -** contains CCS 3.3 / CCS 4 project files to build the SPI library.
- **docs –** Contains doxygen generated API reference.
- **src –** Contains the SPI driver's source code.

7.2.2 Build Options

The SPI library can be built using the CCS v3.3 project file located at <ID>\packages\ti\pspiom\spi\build\C6747\ccs3\spi.pjt. This project file supports the following build configurations.

IMPORTANT NOTE:

All build configurations require environment variable %EDMA3LLD_BIOS5_INSTALLDIR% to be defined. This variable must point to "<EDMA3_INSTALL_DIR>\packages".

Debug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "-DSpi_EDMA_ENABLE" to enable EDMA3 support in SPI driver. It also contains "-i%EDMA3LLD_BIOS5_INSTALLDIR%" to find EDMA3 header files.

iDebug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "-DSpi_EDMA_ENABLE" to enable EDMA3 support in Spi driver. It also contains "-i%EDMA3LLD_BIOS5_INSTALLDIR%" to find EDMA3 header files.
- Defines "Spi_DEBUGPRINT_ENABLE to enable Spi driver to LOG debug messages.

Release:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "-DSpi_EDMA_ENABLE" to enable EDMA3 support in Spi driver. It also contains "-i%EDMA3LLD_BIOS5_INSTALLDIR%" to find EDMA3 header files.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver

iRelease:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "-DSpi_EDMA_ENABLE" to enable EDMA3 support in Spi driver. It also contains "-i%EDMA3LLD_BIOS5_INSTALLDIR%" to find EDMA3 header files.
- Defines "Spi_DEBUGPRINT_ENABLE to enable Spi driver to LOG debug messages.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver

7.2.2.1 Required and Optional Pre-defined symbols

The Spi library must be built with a soc specific pre-defined symbol.

"-DCHIP_C6747" is used above to build for C6747. Internally this define is used to select a soc specific header file (soc_C6747.h). This header file contains information such as base addresses of SPI devices, their event numbers, etc.

The Spi library can also be built with these optional pre-defined symbols.

Use –DSpi_EDMA_ENABLE when building library to enable DMA support in Spi driver. If this symbol is not defined edma specific code will get eliminated and the driver can be used only in POLLED or INTERRUPT mode.

Use -DPSP_DISABLE_INPUT_PARAMETER_CHECK when building library to turn OFF parameter checking. This symbol is defined for Release and iRelease profiles by default in the CCS 3.3 pjts provided.

Use -DNDEBUG when building library to turn off runtime asserts. This symbol is defined for Release and iRelease profiles by default in the CCS 3.3 pjts provided.

7.3 Features

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

7.3.1 Multi-Instance

The SPI driver can operate on all the instances of SPI on the EVM6747. Different instances may be specified during driver creation time, and instances 0 through 2 with corresponding device IDs 0 through 2 are supported, respectively.

These instances can operate simultaneously with configurations supported by the SPI driver. SPI instances are created as follows:

1. Static creation – static creation is done in the "tcf" file of the application; this creation happens at build time. The UDEV module (UDEV.create) is used during static configuration. An instance of the UDEV module at static configuration time corresponds to creating and initializing an SPI instance
2. Dynamic creation – Dynamic creation of an SPI instance is done in the application source files by calling DEV_createDevice(); this creation happens at runtime.

UDEV.create and DEV_createDevice allow user to specify the following:

- iomFxns: Pointer to IOM function table. SPI requires this field to be Spi_IOMFXNS.
- initFxn: SPI requires that the user call Spi_init() as part of this initFxn. Users can also directly hook in Spi_init().
- device parameters: SPI requires the user to pass an `Spi_Params` struct. This struct must exist in the application source files and it must be initialized very early as part of driver specific initFxn.
- deviceId to identify the SPI peripheral.

For more information on configuring UDEV and SPI, please refer to the Spi sample application (included with this driver release), and the DSP/BIOS API Reference (spru403o.pdf, included in your DSP/BIOS installation).

7.3.2 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 `GIO_create()` and specifying `"IOM_INOUT"`). The configuration parameters are explained in the sections to follow.

7.3.3 Supports using the GPIOs (External to SPI) to be used as additional chipselects

In scenario where the number of SPI slaves on the EVM are more than the number of native CS lines of the SPI master on the SOC, this feature comes for help.

Free GPIOs could be used for this purpose and if programmed properly, SPI driver internally talks to GPIO driver to toggle the state of corresponding GPIO to act as CS

signal. Detailed information is given below on how to configure the SPI driver for this purpose

7.4 Configurations

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

7.4.1 Spi_Params

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

Members	Description
enableCache	This option is used if the driver should take care of validating/invalidating the cache for the buffers provided by the user.
opMode	Whether the SPI driver should operate in Polled or Interrupt or DMA Interrupt Mode
outputClkFreq	The clock frequency the SPI instance should generate in case of master mode of operation
loopbackEnabled	If the driver/device works in loopback mode
polledModeTimeout	The data transfer timeout for polled mode of operation
spiHWCfgData	The configuration of hardware instance specifc options
edmaHandle	Handle to PSP EDMA LLD driver
hwiNumber	The hardware interrupt number assigned for SPI events
pscPwrmEnable	Boolean flag to enable (TRUE) or disable (FALSE) any power management in the driver
pllDomain	Not used

Note: Please note that in slave mode, power management is not supported.

Apart from the instance parameters described above module wide constants declared in Spi.h can be changed e.g Spi_BUFFER_DATA_SIZE. These constants apply to all Spi instances. Communication mode of operation whether the instance is acting as a slave or master may also be configured.

Additionally, Build options can be added or removed to add/remove features. e.g –DSpi_EDMA_ENABLE.

7.4.2 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, the necessary define switch – DSpi_EDMA_ENABLE should be thrown, as described in section "Build Options".
hGpio	The handle to the GPIO driver. Required only when using any GPIOs for CS operation.

7.4.3 Spi_ChanParams

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.
gpioPinNum	Specifies which pin should be used as CS in case of GPIO CS
csToTxDelay	Specifies the delay between CS assertion and start of I/O transfer

Note:

- The SPI driver is in transceive 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.
- The "chipSelect" parameter specifies which chip select(s) should be used for the current transaction. This parameter is a bitmask of chip selects that are required to be used. For example if chip select 0 and 2 are to be used (0 being the first chip select) then the "chipSelect" should contain a mask = 0x101. Note that bit 0 and bit 2 are set to indicate the use of chipselect 0 and chipselect 2. This configures the appropriate bits (0 and 2) in SCS0FUN field of the SPIPC0 register along with "csDefault" parameter value as described below.
- The "csDefault" parameter in the "spiHWCfgData" of device parameter specifies the configuration bitmask for chip select(s) state in the inactive period. If suppose, chip select 0 and chip select 2 are to used with the respective chip select lines to be high in the inactive state (active high chip select behavior), then "csDefault" should be like 0x101. This value is set in the CSDEF field of the SPIDEF register.

- Spi_IOCTL_SET_CS_POLARITY can be used to toggle the polarity of "csDefault" values. If "isCsActiveHigh" of the command argument (Spi_CsPolarity structure) is FALSE, then the respective bits in "csMask" of the command argument, is set in "csDefault". If "isActiveHigh" of the command argument is TRUE, then the respective bits in "csMask" of the command argument, is reset in "csDefault".
- If it is required that CS0 and CS2 are to be used in active low configuration, then "csDefault" should be 0x101 (inactive high or active low), "chipSelect" should be 0x101. If it is required that CS0 and CS2 are to be used in active high configuration, then "csDefault" should be 0x000 (inactive low or active high), "chipSelect" should be 0x101.

7.4.4 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 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 results 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.

7.4.5 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.

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

7.4.6 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. Also, as part of chanParams, the handle to the EDMA driver, hEdma, should be passed by the application.

Note that -DSpi_EDMA_ENABLE define 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.

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

7.4.7 Slave Mode

The option of slave mode (or master mode) of operation, should be supplied along with the Spi_HWConfigData (device parameter) structure (masterOrSlave field) in the Spi device parameters, when creating an instance 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 -DSpi_EDMA_ENABLE define 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.

Please note the following

- Slave mode of operation is tested at 2MHz. Because of the wired EVM to EVM connectivity in the test setup, signal integrity was not good to test on further higher frequencies.
- (a) Application protocol also needs to consider the latency caused by software slave implementation. (b) The driver does not support "0" no of byte transfer.

7.5 Control Commands

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

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`

7.6 Use of SPI driver through GIO APIs

The following sections explain the use of parameters of GIO calls in the context of the PSP driver. Note that no effort is made to document the use of GIO calls; any PSP specific requirements are covered below.

