Texas Instruments

Texas Instruments THE WORLD LEADER IN DSP AND ANALOG

User Guide

C6748 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

1.1 Introduction

This chapter introduces the C6748 BIOS PSP to the user by providing a brief overview of the purpose and construction of the C6748 BIOS PSP**,** along with hardware and software environment specifics in the context of the C6748 BIOS PSP deployment.

1.1.1 Overview

The C6748 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
BIOSUSB	DSP/BIOs based USB software stack from TI

1.1.3 References

1	SPRUFK5	C6748 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

1.2 Supported Services and features

The C6748 BIOS PSP provides the following:

- Device drivers for UART, I2C, SPI, McASP, McBSP, PSC, MMCSD, GPIO, LCDC LIDD, LCDC Raster, SATA, NAND, VPIF, uPP and devices specific to the EVM like AIC31 codec, Tvp5147 codec and Adv7343 codecs
- Block Media Interface for storage drivers like MMCSD, NAND and SATA.
- Sample applications that demonstrate use of drivers for UART (loop back & Echo Test), I2C (writes to on board I2c Expander), SPI (Serial Flash), McASP (Plays a tone, EVM to EVM communication),McBSP (EVM to EVM communication) MMCSD (Read/Write to the storage devices), NAND (Read/Write to the storage devices), LCDC Raster (Display RGB stripe with scrolling line), VPIF (BT loopback and raw loopback,uPP(evm to evm communication and internal loopback).rCSL and Examples for selected peripherals

1.2.1 System Requirements

The following products are required to be installed prior to using the C6748 BIOS PSP:

- EDMA 3 LLD – This package (C6748 BIOS PSP) is compatible with EDMA 3 LLD versioned 01.11.00.03 or above.
- DSP-BIOS versioned 5.41.03.17
- CCS 3.3.80.11 (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
- EVM 6748 beta 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

1.3 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 in maintenance and usage of code.

1.4 Installation Guide

This chapter discusses the C6748 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 C6748 BIOS PSP.

1.4.1 Installation and Usage Procedure

1.4.1.1 Installation procedure for DSP/BIOS

1. Install the products mentioned in system requirements sections, as per instructions provided along with the products. Please note that sometimes the code composer studio installation would also contain the installation for other components (like DSP/BIOS and Code gen tools) and might install these automatically.
2. Ensure that the BIOS_INSTALL_DIR in the environment variable is set to appropriate DSP/BIOS version.
3. Install the PSP package (BIOSPSP_xx_yy_zz_bb_Setup.exe) using the self extracting installer.This will be installed at the user specified location and an environment variable "BIOS5PSP_INSTALL_DIR" will be created set to the same path.
4. Please note that this installer in an integrated delivery package and it contains device drivers and examples for more the one SoC. You could choose the custom install option during installation to get options to choose the SoC parts you are interested to have device driver and their examples for
5. Install EDMA-3 LLD Device Driver into preferred drive / folder
6. 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 ensure that EDMA install directory environment variable is as follows: EDMA3LLD_BIOS5_INSTALLDIR =c:\edma3_lld_xx_yy_zz\packages)
7. 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. Please refer to RTFS user guide for more detail.
8. For building the downloadable images refer to section 1.4
9. Download the executable image of the required application onto your platform using CCS.
10. Run the program

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

1.4.1.2 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

1.4.2 PSP Component Folder

This section details the files and directory structure of the installed **C6748** 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.

1.4.2.1 Top level PSP Directory structure:

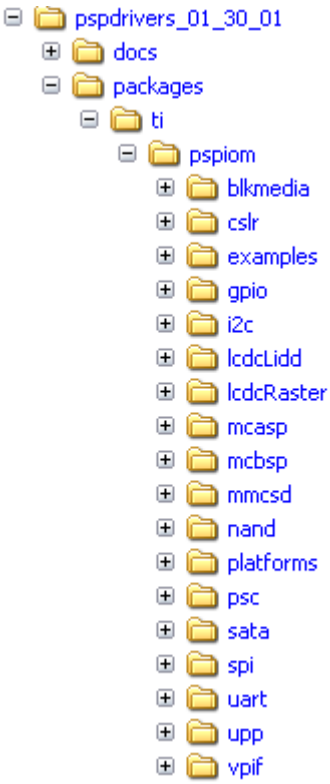


Figure 1: BIOS PSP Top level directory structure

The sections below describe the folder contents.

pspdrivers_xx_xx_xx_xx
Contains 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 C6748 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.