7.6.1 GIO_create

Parameter Number	Parameter	Description
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 TCF or DEV_createDevice`()`
2	Channel Mode	Should be "`IOM_INPUT`" when SPI requires to received data and "`IOM_OUTPUT`" when SPI requires to transmit
3	GIO_Attrs *	Parameters required for the creation of the GIO instance (e.g. channel parameters)

7.6.2 GIO_control

Parameter Number	Parameter	Specifics to PSP
1	GIO_handle	Handle returned by `GIO_create`
2	Command	IOCTL command defined by SPI driver
3	Arguments	Misc arguments if required by the command

7.6.3 GIO_write/read

Parameter Number	Parameter	Specifics to PSP
1	Channel Handle	Handle returned by `GIO_create`
2	Pointer to buffer	Should be pointer to variable of type `PSP_Spi_PktAddrPayload OR Uint32 *` that holds the audio data.
3	Size	Size of the transaction

7.7 Use of GPIO as chip select

In some cases where the SPI slaves that require CS signal is more than that could be supported by the SPI peripheral, 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 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 C file :

Example:

```
/* start with the default params */
Gpio_Params gpioParams = Gpio_PARAMS;
/* update the gpio parameters to our needs */
gpioParams.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 */
gpioParams.BankParams[0].inUse = Gpio_InUse_No;
gpioParams.BankParams[0].hwiNum  = 9;
/*
It is to be noted here that the pin numbers in GPIO peripheral user guide
```

starts from 1 and end at N. However the GPIO params uses arrays to maintain the pin and bank configuration info. Hence, respective position for this pin in the array will be (pinNumber-1).
**/*
gpioParams.BankParams[0].PinConfInfo[12].inUse = Gpio_InUse_No;
gpioParams.BankParams[0].PinConfInfo[12].inUse = Gpio_InUse_No;

/ open the GPIO driver to get a handle to it */*
gpio0 = Gpio_open(&gpioParams);

This GPIO driver handle should be passed as part of channel parameter (hGpio) during channel creation. The GPIO CS operation is un-defined without a valid GPIO handle.

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;
/ GPIO CS is supported only with CSHOLD feature */*
dataParam.flags = Spi_GPIO_CS | Spi_CSHOLD*;*
Here the slave on GPIO is selected, else the slave on CS selected

b. Specify the GPIO pin number to be used as CS.

Example:
dataParam. gpioPinNum = 13

Note:

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

a. GPIO chip select and native chip select functionality are not supported together in a single submit.
b. 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.
c. GPIO as chip select feature is available only if Spi_CSHOLD flag is included in the Spi_DataParams.flags for every transfer.
d. The GPIO pin used as CS is selectable for every transfer since the GPIO pin number is part of the dataParam.

e. The delay required between CS assertion and start of data transfer (clock out) is programmable via *"csToTxDelay"* of the Spi_DataParam structure for each transfer. However, this delay parameter is just a count that is used in a tight loop inside. This delay loop is not calibrated and the application should adjust this parameter as required.

f. If required GPIO CS polarity can be set as required before each transfer by using the Spi_IOCTL_SET_CS_POLARITY ioctl command request.

7.8 Sources that need re-targeting

7.8.1 ti/pspiom/cslr/soc_C6747.h (soc specific header file):

This file contains target (SoC) specific definitions. In most cases, changing the values for the of 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.

7.9 Use of GPIO as chip select

Any available GPIO pin can be configured as SPI Chip select pin. The user can select any free available GPIO pin and set the gpioChipselectFlag, to use that GPIO pin as SPI chip select pin.

7.10 EDMA3 Dependency

SPI 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.

7.10.1 Used Paramset of EDMA 3

SPI 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.

7.11 Known Issues

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

7.12 Limitations

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

7.13 Spi Sample applications

7.13.1 Interrupt mode sample

7.13.1.1 Description:

This sample demonstrates the use of the Spi driver in interrupt mode.

This example uses the Spi bus to write an array of data to the W25X32 Spi flash memory of the evm6747. 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 GIO modules, in combination. The Spi driver is used to configure and set up the Spi bus, and the GPIO 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.tci and spiSample.tcf files, as well as at run time in the spiSample_main.c and spiSample_io.c files.

The spiSample.tcf 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:

```
bios.ECM.ENABLE = 1;
bios.HWI.instance("HWI_INT8").interruptSelectNumber = 1;
```

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 spiParams at runtime (explained further below).

Further Spi static configuration is done in the spiSample.tci file, which uses the UDEV module to configure the user defined init function "SpiUserInit", and also hook in the Spi instance parameters (spiParams).

At run time, this results in the Spi user defined init function to be called before the main() function. This function in turn calls the actual Spi_init() function (a requirement if a user defined init function is used), and then sets up the user's Spi instance parameters via "spiParams".

Once initialization has completed, the main() function runs, configuring the PINMUX. Following this, the user defined task "echoTask()" runs, which creates GIO Spi read and write handles. These handles are then used when calling the GIO_submit() API to actually write and read data to and from the spi flash memory.

7.13.1.2 Build:

This sample can be built using

<ID>/packages/ti/pspiom/examples/evm6747/spi/interrupt/build/ccs3/spiSample.pjt

IMPORTANT NOTE: spiSample.pjt contains references to %EDMA3LLD_BIOS5_INSTALLDIR% environment variable and links with edma3 libraries. This is required because by default the Spi driver library is built with –DSpi_EDMA_ENABLE. The user can remove all references of EDMA3 from spiSample.pjt if he re-builds the Spi library without –DSpi_EDMA_ENABLE.

7.13.1.3 Setup:

No special setup is needed to run the Spi example

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

7.13.1.4 Output:

When the sample runs, it will output the following:

```
write is Enabled
write is Enabled

BIOS SPI:SPI sample transceive ended successfully

!!! PSP HrtBt
!!! PSP HrtBt
.....
```

7.13.2 Dma mode sample

7.13.2.1 Description:

This sample demonstrates the use of the Spi driver in EDMA mode. In EDMA mode, the Spi driver uses DMA for data transfers, instead of the CPU.

This example uses the Spi bus to write an array of data to the W25X32 Spi flash memory of the evm6747. 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 GIO modules, in combination. The Spi driver is used to configure and set up the Spi bus, and the GIO 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.tci and spiSample.tcf files, as well as at run time in the spiSample_main.c and spiSample_io.c files.

The spiSample.tcf 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 which the user would need in their application are:

```
bios.ECM.ENABLE = 1;
bios.HWI.instance("HWI_INT7").interruptSelectNumber = 0;
bios.HWI.instance("HWI_INT8").interruptSelectNumber = 1;
bios.HWI.instance("HWI_INT9").interruptSelectNumber = 2;
bios.HWI.instance("HWI_INT10").interruptSelectNumber = 3;
```

The above configuration settings are needed to correctly set up the ECM module and map the EDMA events to CPU interrupts. Since the CPU is not used in Spi transfers in EDMA mode, these ECM groups must be mapped to the EDMA events as shown.

Further Spi static configuration is done in the spiSample.tci file, which uses the UDEV module to configure the user defined init function "SpiUserInit", and also hook in the Spi instance parameters (spiParams).

At run time, this results in the Spi user defined init function to be called before the main() function. This function in turn calls the actual Spi_init() function (a requirement if a user defined init function is used), and then sets up the user's Spi instance parameters via "spiParams".

Once initialization has completed, the main() function runs, configuring the PINMUX. Following this, the user defined task "echoTask()" runs, which creates GIO Spi read and write handles. These handles are then used when calling the GIO_submit() API to actually write and read data to and from the spi flash memory.

7.13.2.2 Build:

This sample can be built using

<ID>/packages/ti/pspiom/examples/evm6747/spi/edma/build/ccs3/spiSample.pjt

IMPORTANT NOTE: spiSample.pjt assumes that the Spi driver library is built with –DSpi_EDMA_ENABLE.

7.13.2.3 Setup:

No special setup is needed to run the Spi example

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

7.13.2.4 Output:

When the sample runs, it will output the following:

```
EDMA3 : edma3init() passed
write is Enabled
write is Enabled

BIOS SPI:SPI sample transceive ended successfully

!!! PSP HrtBt
!!! PSP HrtBt
```

8 PSC driver

8.1 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 directly to configure the Psc peripherals. The following sections describe in detail, procedures to use this driver. It is recommended to go through the sample applications to get familiar with using the Psc driver.

8.1.1 Key Features

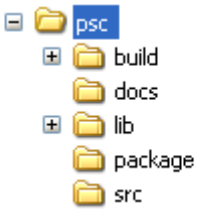
- Does NOT support instances. Simple module level functions.
- Standalone module (driver) ; does not implement IOM interface.

8.2 Installation

The Psc device driver is a part of PSP product for EVM6747 and would be installed as part of whole package installation.

8.2.1 PSC Component folder

On installation of PSP package for C6747, the PSC driver can be found at <ID>\ ti\pspiom\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 driver. This folder contains Psc.h which is the header file included by the application.
- **build –** contains CCS 3.3 / CCS 4 project file to build Psc library.
- **docs –** Contains doxygen generated API reference.
- **lib –** contains Psc libraries
- **src –** contains Psc driver's source code.

8.2.2 Build Options

The Psc library can be built using the CCS v3.3 project file located at <ID>\packages\ti\pspiom\psc\build\C6747\ccs3\psc.pjt. This project file supports the following build configurations.

IMPORTANT NOTE:

All build configurations require environment variable %EDMA3LLD_BIOS5_INSTALLDIR% to be defined. This variable must point to "<EDMA3_INSTALL_DIR>\packages".

Debug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.

iDebug

Release

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.

8.3 Features

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

8.4 Use of PSC driver through module APIs

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

8.4.1 Psc_ModuleClkCtrl

Parameter Number	Parameter	Specifics to PSP
1	Psc device Id	Psc_DevId_0 or Psc_DevId_1
2	Module Id	LPSC number for module
3	isClockEnabled	TRUE or FALSE

This call returns enables/disables the clock domain for the module specified. The sample applications (PSC does not have a separate sample application) all use Psc APIs to configure enable the peripherals.

8.5 Sources that need re-targeting

8.5.1 ti/pspiom/cslr/soc_C6747.h (soc specific header file):

This file contains target (SoC) specific definitions. In most cases, changing the values for the of 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.

8.6 EDMA3 Dependency

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

8.7 Known Issues

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

8.8 Limitations

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

9 Mcasp driver

9.1 Introduction

This document is the reference guide for the Mcasp device driver which explains the features and guidelines for using the driver.

DSP/BIOS applications use the driver typically through APIs provided by SIO layer, to transmit and receive audio data. The following sections describe in detail, the 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 Mcasp driver.

9.1.1 Key Features

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

9.1.2 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

9.1.3 References

1	SPRUFM1	C6747 McASP Reference Guide
2	TLV320AIC31IRHBRG4_3960631	Stereo Audio Codec Data Manual

9.2 Installation

The Mcasp device driver is a part of PSP product for C6747 and would be installed as part of product installation.

9.2.1 PSP Component folder

On installation of the PSP package for C6747, the PSP driver can be found at <ID>\ti\pspiom\mcasp

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

- **Mcasp -** The Mcasp folder is the place holder for the entire Mcasp driver. This folder contains Mcasp.h which is the header file included by the application.
- **build –** contains CCS 3.3 / CCS 4 project file to build Mcasp library.
- **docs –** Contains doxygen generated API reference.
- **lib –** contains Mcasp libraries
- **src –** contains Mcasp driver's source code.

9.2.2 Build Options

The Mcasp library can be built using the CCS v3.3 project file located at <ID>\packages\ti\pspiom\mcasp\build\C6747\ccs3\mcasp.pjt. This project file supports the following build configurations.

Optionally it can also be build using the CCSv4 using the project files located at

<ID>\packages\ti\pspiom\mcasp\build\C6747\ccs4

IMPORTANT NOTE:

All build configurations require environment variable %EDMA3LLD_BIOS5_INSTALLDIR% to be defined. This variable must point to "<EDMA3_INSTALL_DIR>\packages".

Debug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "-DMcasp_EDMA_ENABLE" to enable EDMA3 support in Mcasp driver. It also contains "-i%EDMA3LLD_BIOS5_INSTALLDIR%" to find EDMA3 header files.

iDebug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "-DMcasp_EDMA_ENABLE" to enable EDMA3 support in Mcasp driver. It also contains "-i%EDMA3LLD_BIOS5_INSTALLDIR%" to find EDMA3 header files.

- Defines "Mcasp_DEBUGPRINT_ENABLE to enable Mcasp driver to LOG debug messages.

Release:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "-DMcasp_EDMA_ENABLE" to enable EDMA3 support in Mcasp driver. It also contains "-i%EDMA3LLD_BIOS5_INSTALLDIR%" to find EDMA3 header files.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver

iRelease:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "-DMcasp_EDMA_ENABLE" to enable EDMA3 support in Mcasp driver. It also contains "-i%EDMA3LLD_BIOS5_INSTALLDIR%" to find EDMA3 header files.
- Defines "Mcasp_DEBUGPRINT_ENABLE to enable Mcasp driver to LOG debug messages.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver

9.2.2.1 Required and Optional Pre-defined symbols

The Mcasp library must be built with a soc specific pre-defined symbol.

"-DCHIP_C6747" is used above to build for EVM6747. Internally this define is used to select a soc specific header file (soc_C6747.h). This header file contains information such as base addresses of mcasp devices, their event numbers, etc.

The Mcasp library can also be built with these optional pre-defined symbols.

Use -DPSP_DISABLE_INPUT_PARAMETER_CHECK when building library to turn OFF parameter checking. This symbol is defined for Release and iRelease profiles by default in the CCS 3.3 pjts provided.

Use -DNDEBUG when building library to turn off runtime asserts. This symbol is defined for Release and iRelease profiles by default in the CCS 3.3 pjts provided.

Use –DMcasp_LOOPJOB_ENABLED when the loop job buffer support needs to be enabled. If this support is not enabled, the Mcbsp driver works in non loop job enabled mode

9.3 Features

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

9.3.1 Multi-Instance

The Mcasp driver can operate on all the instances of Mcasp on the EVM6747. Different instances may be specified during driver creation time, and instances 0 through 2 with corresponding device IDs 0 through 2 are supported, respectively.

These instances can operate simultaneously with configurations supported by the Mcasp driver. Mcasp instances are created as follows:

1. Static creation – static creation is done in the "tcf" file of the application; this creation happens at build time. The UDEV module (UDEV.create) is used during static configuration. An instance of the UDEV module at static configuration time corresponds to creating and initializing an MCASP instance
2. Dynamic creation – Dynamic creation of an Mcasp instance is done in the application source files by calling DEV_createDevice(); this creation happens at runtime.

UDEV.create and DEV_createDevice allow user to specify the following:

- iomFxns: Pointer to IOM function table. Mcasp requires this field to be Mcasp_IOMFXNS.
- initFxn: MCASP requires that the user call Mcasp_init() as part of this initFxn. Users can also directly hook in Mcasp_init().
- device parameters: Mcasp requires the user to pass an Mcasp_`Params` struct. This struct must exist in the application source files and it must be initialized very early as part of driver specific initFxn.
- deviceId to identify the Mcasp peripheral.

For more information on configuring UDEV and Mcasp, please refer to the Audio sample application (included with this driver release), and the DSP/BIOS API Reference (spru403o.pdf, included in your DSP/BIOS installation).

9.3.2 Each Instance as Transmitter and / or receiver

Mcasp driver can be simultaneously operated as a transmitter and or receiver. This could be achieved by creating an SIO Channel as an INPUT channel and creating another SIO Channel as an OUTPUT channel. The type of Channel is specified while creating the channel (using `SIO_create ()specify "IOM_OUTPUT" or "IOM_INPUT"`).

The key configuration would be to specify if the transmission section and reception sections clocks are synchronous or not. This is specified by `Mcasp_HwSetupData. clk.clkSetupHiClk` by clearing the BIT 6 or setting the bit for asynchronous mode.

9.3.3 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 and 2 or more serializers. Some of the multi-channel DACs (such as WM8746) expects the samples for all the channels to be received over single serializers. To support these DACs, Mcasp provides support for couple of more data formats. The required buffer format could be configured at driver creation time. The sections below capture the details of supported data formats.

McASP Mode	Single Serializer	Multiple Serializer
Burst Mode / DSP Mode	Interleaved Data Format	Non-interleaved data format
TDM 1 Slot	Interleaved Data Format	Non-interleaved data format
Multi-Slots TDM	Interleaved Data Format Non-interleaveddata format	Non-interleaved data format Semi-interleaved data format
DIT	Interleaved Data Format	Non-interleaved data format

9.3.3.1 *Interleave Data Format (Burst Mode / 1 Slot TDM mode / Multi-Slots TFM / DIT mode)*

When configured as interleaved format, it is expected that McASP is configured to use 1 serializer. The expected data format is as depicted below.

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

The size (number of bytes) that would be required to specify during an IO request is computed using the formula size = <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.