1.4.2.2 *Driver Directory structure:*

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

Figure 2: C6748 PSP driver directory structure

docs

Contains peripheral specifically documentation like Design documentation etc.

build

Contains the CCS3 and CCS4 project files required for building the library.

lib

Contains generated driver library file(s)

src

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

1.4.2.3 *examples Directory structure:*

The example applications for drivers for each EVM platform are arranged under (**ti/pspiom/examples/<evmName>** as follows:

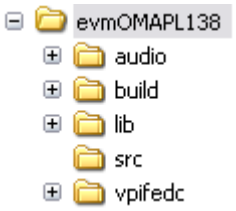
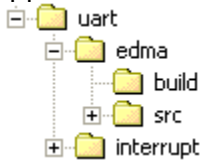


Figure 3: C6748 PSP driver sample application directory structure

evm6748

Contains the EVM/platform specific examples. Further the each sample application is arranged in its own folder as 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

1.4.2.4 platforms Directory structure:

Each platform related specific driver modules are further organized as:

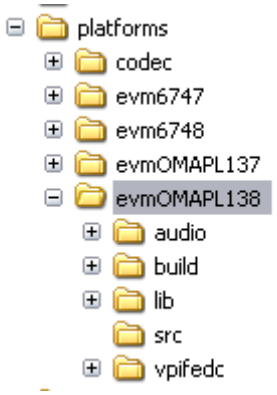


Figure 4 Platforms directory structure

Any EVM dependent driver that could be used across EVMs is kept directly under the *platforms* directory (e,g. codec) and all other EVM specific software content is kept under the <evmName> folder. Typical such candidate is evmInit code and audio driver that encapsulates codec on EVM, audio peripheral on the SOC etc.

codec

Contains codec driver related docs, build files, library files and source files

<evmName>

Contains very EVM specific content

<evmName>\audio

Contains audio interface driver related docs, build files, library files and source files

<evmName>\lib

Contains generated EVM specific initialization (evmInit) library file(s)

<evmName>\src

Contains EVM specific initialization routines source file(s)

<evmName>\build

Contains EVM specific initialization (evmInit) library project files and CCS build files

1.5 Integration Guide

This chapter discusses the **C6748 BIOS PSP** package usage. As part of the PSP package, a sample application is provided to check the basic functionality and usage for each of the device/driver.

1.5.1 Building the PSP Sample Applications

The PSP package contains separate sample applications for each of the DSP/BIOS based drivers provided as part of the package (except PSC). These sample applications can be built using CCS v3.3 project files. These project files can be found in the build folder of the respective sample application in the examples directory (ti\pspiom\examples\<evmName>\<peripheral>).

Also the PSP package can be compiled using CCS4. The project files for compiling the modules will be found in the build folder of the respective module.

1.5.2 BIOS PSP EVM Library Module

1.5.2.1 Description

The sample applications available in the package demonstrate the usage of the BIOS PSP drivers for DSP BIOS 5.33.x on EVM C6748 platform. For successful operation of the applications, some basic initialization (ex., enable the LPSC (clock) for the peripheral, 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. Hypothetically, a device with which the sample application interacts, might be needed to be enabled/selected (multiplexed on the EVM) via an I2C expander, or a configurable switch.

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, PINMUX settings for the module, any configuration required to be done by using the driver. 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.

NOTE:

Please note that all the routines used here are EVM specific and will need to be modified by the system integrator according to the actual EVM used and/or the system use case.

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 and would be called first before any other function from either main or the task.

1.5.2.2 Building the EVM library module

- The CCS3.3 based pjt

  (packages\ti\pspiom\platforms\evm6748\build\ccs3\evmInit.pjt) could be used to build the evmInit library.
- The CCS4 based pjt

  (packages\ti\pspiom\platforms\evm6748\build\ccs4\) could be used to build the evmInit library.

1.5.2.3 Using the EVM library module

- Include the required <ID>\packages\ti\pspiom\platforms\evmXYZ\xxx.tci file in the application tcf file. This file will be required if the platform library for the driver under consideration uses and creates device instances (like in the case if I2C driver is need for I2C IO expander programming etc).
- Include the <ID>\packages\ti\pspiom\platforms\evmXYZ\xxx_evmInit.h file. This will provide the prototypes/declarations
- Link the ti.pspiom.platforms.evmXYZ.evmInit.a674
- Call the required EVM configuration function in the application (depending on the peripheral to use).