The key configurations are

- `Mcasp_ChanParams.noOfChannels = 0x00`
- `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`>. This value should be given as a size parameter of SIO`_submit` ()
- Idle Time[9.4] data pattern length computation. Minimum length should be **<word width in bytes>** or an integral multiple of computed value. While allocating buffer, allocate `<computed value> * <no of slots enabled>`.

9.3.3.2 *Non-Interleaved Data Format (Burst Mode / 1 Slot TDM mode / Multi-Slots TDM / DIT mode)*

When configured as non-interleaved format, it is expected that PSP driver is configured to use multiple serializers. The expected data format is as depicted below. When configured to use multiple serializers, the samples are expected to be contiguous for a serializer, as depicted below. The assumption here is no of serializers is 2 and no of samples is N

[<**Seriliazer1**-Sample**1**>, <**Seriliazer1**-Sample**2**>…<**Seriliazer1**-Sample**N**>,
<**Seriliazer2**-Sample**1**>, <**Seriliazer2**-Sample**2**>, <**Seriliazer2**-Sample**N**>,
<**Seriliazer3**-Sample**1**>, <**Seriliazer3**-Sample**2**>…<**Seriliazer3**-Sample**N**>]

The key configurations are

- `Mcasp_ChanParams.noOfChannels = 0x00`
- `Mcasp_ChanParams.noOfSerRequested = 0x03`
- `Mcasp_ChanParams.indexOfSersRequested[0] = SERIALIZER_0`
- `Mcasp_ChanParams.indexOfSersRequested[1] = SERIALIZER_6`

- `Mcasp_ChanParams.indexOfSersRequested[2] = SERIALIZER_8`
- The size of the IO request is computed as <`No of Bytes per Sample`> * < `No of Samples` **`per Serializer`**>. This value should be given as a size parameter of `SIO_submit` ()
- Idle Time[9.4] data pattern length computation. Minimum length should be **<word width in bytes>** or an integral multiple of computed value. While allocating the buffer allocate `computed value * no of serializers enabled`.

9.3.3.3 Non-Interleaved Data Format (Multiple Slots Single serializer)

When configured to use multiple slots, one serializer and non-interleaved format. The samples are expected to be contiguous for a slot, as depicted below. The assumption here is no of slots is 2 and no of samples is N

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

<**Slot2**-Sample**1**>, <**Slot2**-Sample**2**>, <**Slot2**-Sample**N**>]

i.e. The samples of Slot1 are contiguous followed by contiguous samples of Slot 2

The key configurations are

- `Mcasp_ChanParams.noOfChannels = 0x00`
- `Mcasp_ChanParams.noOfSerRequested = 0x01`
- The size of the IO request is computed as <`No of Bytes per Sample`> * < `No of Samples` **`per Slot`**>. This value should be given as a size parameter of `SIO_submit` ()
- Idle Time[9.4] data pattern length computation. Minimum length should be **<number of slots enabled> * <word width in bytes>** or an integral multiple of computed value. While allocating the buffer, allocate <compute value> * <no of slots>

Consider as an example where the no of slots are 3 and no of samples per slot is N

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

<**Slot2**-Sample**1**>, <**Slot2**-Sample**2**>, <**Slot2**-Sample**N**>,

<**Slot3**-Sample**1**>, <**Slot3**-Sample**2**>…<**Slot3**-Sample**N**>]

9.3.3.4 Semi-Interleaved Data Format (Multiple Slots Multiple serializer)

When configured to use multi-slots with multi-serializer, the sample for all serializer for a give slot is contiguous, further the samples for all slots are interleaved. The following representation specifies the expected data format. The assumption in this example is we have enabled 2 serializer and two slots in each serializer.

[<**Slot1**-Sample**1**-Serializer**1**>, <**Slot1**-Sample**1**-Serializer**2**>,

<**Slot2**-Sample**2**-Serializer**1**>, <**Slot2**-Sample**2**-Serializer**2**>,…

<**Slot1**-Sample**N**-Serializer**1**>, <**Slot2**-Sample**N**-Serializer**2**>]

The key configurations are

- `Mcasp_ChanParams.noOfChannels = 0x00`
- `Mcasp_ChanParams.noOfSerRequested = 0x02`
- The size of the IO request is computed as <`No of Bytes per Sample`> * < `No of Samples` **`per Slot`**>. This value should be given as a size parameter of `SIO_submit` ()

- Idle Time[9.4] data pattern length computation. Minimum length should be **<number of slots enabled> * <word width in bytes>** or an integral multiple of computed value. While allocating memory for the loopJobBuffer allocate the `computed size * no of serializers enabled`.

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

9.3.4.1 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 9.3.3 for details on the supported data format. The sample application (audioSample_io.c) file demonstrates the required configurations.

9.3.4.2 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. Please refer to sample application (audioSample_io.c) for the required configurations.

9.3.4.3 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. Refer section 9.3.3 for details on the supported data format.

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.

9.3.4.4 DIT (S/PDIF)

To change the User Bits and Channel Status Bits that would be embedded by the S/PDIF SIO, 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.

9.4 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.

9.4.1 Create Time

On successful creations of SIO instances, the Mcasp driver starts generating the clock, Frame Sync and data (if configured as source / if configured as sink Mcasp expects these 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 driver uses driver's internal buffers and length of these NULL buffers is 4 bytes.

9.4.2 Run Time

If the applications could not meet the real time needs of transmission/reception of data, Mcasp driver 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 4 bytes.

9.4.3 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 4 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.

9.5 Explicit control of IO PINS

Mcasp driver 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.

9.6 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.

9.6.1 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.

9.6.2 External Clock

9.6.2.1 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.

9.6.2.2 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)

9.7 Clock Configuration (EVM6747)

Mcasp drivers sample application that came with this release is configured to use external Clock. The configurations are as explained in section 9.6.1. The sample application demonstrates the audio data capturing through the line in and transmits the same data through the line out Pin.

9.8 Configurations

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

9.8.1 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. SIO_create).

Members	Description
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.

pscPwrmEnable	Option to enable/disable the power management features

9.8.2 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

9.8.3 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.

9.8.4 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

9.8.5 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

9.8.6 Mcasp_ChanParams

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

Members	Description
noOfSerRequested	The number of serializers required to use by the channels.
indexOfSersRequested	Index of the serializer that would be required.
mcaspSetup	The hardware configurations required for the channel specifically. Please refer section `Mcasp_HwSetupData`.
channelMode	To operate in DIT/TDM mode
wordWidth	Required word width 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 9.3.4.3 for details.
dataFormat	Buffer format for the audio data to be used by the driver.
EnableHwFifo	Flag to indicate if the Hardware FIFO is to be enabled for this channel.
isDataPacked	flag to indicate if the buffer data needs to be packed, i.e. the EDMA needs to be programmed for the exact

	slot width or a rounded width of 32,16, or 8 Bit is to be used.

9.8.7 Mcasp_PktAddrPayload

Application are expected to pass pointer to this structure in `SIO_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 9.3.3 for details on the supported data formats.

9.9 IO Request Format

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

9.9.1 TDM Mode

Application could pass the address of the audio buffer to McASP via the SIO_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 `Mcasp_PktAddrPayload`. The member `writeDitParams` should be set to `FALSE`.

9.9.2 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 `SIO_submit ()` as a pointer to structure M`casp_PktAddrPayload` and all its members are populated.

9.10 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 SIO_submit () call by providing pre-flushed/pre-invalidate data and disabling the cache option.

9.11 Control Commands

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

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.
Mcasp_IOCTL_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
Mcasp_IOCTL_SET_SAMPLE_RATE	Uint32 *	Command to modify the sample rate.
Mcasp_IOCTL_GET_DEVINFO	Mcasp_AudioDevData *	Command to retrieve the device specific information.

9.12 Use of PSP driver through SIO APIs

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

9.12.1 SIO_create

Parameter Number	Parameter	Specifics to PSP
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 TCF or DEV_createDevice)
2	IO Type	Should be "IOM_INPUT" when McASP requires to received data and "IOM_OUTPUT" when McASP requires to transmit
3	bufSize	Stream buffer size
4	SIO_Attrs *	Parameters required for the creation of the SIO (e.g. channel parameters)

9.12.2 SIO_ctrl

Parameter Number	Parameter	Specifics to PSP
1	SIO_Handle	Handle returned by SIO_create
2	Command	IOCTL command defined by Mcasp driver
3	Arguments	Misc arguments if required by the command

9.12.3 SIO_issue

Parameter Number	Parameter	Specifics to PSP
1	channel Handle	Handle returned by `SIO_create`
2	Pointer to buffer	Should be pointer to variable of type `Mcasp_PktAddrPayload OR Uint32 *` that holds the audio data.
3	arg	User argument
4	Size	Size of the transaction

9.12.4 SIO_reclaim

Parameter Number	Parameter	Specifics to PSP
1	channel Handle	Handle returned by `SIO_create`
2	Pointer to buffer	Should be pointer to variable of type `Mcasp_PktAddrPayload OR Uint32 *` that holds the audio data.
3	Pointer to arg	User argument

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

The behavior of Mcasp driver is better explained under these two sections.

9.13.1 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 C6747, 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.

9.13.2 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.

9.14 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 IOM interface. The driver is tightly coupled with the DSP BIOS operating system

9.15 Sources that need re-targeting

9.15.1 ti/pspiom/cslr/soc_C6747.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.

9.16 EDMA3 Dependency

Mcasp driver relies on PSP EDMA3 driver to move data from/to application buffers to peripheral; typically PSP 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.

9.16.1 Used Paramset of EDMA 3

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

9.17 How to support "NEW" data format

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

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

9.18 Known Issues

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

9.19 Limitations

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

9.20 Mcasp DIT Sample application

9.20.1.1 Description:

This sample 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.

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

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

9.20.1.2 Build:

This sample can be built using the CCS3 or CCS4 interface.

IMPORTANT NOTE: The mcaspDitSample project contains the references to %EDMA3LLD_BIOS5_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.

9.20.1.3 Setup:

You need an "audio board" to be connected to the evm6747. 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).

9.20.1.4 Output:

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

9.21 McASP Sample application

9.21.1.1 Description:

This sample demonstrates the use of the McASP driver in EVM to EVM data communication mode. Mcasp driver supports only DMA mode of operation.

The Mcasp sample application has two projects

1. Master mode project
2. Slave mode project.

Master mode sample application is used to configure one of the EVM as master i.e. it supplies all the required clocks, while the slave mode sample application takes the clocks from an external device.

The driver along with the required component modules are configured statically in mcaspSample.tcf file. The required task for the test application and the memory for the heap are also created here.

The mcaspSample.tcf 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 "Mcasp_echo_task() task exercises the Mcasp driver. It uses Stream APIS to create mcasp driver channels and also to perform the IO operations.

9.21.1.2 Build:

This sample can be built using the CCS3 or the CCS4 interface.

IMPORTANT NOTE: The sample application project contains the references to %EDMA3LLD_BIOS5_INSTALLDIR% environment variable and links with edma3 libraries. This is required because driver by default requires that the EDMA be present.

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

9.21.1.3 Setup:

You need to connect two EVMs with the McASP instance 1 on one EVM connected to the McASP instance 1 on the other EVM. The other settings are as described below.

1. *The Multi channel board is connected on each of the EVMs and the test points are connected as given below..*
2. *The connections for the EVM to EVM are as follows. Refer to the schematics for the PIN number references.*

Master	Slave
ACLKX1	ACLKR1
AFSX1	AFSR1
AXR1[5]	AXR1[5]
GND	GND

9.21.1.4 Output:

The sample on the slave side is loaded and executed first. Next the sample application on the master side is loaded and executed. The output log will indicate if the transmission has passed and also if the reception and data compare is successful.

10 Audio driver

10.1 Introduction

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

DSP/BIOS applications use the driver typically through APIs provided by SIO 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

10.1.1 Key Features

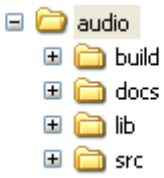
- Multi-instance support and re-entrant driver(10.3.1)
- Each instance can be used to configure a complete receive and transmit section of an audio configuration consisting of an audio device and multiple audio codecs (0).

10.2 Installation

The Audio device driver is a part of PSP product for C6747 and would be installed as part of product installation.

10.2.1 Audio Component folder

On installation of PSP package for C6747, the Audio driver can be found at <ID>\ ti\pspiom\platforms\evm6747\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. This folder contains Audio.h which is the header file included by the application.
- **build –** contains CCS 3.3 / CCS 4 project file to build Audio library.
- **docs –** Contains doxygen generated API reference.
- **lib –** Contains Audio libraries
- **src –** Contains Audio driver's source code.

10.2.2 Build Options

The Audio library can be built using the CCS v3.3 project file located at <ID>\packages\ti\pspiom\platforms\evm6747\audio\build\ccs3\audio.pjt. This project file supports the following build configurations.

Debug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.

iDebug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "Audio_DEBUGPRINT_ENABLE to enable Audio driver to LOG debug messages.

Release:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver

iRelease:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "Audio_DEBUGPRINT_ENABLE to enable Audio driver to LOG debug messages.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver

10.2.2.1 Required and Optional Pre-defined symbols

The Audio library can also be built with these optional pre-defined symbols.

Use -DPSP_DISABLE_INPUT_PARAMETER_CHECK when building library to turn OFF parameter checking. This symbol is defined for Release and iRelease profiles by default in the CCS 3.3 pjts provided.

Use -DNDEBUG when building library to turn off runtime asserts. This symbol is defined for Release and iRelease profiles by default in the CCS 3.3 pjts provided.

10.3 Features

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

10.3.1 Multi-Instance

The Audio driver can operate on all the instances of Mcasp and audio codecs on the EVM6747. Different instances may be specified during driver creation time, and instances 0 through 2 with corresponding device IDs 0 through 2 are supported, respectively.

These instances can operate simultaneously with configurations supported by the Audio driver. Audio instances are created as follows:

1. Static creation – static creation is done in the "tcf" file of the application; this creation happens at build time. The UDEV module (UDEV.create) is used during static configuration. An instance of the UDEV module at static configuration time corresponds to creating and initializing an Audio instance
2. Dynamic creation – Dynamic creation of an Audio instance is done in the application source files by calling DEV_createDevice(); this creation happens at runtime.

UDEV.create and DEV_createDevice allow user to specify the following:

- iomFxns: Pointer to IOM function table. Audio requires this field to be Audio_IOMFXNS.
- initFxn: Audio Interface requires that the user call Audio_init() as part of this initFxn. Users can also directly hook in Audio_init().
- device parameters: Audio driver requires the user to pass an `Audio_Params` struct. This struct must exist in the application source files and it must be initialized very early as part of driver specific initFxn.
- deviceId to identify the Audio peripheral.

For more information on configuring UDEV and Audio, please refer to the Audio sample application (included with this driver release), and the DSP/BIOS API Reference (spru403o.pdf, included in your DSP/BIOS installation).

10.3.2 Each Instance as Transmitter and / or receiver

Audio driver can be simultaneously operated as a transmitter and or receiver. This could be achieved by creaing an SIO Channel as an INPUT channel and creating another SIO Channel as an OUTPUT channel. The type of Channel is specified while creating the channel (using `SIO_create ()specify "IOM_OUTPUT" or "IOM_INPUT"`). The configuration parameters are explained in the sections to follow.

10.4 Configurations

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

10.4.1 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. SIO_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

Apart from the instance parameters described above build options can also be added or removed to add/remove features. e.g –DPSP_DISABLE_INPUT_PARAMETER_CHECK

10.4.2 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.

10.5 Control Commands

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

Command	Arguments	Description
Audio_IOCTL_SAMPLE_RATE	Uint32 *	Changes the sample rate for the audio configurations.

10.6 Use of Audio driver through SIO APIs

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

10.6.1 SIO_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 TCF or AIC31_createDevice ()
2	IO Type	Should be "IOM_INPUT" when Audio requires to received data and "IOM_OUTPUT" when Audio requires to create a transmit channel.
3	bufSize	Stream buffer size
4	SIO_Attrs *	Parameters required for the creation of the SIO (e.g. channel parameters)

10.6.2 SIO_ctrl

Parameter Number	Parameter	Specifics to Audio
1	SIO_Handle	Handle returned by SIO_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.

10.6.3 Stream_issue

Parameter Number	Parameter	Specifics to Audio
1	Channel Handle	Handle returned bySIO_create
2	Pointer to buffer	Should be pointer to variable of type that holds the data to be transmitted.
3	arg	User argument
4	Size	Size of the transaction

10.6.4 SIO_reclaim

Parameter Number	Parameter	Specifics Audio
1	channel Handle	Handle returned by SIO_create
2	Pointer to buffer	Should be pointer to variable Uint32 * that holds the audio data.
3	Pointer to arg	User argument return

10.7 Sources that need re-targeting

10.7.1 ti/pspiom/cslr/soc_C6747.h (soc specific header file):

This file contains target (SoC) specific definitions. In most cases, changing the values for the of 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.

10.8 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.