1.5.2.4 Porting for another EVM

Please note the current content of this package was targeted for the TI C6748 EVM. In case the package is intended for another custom EVM, the code that needs retargeting is <ti\pspiom\platforms>

- Any new codec driver could be kept at root of "platforms" folder.
- New folder in the name of custom EVM can be created under "platforms folder"
- Duplicate the contents of the "EVM6748" into new folder.
- Change the content of the xxxxinit.c files for appropriate PINMUX, EVM MUX, I2C GPIO expander etc.

1.5.3 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, the user may have to update the versions for DSP/BIOS™, Code generation tools etc for the workspace created. Also, ensure that the settings for the project like output executable/library name etc are retained after switching to the new versions.

1.5.4 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 blinking the LEDs that are connected to a I2C GPIO expander

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 loopback audio capture (the audio fed through Line-in stereo pin from an audio source and playback the audio through the LINEOUT pin on a speaker or headphone).

Mcasp – This sample application demonstrates an EVM to EVM communication example using Mcasp. A known pattern of data to transmit from a Mcasp and another McASP receives the data and compares the same and prints the result.

MMCSD – The sample applications demonstrates the use 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. Find the details of this filesystem package in the System requirement section.

NAND – The sample applications demonstrates the use 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. Find the details of this filesystem package in the System requirement section.

LCDC Raster – The sample application demonstrates the use of the LCDC Raster controller driver by displaying a Video made up of 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.

McBSP – The sample application demonstrates the use of the McBSP driver via EVM to EVM master/slave communication.

VPIF – The sample application demonstrates the use of the VPIF driver by capturing and displaying video in NTSC and RAW modes using different VPIF channels.

UPP – The sample application demonstrates the use of uPP in a EVM to EVM communication example and also an internal loopback example is provided.

Note: Please note that the HWI numbers used for ECM groups 0,1,2,3 are HWI7, HWI8, HWI9 and HWI10 and this would remain common across the sample application of all peripherals.

1.5.5 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\pspiom\cslr\docs\cslr_userguide.doc.

1.6 Power Management

The PSP drivers support various power management features. The following sections explain in detail the power management features supported by the PSP drivers.

1.6.1 Module clock gating

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 implementation uses DSP/BIOS™ PWRM module APIs or BIOSPSP PSC driver APIs depending upon the configuration by the user.

The user can configure the driver to either use DSP/BIOS™ PWRM module APIs by enabling **BIOS_PWRM_ENABLE** compiler switch, or to use the BIOSPSP PSC driver APIs by disabling the BIOS_PWRM_ENABLE compiler switch. That is, when BIOS_PWRM_ENABLE compiler switch is used the drivers shall use the DSP/BIOS™ PWRM API calls. If BIOS_PWRM_ENABLE compiler switch is not used, then the BIOSPSP PSC driver APIs shall be used.

The user shall have to include the following two lines in the application TCF file for DSP/BIOS™ based power management.

```
bios.PWRM.ENABLE = 1;
bios.PWRM.RESOURCETRACKING = 1;
```

Also, if a user wishes not to enable any power management functionality at all in the driver, 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(mdBindDev()) and disabled once during driver instance deletion(mdUnbindDev()).

Please note that DSP/BIOS™ based power management support is currently for **C6748 and OMAPL138 based platform only and only BIOS power management must be used for these platforms**.

1.6.2 DVFS

On the C6748 SoC, dynamic changes to the operating voltage and frequency of the CPU are possible. This is called V/F scaling. Since power usage is linearly proportional to the frequency and quadratically proportional to the voltage, using the V/F scaling can result in significant power savings.

The application can request the DSP for a transition to a new V/F set point whenever it wants to enter a low power state. Whenever the application requests a DVFS setpoint change, the driver internally takes care to suspend the pending IO and resume the same when the V/F scaling is completed. It also takes care to reprogram the various clock dividers so that the actual programmed peripheral IO clock is not affected by the transition to the new setpoint.

Note:

1. The driver shall do the following with respect to the implementation aspects of PWRM "events"
    a. Register notification for PWRM events
    b. Register constraints for non-plausible power states
    c. Perform required operations on notification like deferring the completion of PWRM event if the IO is in progress, stalling subsequent I/O pending

inside the driver until the event is complete, re-configuring clocks (if required) after the event is complete and restarting the IO.