10.9 Known Issues

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

10.10 Limitations

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

10.11 Audio Sample Application

10.11.1 **Description:**

This sample demonstrates the use of the Audio driver. This application configures the Audio driver to communicate with the Mcasp driver and the Aic31 driver. The Aic31 driver uses the I2c driver. The flow is as follows:

All drivers used in this application are configured in audioSample.tci. The corresponding init functions and global variables are located in audioSample_instParams.c

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

```
bios.ECM.ENABLE = 1;
        bios.HWI.instance("HWI_INT8").interruptSelectNumber = 0;
        bios.HWI.instance("HWI_INT9").interruptSelectNumber = 1;
        bios.HWI.instance("HWI_INT10").interruptSelectNumber = 2;
```

These lines configure the ECM module and map ECM events to CPU interrupts.

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

The Audio_echo_Task () task is the work task that transfers buffers from SIO input channel to SIO output channel.

10.11.1.1 Build:

This sample can be built using

<ID>/packages/ti/pspiom/examples/evm6747/audio/build/ccs3/audioSample.pjt

IMPORTANT NOTE: audioSample.pjt contains references to %EDMA3LLD_BIOS5_INSTALLDIR% environment variable and links with edma3 libraries.

10.11.1.2 Setup:

You need to connect an audio cable from the Host PC audio output to Line IN of evm6747. Then connect another audio cable from Line OUT of evm6747 to a speaker. Play music on the host PC while running the application. Please ensure that the "Multi Channel Audio Board" is NOT plugged into the audio expansion slot of the EVM.

Note: The Multi-channel Audio Board should not be plugged into the EVM while running this sample application.

10.11.1.3 Output:

When the sample runs, you can hear the music from the speakers.

10.12 Dependencies

The audio sample application is dependent on the following drivers

- Audio interface.
- Mcasp driver.
- Aic31 codec driver.

- I2C driver.

10.12.1 Audio Interface

The audio interface provides a high level interface for the user to configure a audio configuration consisting of one audio device and multiple audio codecs. An instance of the Audio interface is used for any data exchange between the application and the underlying audio device/driver .For further details on the usage of the audio interface please refer to the Audio interface user guide and design documents.

10.12.2 McASP Driver

The McASP driver is used to transport audio data to and from the McASP peripheral. The application submits the data read and write requests to the audio interface driver, which in turn are submitted to the Mcasp driver. The McASP driver then reads/writes data to/from the McASP peripheral. For further details on the usage of the Mcasp device and interfaces, please refer to the Mcasp user guide and design documents.

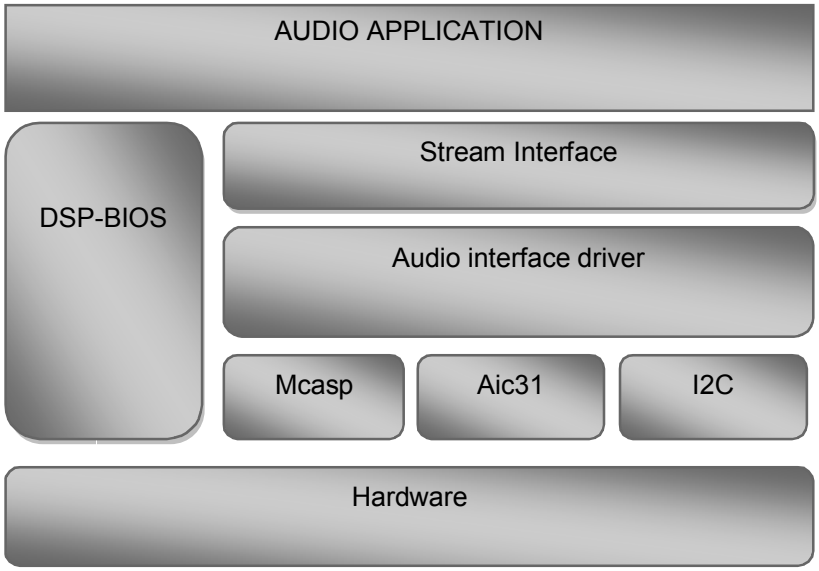
10.12.3 Aic31 Codec Driver

The Aic31 Codec control is interfaced to the SoC through the I2C. The codec can be configured by the application through an I2C interface only. The Aic31 codec converts the digital audio data from the McASP to the analog audio signal and vice versa. Please note that the codec driver does not handle any data transfer request from the application. It only handles the configuration of the audio codec as requested by the audio interface (or application). The application payload (audio) data is transferred to/from the codec is via McASP peripheral pins connected to the codec and this transfer occurs without any explicit request from the application. For further details on the usage of the Aic31 codec please refer to the Aic31 codec driver user guide and design documents.

10.12.4 I2C Driver

The codec cannot be configured directly by the McASP driver. The Aic31 codec control is interfaced to the SoC through an I2C interface. Hence the I2C driver is required for configuring the codec driver. The codec driver internally uses the I2C driver APIs to read and write to the codec registers. The application is expected to initialize the I2 driver prior to using the codec driver. For further details on the usage of the I2C please refer to the I2C user guide and design documents.

The block diagram below depicts the dependencies between the different drivers in the sample application. The audio application interacts with the audio interface driver through stream interface APIs. The audio interface driver internally interacts with the McASP driver and Aic31 driver. The Aic31 driver internally uses the I2C driver to configure the codec registers. The application needs to configure the drivers in the required modes before creating the channels for the audio application.



11 AIC31 CODEC driver

11.1 Introduction

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

DSP/BIOS application uses the driver typically through APIs provided by SIO layer to configure transmit and receive sections. 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 Aic31 driver

11.1.1 Key Features

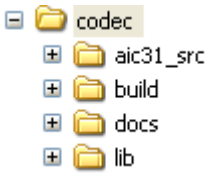
- Multi-instance support and re-entrant driver.
- Each instance can operate as a receiver and or transmitter.
- Interfaces to control the codec specific features like sample rate etc.

11.2 Installation

The Aic31 device driver is a part of PSP product for C6747 and would be installed as part of product installation.

11.2.1 Codec Component folder

On installation of PSP package for C6747, the codec driver can be found at <ID>\ ti\pspiom\platforms\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 all codec driver. This folder contains ICodec.h and Aic31.h which is the header file included by the application.
- **build –** contains CCS 3.3 / CCS 4 project file to build Aic31 library.
- **docs –** Contains doxygen generated API reference.
- **lib –** Contains Aic31 libraries
- **src –** Contains Aic31 driver's source code.

11.2.2 Build Options

The Aic31 library can be built using the CCS v3.3 project file located at <ID>\packages\ti\pspiom\platforms\evm6747\codec\build\ccs3\aic31.pjt. This project file supports the following build configurations.

It can also be built using the CCS v4 project file located at

<ID>\packages\ti\pspiom\platforms\evm6747\codec\build\ccs4

IMPORTANT NOTE:

All build configurations require environment variable %EDMA3LLD_BIOS5_INSTALLDIR% to be defined. This variable must point to "<EDMA3_INSTALL_DIR>\packages".

Debug:

- "-g –mo –mv6740" compile options used to build library.

iDebug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "Aic31_DEBUGPRINT_ENABLE to enable Aic31 driver to LOG debug messages.

Release:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver

iRelease:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "Aic31_DEBUGPRINT_ENABLE to enable Aic31 driver to LOG debug messages.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver

11.2.2.1 Required and Optional Pre-defined symbols

The Aic31 library must be built with a soc specific pre-defined symbol.

"-DCHIP_C6747" is used above to build for the EVM6747. Internally this define is used to select a soc specific header file (soc_C6747.h). This header file contains information such as base addresses of Aic31 devices, their event numbers, etc.

The Aic31 library can also be built with these optional pre-defined symbols.

Use -DPSP_DISABLE_INPUT_PARAMETER_CHECK when building library to turn OFF parameter checking. This symbol is defined for Release and iRelease profiles by default in the CCS 3.3 pjts provided.

Use -DNDEBUG when building library to turn off runtime asserts. This symbol is defined for Release and iRelease profiles by default in the CCS 3.3 pjts provided.

11.3 Features

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

11.3.1 Multi-Instance

The Aic31 codec driver can operate on all the instances of Aic31 on the EVM6747 board. Different instances are specified during driver creation time. Supported instance currently are 1 with instance id 0.

These instances can be operated simultaneously with configurations supported by AIC31 driver.

These instances can operate simultaneously with configurations supported by the Aic31 driver. Aic31 instances are created as follows:

1. Static creation – static creation is done in the "tcf" file of the application; this creation happens at build time. The UDEV module (UDEV.create) is used during static configuration. An instance of the UDEV module at static configuration time corresponds to creating and initializing an Aic31 instance
2. Dynamic creation – Dynamic creation of an Aic31 instance is done in the application source files by calling DEV_createDevice(); this creation happens at runtime.