2. The application shall only need to use the PWRM module APIs for a required event. Please note that the pscPwrmEnable should be set to TRUE for driver to respond to the PWRM API calls and perform the required functionality inside the driver
3. All peripheral I/O clock rates may not be possible at all the setpoints available in the system. There could be prescalar programming constraints. In such cases during the PWRM DVFS event notifications, the driver shall (re)register the constraints for the particular non-plausible set point and the driver shall not allow switching to this setpoint. Hence it is the system integrators responsibility to decide on proper setpoints vis-à-vis the IO rates of the system
4. Some drivers may not support power management features in the some modes of operation. Please refer to driver specific section on power management for details
5. If a driver is not power management aware (pscPwrmEnable = FALSE) and the system still performs power management then the driver shall not be able to perform any related functionality during/after the transition PWRM events and the system behavior is unpredictable.

1.6.3 Sleep States

The driver also supports the below mentioned sleep states for the power management and low power states.

1. STANDBY - The GEM is put into a power-saving standby mode. Its clock is turned off at the GEM boundary. This mode has a low latency for wakeup.
2. SLEEP - In addition to putting the GEM into standby, the core voltage is reduced, and the PLLs are slowed down or bypassed.
3. DEEPSLEEP - In addition to the actions for SLEEP, the GEM clock is gated up-stream at the power sleep controller, memories are put into retention, and PLLs are powered down.

The application can use the PWRM provided API's to request the DSP to transition to the required sleep state.

The wakeup events for the sleep states are as given below

1. STANDBY – any enabled interrupt will bring the system out of STANDBY.
2. SLEEP – any enabled interrupt will bring the system out of SLEEP.
3. DEEPSLEEP – only RTC ALARM interrupt (on the OMAPL138 EVM) will bring the system out of DEEPSLEEP.

Please refer to the notes given below for the special considerations to be taken when using the power management features.

The application can request for the V/F scaling and the sleep states to be enabled by supplying the **"pscPwrmEnable"** as TRUE during the device creation. Also it may be required to supply the "**pllDomain**" in which the device is configured.

The user shall have to include the following two lines in the application TCF file for DSP/BIOS™ based power management features of V/F and sleep states to be enabled

```
bios.PWRM.ENABLE = 1;
bios.PWRM.SCALING = 1;
```

The driver internally takes care to suspend any IO pending in the driver and then resume the same when the V/F scaling is completed successfully.

Also note that the driver should be compiled with the "**BIOS_PWRM_ENABLE"** option enabled for the above power management features to be supported.

Note:

- The "pllDomain" parameter is used to notify the driver as to which PLL te device is based of. This is required to appropriately perform the power management related functions in the driver. The "pllDomain" is an enum defined in the driver header files. pllDomain_0 should be passed if the device for which the driver is being instantiated is based off PLL0 and pllDomain_1 should be passed of the device for which the driver is being instantiated is based off PLL1. "pllDomain_0" and "pllDomain_1" correspond to "PWRM_CPU" and "PWRM_PER" type of events respectively. For example, if the pllDomain parameter is set to "pllDomain_0", then the driver shall responed to PWRM_CPU type events. Also, in certain cases the device may be based off external clocks – ASYNC domains. Then in this case the "pllDomain" must be set to "pllDomain_NONE". This important to avoid unneccessary scaling etc inside the driver.
- The V/F scaling and sleep states should be supported by both the underlying SoC and also by the BIOS PWRM module. The BIOS PWRM module is currently supported only on the C6748 and OMAPL138 platforms only.
- The C6748 libraries need to be compiled with the preprocessor "BIOS_PWRM_ENABLE" enabled mandatorily, otherwise the compilation of the libraries will fail.
- One can refer to DSP/BIOS™ API reference guide for PWRM APIs available.
- Additionally, SLEEP and DEEPSLEEP states impose certain constraints on the system under consideration. Please refer to "Known issues" in DSP/BIOS release notes located at the DSP/BIOS 5.41.02.14 installation directory.
- As a jump start one can refer to some basic examples for application level implementation of power management (sleep/vf scaling) found at "packages\ti\bios\examples\advanced" in the DSP/BIOS installation directory.

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

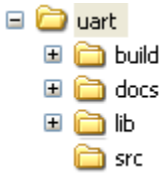
- Multi-instance support and re-entrant driver
- Each instance supports a transmit channel and a receive channel
- Supports Polled, Interrupt and DMA Interrupt Mode of operation

2.2 Installation

The UART device driver is a part of BIOSPSP product for C6748 and would be installed as part of product installation.