UDEV.create and DEV_createDevice allow user to specify the following:

- iomFxns: Pointer to IOM function table. Aic31 driver requires this field to be Aic31_IOMFXNS.
- initFxn: Codec driver requires that the user call Aic31_init() as part of this initFxn. Users can also directly hook in Aic31_init().
- device parameters: Aic31 requires the user to pass an `Aic31_Params` struct. This struct must exist in the application source files and it must be initialized very early as part of driver specific initFxn.
- deviceId to identify the Aic31 peripheral.

For more information on configuring UDEV and Aic31, please refer to the Aic31 sample application (included with this driver release), and the DSP/BIOS API Reference (spru403o.pdf, included in your DSP/BIOS installation).

11.3.2 Each Instance as Transmitter and receiver

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 creating an SIO Channel as an INPUT channel and creating another SIO Channel as an OUTPUT channel. The type of Channel is specified while creating the channel (using `SIO_create()specify "IOM_OUTPUT" or "IOM_INPUT"`). The configuration parameters are explained in the sections to follow.

11.3.3 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 11.5

11.4 Configurations

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

11.4.1 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. SIO_create).

Members	Description
acType	Type of the codec
acControlBusType	Control bus to be used by the AIC for configuring of

	the codec(I2C/SPI)
acCtrlBusName	Name of the control bus in the driver table.
acOpMode	Operational mode of the codec(Master/slave)
acSerialDataType	Data transfer format(DSP/TDM/I2S etc)
acSlotWidth	Slot width of the data
acDataPath	Mode to configure the codec.
isRxTxClockIndependent	is the clocks for the RX and TX sections independent

Apart from the instance parameters described above build options can also be added or removed to add/remove features.e.g–DPSP_DISABLE_INPUT_PARAMETER_CHECK

11.4.2 Aic31_ChannelConfig

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

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

11.4.3 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.

11.5 Control Commands

Following table describes some of important the control commands, for a comprehensive list please refer the ICOTL defined in `Aic31.h`

Command	Arguments	Description
Aic31_AC_IOCTL_MUTE_ON	None	Configures the mute for the codec
Aic31_AC_IOCTL_MUTE_OFF	None	Mutes the volume.
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_WR	Aic31_RegData	Writes to the specified

ITE	*	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	ICodec_OutputDest *	Selects the output destination of the audio codec
Aic31_AC_IOCTL_SELECT_INPUT_SOURCE	ICodec_InputDest *	Selects the input source of the Audio codec
Aic31_AC_IOCTL_GET_CODEC_INFO	ICodec_CodecData *	Gets the codec specific information

11.6 Use of AIC31 driver through SIO APIs

Following sections explain the use of parameters of SIO 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.

11.6.1 SIO_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 TCF or DEV_createDevice ()
2	IO Type	Should be "IOM_INPUT" when Audio requires to received data and "IOM_OUTPUT" when Audio requires to create a transmit channel.
3	bufSize	Stream buffer size
4	SIO_Attrs *	Parameters required for the creation of the SIO (e.g. channel parameters)

11.6.2 SIO_ctrl

Parameter Number	Parameter	Specifics to Aic31
1	SIO_Handle	Handle returned by SIO_create
2	Command	IOCTL command defined by device driver to which the command is intended.
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.

11.6.3 Stream_issue

Parameter Number	Parameter	Specifics to Aic31
1	Channel Handle	Handle returned by SIO_create
2	Pointer to buffer	Should be pointer to variable of type that holds the data to be transmitted.
3	arg	User argument
4	Size	Size of the transaction

11.6.4 SIO_reclaim

Parameter Number	Parameter	Specifics to Aic31
1	channel Handle	Handle returned by SIO_create
2	Pointer to buffer	Should be pointer to variable Uint32 * that holds the audio data.
3	Pointer to arg	User argument return

11.7 Sources that need re-targeting

11.7.1 ti/pspiom/cslr/soc_C6747.h (soc specific header file):

This file contains target (SoC) specific definitions. In most cases, changing the values for the of 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.

11.8 EDMA3 Dependency

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

11.9 Known Issues

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

11.10 Limitations

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

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.

The Block Media Driver is written for working with ERTFS. Hence only a ERTFS adaptation is provided. The terms File System and ERTFS are used interchangeably throughout this document.

The interface to the ERTFS file system is guarded by the PSP_FILE_SYSTEM macro which is set to '0' (zero) in blkmediaRaw.pjt. This is enabled to '1' (one) in blkmediaFileSystem.pjt. The library generated by this should be used when using block media driver with ERTFS file system.

Note: The lower level media (mmcsd, nand etc) initialization routines use semaphores and hence can only be called from a task context.

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 C6747 and would be installed as part of product installation.

12.2.1 Block Media Component folder

On installation of PSP package for the C6747, the Block media driver can be found at <ID>\ ti\pspiom\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. This folder contains `psp_blkdev.h` which is the header file included by the application.
- **build –** contains CCS 3.3 / CCS 4 project file to build Block media library. This folder contains two projects inside ccs3 folder:
  - **blkmediaRaw.**pjt – This pjt is used when block media is working in Raw mode.

- **blkmediaFileSystem.pjt** – This pjt is used when block media when File system is used

The respective ccs 4 projects are inside the ccs4\filesystem and ccs4\raw folder

- **docs –** Contains doxygen generated API reference.
- **lib –** Contains Block media libraries
- **src –** Contains Block media driver's source code.

12.2.2 Build Options

The Block media library can be built using the CCS v3.3 project file located at <ID>\packages\ti\pspiom\blkmedia\build\ccs3\C6747\. The project files support the following build configurations.

IMPORTANT NOTE:

All build configurations require environment variable %EDMA3LLD_BIOS5_INSTALLDIR% to be defined. This variable must point to "<EDMA3_INSTALL_DIR>\packages".

Debug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.

Release:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver.

iDebug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "BLKMEDIA_INSTRUMENTATION_ENABLED" to enable Block media driver to LOG debug messages.

iRelease:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "BLKMEDIA_INSTRUMENTATION_ENABLED" to enable Block media driver to LOG debug messages.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver.

IMPORTANT NOTE:

Instrumentation code inside macros for idebug and irelease are not implemented and are just a place holder for future implementation.

12.2.2.1 Required and Optional Pre-defined symbols

The Block media library must be built with a soc specific pre-defined symbol.

"-DCHIP_C6747" is used above to build for C6747. Internally this define is used to select a soc specific header file (soc_C6747.h). This header file contains information such as base addresses of block media devices, their event numbers, etc.

The Block media library can also be built with these optional pre-defined symbols.

Use -DPSP_DISABLE_INPUT_PARAMETER_CHECK when building library to turn OFF parameter checking. This symbol is defined for Release profiles by default in the CCS 3.3 pjts provided.

Use -DNDEBUG when building library to turn off runtime asserts. This symbol is defined for Release profiles by default in the CCS 3.3 pjts provided.

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	Number of Instances of storage drives

	= 2	supported. Currently set to PSP_BLK_DRV_MAX (MMC,NAND and SATA, USB) 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.

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. (Depends on the underlying storage driver support for this IOCTL command)
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`.