2.2.1 UART Component folder

On installation of BIOSPSP package for the C6748, 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 to be included by the applications.
- **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\C6748\ccs3\uart.pjt. This project file supports the following build configurations.

It can also be built using the CCS v4 project files located at <ID>\packages\ti\pspiom\uart\build\C6748\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_C6748" to build library for C6748 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_C6748" to build library for C6748 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_C6748" to build library for C6748 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_C6748" to build library for C6748 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_C6748" is used above to build for C6748. Internally this define is used to select a soc specific header file (soc_C6748.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 -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 and CCS 4 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 EVM C6748. 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; the allocation of device 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	This option is used if the driver should take care of validating/invalidating the cache for the buffers provided by the user.
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 baud rate 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	This defines the type of flow control to be used and the respective flow control parameter.
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	Pll domain where the current device instance is connected to.

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

Please note that the EDMA LLD driver supports multiple instances of the EDMA hardware (2 in case of C6748). The handles to these instances will be valid after calling the edma3init() API. The application should then appropriately pass the EDMA handle via hEdma field above (hEdma[0] or hEdma[1]). If the application is instantiating the driver for device instance number 0 and EDMA event from this device instance are mapped to EDMA controller 0 then the application has to pass hEdma[0].

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 UART
1	Device Name string	Unique identifier used to identify this driver. Please note the name should be same as specified while creating the driver. (Either through 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 UART
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 UART
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 buffer	Size of the transaction

2.7 Sources that need re-targeting

2.7.1 ti/pspiom/cslr/soc_C6748.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.

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 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 available the driver creation would fail. These paramsets are used through the life time of UART 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/evm6748/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 EVM C6748 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\evm6748\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/evm6748/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 EVM C6748 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 ouput channel.

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

The user needs to type or send 100 bytes. This sample application will acho the received characters to the terminal.

The user needs to type or send 1000 bytes. This sample application will echo the received characters to the terminal. The user could make use of the sample.txt file provided with the package at ti\pspiom\examples\evm6748\uart. This file contains 1000 characters of data.

3 I2C driver

3.1 Introduction

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

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

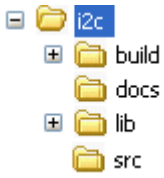
- Multi instantiable and re-entrant driver
- Each instance can operate as an receiver and/or transmitter
- Supports Polled, Interrupt and DMA Interrupt Mode of operation

3.2 Installation

The I2c device driver is a part of the PSP package for the C6748 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 C6748, 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\C6748\ccs3\i2c.pjt. This project file supports the following build configurations.

The project can also be built using the CCS v4 project file located at the <ID>\packages\ti\pspiom\i2c\build\C6748\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_C6748" to build library for C6748 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_C6748" to build library for C6748 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_C6748" to build library for C6748 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_C6748" to build library for C6748 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_C6748" is used above to build for C6748. Internally this define is used to select a soc specific header file (soc_C6748.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 EVM C6748. 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 Power management Considerations

The I2c driver supports the V/F scaling and sleep mode power management features. The following points should be kept in mind when working with the power management enabled.

- The I2c driver will not support power management features when operating in the slave mode.
- In the I2C driver, for device id 0, one should enable pscPwrmEnabe = TRUE even though the instance is on Aysnc3 domain. This is because PWRM also uses the same instance of I2C for communication with the PMIC on board. To workaround any setting user modification during the time PWRM has used the instance, the I2C driver implements a save and restore of the current device context like clock settings, reset condition etc. This has to be done in the PWRM notification callback context which shall be registered only if the pwcPwrmEnable is TRUE.

For other details on the power management support please refer to Power Management.

3.5 Configurations

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

3.5.1 I2c_Params

This structure defines the device configurations, expected to supply while instantiating 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 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	PLL domain where the current device instance is connected to.

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.

3.5.2 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 defined, as described in section 3.2.2 "Build Options".
masterOrSlave	Whether the channel is in Master mode or Slave mode

Please note that the EDMA LLD driver supports multiple instances of the EDMA hardware (2 in case of C6748). The handles to these instances will be valid after calling the edma3init() API. The application should then appropriately pass the EDMA handle via hEdma field above (hEdma[0] or hEdma[1]). If the application is instantiating the driver for device instance number 0 and EDMA event from this device instance are mapped to EDMA controller 0 then the application has to pass hEdma[0].