Before registering device for RAW access, application must inform block media driver about which device, application wants to set as a RAW device using PSP_blkmediaDrvIoctl() function as explained below, 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);
```

Note: Once the application set a RAW device to MMC/SD, the block media continues to use MMCS/SD as a RAW device, until the application changes the RAW device using the IOCTL call to set RAW device to NAND. Once application set the RAW device to MMC/SD or NAND. Block media remembers the registered RAW device irrespective of multiple times the application calls PSP_blkmediaAppRegister() and PSP_blkmediaAppUnRegister() function.

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 adaptation of storage driver to ERTFS. Please note

it is required to set the FILE_SYSTEM macro to 1 for block media to work seamlessly with the ERTFS file system. The macro is available in `psp_blkdev.h`. Once 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/pspiom/cslr/soc_C6747.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.

12.8 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

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

Please note that the ti.pspiom.blkmedia.raw.a674 library needs to be linked for block media to work seamlessly with media devices in raw mode.

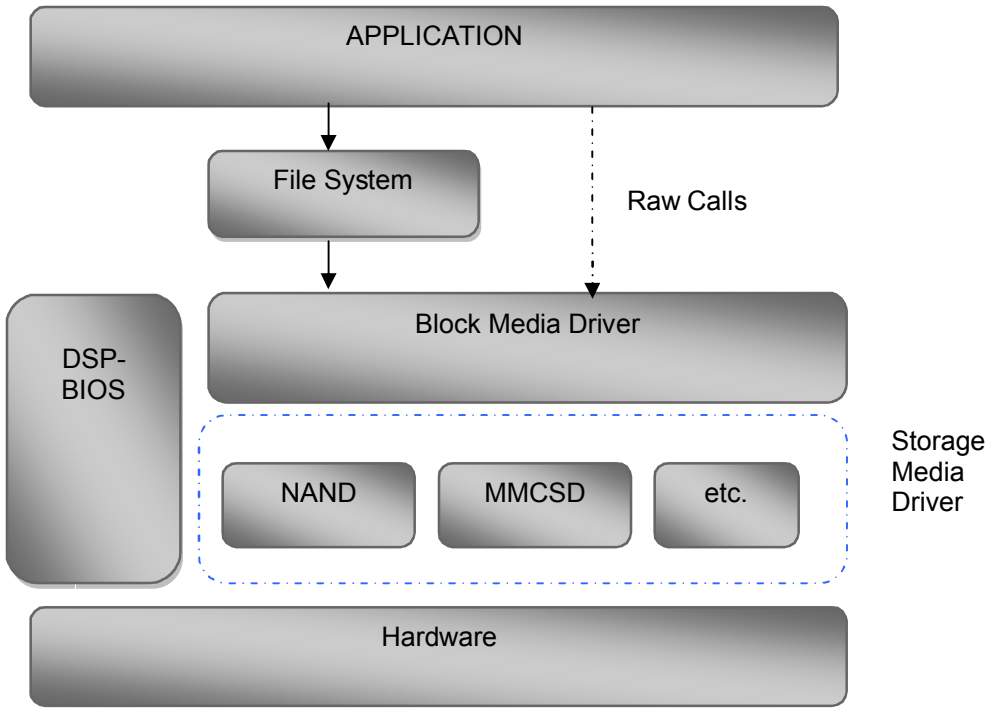
Please refer to the examples section in the File system package for using the file system interface. Please note that the ti.pspiom.blkmedia.filesystem.a674 library needs to be linked for block media to work seamlessly with the ERTFS file system.

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 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.



12.12.1.1 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.

12.12.1.2 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.

12.12.1.3 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.

12.12.1.4 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 C6747 and would be installed as part of product installation.

13.2.1 MMCSD Component folder

On installation of PSP package for the C6747, the MMCSD driver can be found at <ID>\ ti\pspiom\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. This folder contains `psp_mmcsd.h` which is the header file included by the application.
- **build –** contains CCS 3.3 / CCS 4 project file to build Mmcsd library.
- **docs –** Contains doxygen generated API reference.
- **lib –** Contains Mmcsd libraries
- **src –** Contains MMCSD driver's source code.

13.2.2 Build Options

The MMCSD library can be built using the CCS v3.3 project file located at <ID>\packages\ti\pspiom\mmcsd\build\C6747\ccs3\mmcsd.pjt. This project file supports the following build configurations.

IMPORTANT NOTE:

All build configurations require environment variable %EDMA3LLD_BIOS5_INSTALLDIR% to be defined. This variable must point to "<EDMA3_INSTALL_DIR>\packages".

Debug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.

iDebug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "MMCSD_INSTRUMENTATION_ENABLED" to enable Mmcsd driver to LOG debug messages.

Release:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver

iRelease:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "MMCSD_INSTRUMENTATION_ENABLED" to enable Mmcsd driver to LOG debug messages.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver.

IMPORTANT NOTE:

Instrumentation code inside macros for idebug and irelease are not implemented and are just a place holder for future implementation.

13.2.2.1 Required and Optional Pre-defined symbols

The Mmcsd library must be built with a soc specific pre-defined symbol.

"-DCHIP_C6747" is used above to build for C6747. Internally this define is used to select a soc specific header file (soc_C6747.h). This header file contains information such as base addresses of mmcsd devices, their event numbers, etc.

The MMCSD library can also be built with these optional pre-defined symbols.

Use -DPSP_DISABLE_INPUT_PARAMETER_CHECK when building library to turn OFF parameter checking. This symbol is defined for Release and iRelease profiles by default in the CCS 3.3 pjts provided.

Use -DNDEBUG when building library to turn off runtime asserts. This symbol is defined for Release and iRelease profiles by default in the CCS 3.3 pjts provided.

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 EVM6747.

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. 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.
pscPwrmEnable	Boolean flag to enable (TRUE) or disable (FALSE) any power management in the driver

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 Interrupt 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.

Note: The lower level media (mmcsd, nand etc) initialization routines use semaphores and hence can only be called from a task context.

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_mmcsdDrvDeInit**

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/pspiom/cslr/soc_C6747.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.

13.8 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 configures I2C statically in mmcsd.tci file inside platforms\evm6747 folder. This file can be directly imported into an application's tcf script.

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

bios.ECM.ENABLE = 1;

bios.HWI.instance("HWI_INT7").interruptSelectNumber = 0;

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

The echo() task exercises the mmcsd driver. The configureMmcsd function inside the platform file takes care of configuring the pin mux (for mmcsd, I2C and GPIO) and select MMCSD through I2C expander, if UI card is connected and also take care of GPIO based card detection.

Please note that mmcsdStorageInit and mmcsdStorageDeinit functions provided by the platform layer are for the ease for 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 init function is mmcsdStorageInit calls the edma3init, block media init and then the mmcsd init, which initializes the mmcsd driver.

The sample application uses interrupt based detection of card insertion and write protect status via GPIO. To enable this Mmcsd_GPIO_CDWP_ENABLE should be defined in the project as a compiler definition. The macro Mmcsd_GPIO_CDWP_ENABLE is by default enable in the sample application pjt.

The edma3init() initializes the EDMA3 driver and sets up hEdma.

13.11.1.2 Build:

This sample can be built using

<ID>/packages/ti/pspiom/examples/evm6747/mmcsd/edma/build/ccs3/mmcsdSample.pjt

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 trace 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 C6747 and would be installed as part of product installation.

14.2.1 NAND Component folder

On installation of PSP package for the C6747, the NAND driver can be found at <ID>\ ti\pspiom\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. This folder contains `psp_nand.h` which is the header file included by the application.
- **build –** contains CCS 3.3 / CCS 4 project file to build Nand library.
- **docs –** Contains doxygen generated API reference.
- **lib –** Contains Nand libraries
- **src –** Contains Nand driver's source code.

14.2.2 Build Options

The Nand library can be built using the CCS v3.3 project file located at <ID>\packages\ti\pspiom\nand\build\C6747\ccs3\nand.pjt. This project file supports the following build configurations.

IMPORTANT NOTE:

All build configurations require environment variable %EDMA3LLD_BIOS5_INSTALLDIR% to be defined. This variable must point to "<EDMA3_INSTALL_DIR>\packages".

Debug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.

iDebug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "NAND_INSTRUMENTATION_ENABLED" to enable Nand driver to LOG debug messages.

Release:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver

iRelease:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6747" to build library for C6747 soc.
- Defines "NAND_INSTRUMENTATION_ENABLED" to enable Nand driver to LOG debug messages.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver

IMPORTANT NOTE:

Instrumentation code inside macros for idebug and irelease are not implemented and are just a place holder for future implementation.

14.2.2.1 Required and Optional Pre-defined symbols

The Nand library must be built with a soc specific pre-defined symbol.

"-DCHIP_C6747" is used above to build for C6747. Internally this define is used to select a soc specific header file (soc_C6747.h). This header file contains information such as base addresses of nand devices, their event numbers, etc.

The Nand library can also be built with these optional pre-defined symbols.

Use -DPSP_DISABLE_INPUT_PARAMETER_CHECK when building library to turn OFF parameter checking. This symbol is defined for Release and iRelease profiles by default in the CCS 3.3 pjts provided.

Use -DNDEBUG when building library to turn off runtime asserts. This symbol is defined for Release and iRelease profiles by default in the CCS 3.3 pjts provided.

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 EVM6747.

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
pscPwrmEnable	Boolean flag to enable (TRUE) or disable (FALSE) any power management in the driver

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.

Note: The lower level media (mmcsd, nand etc) initialization routines use semaphores and hence can only be called from a task context.

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/pspiom/cslr/soc_C6747.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 configures I2C statically in nand.tci file inside platforms\evm6747 folder. This file can be directly imported into an application's tcf script.

The nandSample.tcf file contains the remaining BIOS configuration.

The echo() 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 edma handle. Please refer to the platforms section in this guide for more details.

Please note that nandStorageInit and nandStorageDeinit functions provided by the platform layer are for the ease for 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.

14.11.1.2 Build:

This sample can be built using

<ID>/packages/ti/pspiom/examples/evm6747/nand/edma/build/ccs3/nandSample.pjt

14.11.1.3 Setup:

You need to connect a UI daughter card having NAND to the evm6747 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 trace window.