3.5.3 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.

3.5.4 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.

3.5.5 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.

3.5.6 Slave mode

This version of I2C driver supports slave mode and to use this driver in I2C slave mode the following option are mandatory.

a) masterOrSlave flag in chanparams to select slave mode.

b) Do not 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.
- (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 is exhausted, the slave sends a "NACK".
- In transmit mode, the current IOP is completed when an SCD is detected (generated) on the bus. However, when the transmit buffer has exhausted, though the IOP is completed, dummy bytes are transferred ("I2c_SLAVE_TX_DATA"). This is done to prevent the call to the driver from the application from blocking indefinitely.

3.5.7 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.

3.6 Control Commands

Following table describes some of important the control commands, for a comprehensive list please refer the IOCTL 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

3.7 Use of I2C 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; any PSP specific requirements are covered below.

3.7.1 GIO_create

Parameter Number	Parameter	Specifics to I2C
1	Device Name string	Unique identifier used to identify this driver. Please note the name should be same as specified while creating the driver. (Either through 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)

3.7.2 GIO_control

Parameter Number	Parameter	Specifics to I2C
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

3.7.3 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 buffer that holds the audio data.
3	Size	Size of the transaction

3.8 Sources that need re-targeting

3.8.1 ti/pspiom/cslr/soc_C6748.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.

3.9 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.

3.9.1 Used Paramset of EDMA 3

I2C driver uses TWO paramsets of EDMA3 per instance, one for Tx and another for Rx; if there are no paramsets available the I2C driver creation would fail. These paramsets are used through the lifetime of I2C driver. No link paramsets are used.

3.10 Known Issues

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

3.11 Limitations

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

3.12 I2c Sample application

3.12.1 Interrupt mode sample

3.12.1.1 Description:

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

This example writes to the I2C GPIO expander (TCA6416) to blink the LEDs connected on Port0 of the expander.

The writes to the expander 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 expander, 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.

3.12.1.2 Build:

This sample can be built using

<ID>/packages/ti/pspiom/examples/evm6748/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.

3.12.1.3 Setup:

No special setup is needed to run the I2c example

3.12.1.4 Output:

- When the sample runs, it will output the following in DSP/BIOS message log

  I2C Sample Application

  I2C :Start of I2C sample application

  I2C :End of I2C sample application
- The user LEDs connected to the I2c expander will blink.

3.12.2 DMA Interrupt mode sample

3.12.2.1 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 writes to the I2C GPIO expander (TCA6416) to blink the LEDs connected on Port0 of the expander.

The writes to the expander 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 expander, 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.

3.12.2.2 Build:

This sample can be built using

<ID>/packages/ti/pspiom/examples/evm6748/i2c/edma/build/ccs3/i2cSample.pjt

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

3.12.2.3 Setup:

No special setup is needed to run the I2c example

3.12.2.4 Output:

- When the sample runs, it will output the following in DSP/BIOS message log

```
I2C Sample Application

I2C  :Start of I2C sample application

I2C  :End of I2C sample application
```

- The user LEDs connected to the I2c expander will blink.

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

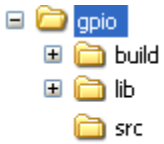
- Setting GPIO pin directions
- Marking pins or banks as available for use
- Enabling and Disabling of bank interrupts
- Registering interrupt handlers for a pin or bank interrupt
- Getting or setting a group of pins to a value

4.2 Installation

The Gpio device driver is a part of the PSP package for the C6748 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 C6748, 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\C6748\ccs3\gpio.pjt. This project file supports the following build configurations.

It can also be built using the CCS v4 project files located at <ID>\packages\ti\pspiom\gpio\build\ C6748\ccs4

Debug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6748" to build library for C6748 soc.

iDebug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6748" to build library for C6748 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_C6748" to build library for C6748 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_C6748" to build library for C6748 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_C6748" is used above to build for C6748. Internally this define is used to select a soc specific header file (soc_C6748.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_C6748 or -DCHIP_C6748)"

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 EVM 6748. 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 Power Management considerations

The GPIO module does not have any kind of power management support.

4.5 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.

4.5.1 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.

4.5.2 Gpio_BankConfig

This structure represents 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).

4.5.3 Gpio_PinConfig

This structure represents 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
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).

4.5.4 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.

4.6 Gpio Bank Event Numbers

The bank event numbers are configured for the Gpio banks on the EVM 6748 can be obtained from the SoC reference Guide. 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.

4.7 Sources that need re-targeting

4.7.1 ti/pspiom/cslr/soc_C6748.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.

4.8 Known Issues

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

4.9 Limitations

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

4.10 GPIO Sample application

4.10.1.1 Description:

This sample demonstrates the use of the GPIO driver.

This example demonstrates the use of GPIO driver in detecting MMCSD cards. The MMCSD card when inserted/removed toggles GPIO pin.

GPIO module APIs are used to interact with the GPIO driver for GPIO operations.

The GPIO driver is configured at run time in the gpioSample_main.c and gpioSample_io.c files. Since, it is not an IOM driver there will be no configuration possible in BIOS configurations file (*.tcf/*.tci).

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 (for example):

```
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 GPIO Bank/Pin event to CPU interrupt.

Once initialization has completed, the main() function runs, configuring the PINMUX. Following this, the user defined task "gpioExampleTask()" runs, which initializes necessary pins and registers interrupt handler. This interrupt handler is invoked whenver there MMCSD card is inserted/removed from the MMCSD slot.

4.10.1.2 Build:

This sample can be built using

<ID>/packages/ti/pspiom/examples/evm6748/gpio/build/ccs3/gpioSample.pjt

4.10.1.3 Setup:

Requires a MMCSD card that will be detected via GPIO.

4.10.1.4 Output:

When the sample runs, the task waits for the MMCSD card insertion. Once the card is inserted the interrupt occurs, which invokes the interrupt handler registered and the messages are printed in the DSP/BIOS message log window.

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

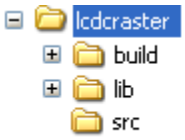
- Multi-instance able, asynchronous and re-entrant driver.
- Each instance operates as a raster controller instance of the LCDC.
- Supports multiple frame sizes – only limited by the hardware.

5.2 Installation

The LCDC Raster device driver is a part of PSP package for C6748 platform and is installed as part of whole package installation.

5.2.1 LCDC Raster Component folder

On installation of PSP package for the C6748, 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\C6748\ccs3\lcdcraster.pjt. This project file supports the following build configurations.

It can also be built using the CCS v4 project files located at <ID>\packages\ti\pspiom\lcdcraster\build\C6748\ccs4

Debug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6748" to build library for C6748 soc.

iDebug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6748" to build library for C6748 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_C6748" to build library for C6748 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_C6748" to build library for C6748 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_C6748" is used above to build for C6748. Internally this define is used to select a soc specific header file (soc_C6748.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_C6748 or -DCHIP_C6748)"

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 C6748. Currently, only one driver instance for LCDC Raster is supported during driver creation time for the C6748. 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 C6748 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 C6748).

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 Power management Considerations

The raster driver supports the V/F scaling and sleep mode power management features. The following points should be kept in mind when working with the power management enabled.

- The application must ensure that the raster is disabled before any of the PWRM events are invoked by the application. Otherwise, the raster driver will return an error.

For other details on the power management support please refer to Power Management

5.5 Configurations

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

5.5.1 Device Parameters

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

Raster_Params

Members	Description
devConf	The device configuration provided as a Raster_DeviceConf structure

5.5.1.1 Raster_DeviceConf

This structure defines the LCDC device setting configuration.

Members	Description
clkFreqHz	The output pixel clock frequency desired to be set
opMode	Mode of operation
hwiNum	The HWI event number assigned to the group the LCDC CPU event belongs to
dma	Configuration for the DMA controller internal to LCDC. This is provided as a Raster_DmaConfig structure
pscPwrmEnable	Boolean flag to enable (TRUE) or disable (FALSE) any power management in the driver
pllDomain	PLL domain where the current device instance is connected to.

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 C6748. 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 TMS320C6748 DSP LCD Controller User's Guide.

5.5.1.2 *Internal DMA Configuration*

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

Raster_DmaConfig

Members	Description
fbMode	The device should operate in single frame buffer mode or double frame buffer mode (ping-pong mode)
burstSize	The chunks of 4-bytes in which the DMA should transfer the data
bigEndian	The operation is big endian mode or little endian mode
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.

5.5.2 Channel Parameters

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

Raster_ChanParams

Members	Description
Controller	The controller type to be configured. This should be configured as a Raster_Controller
chanConf	The Raster controller configuration, given as Raster_RasterConf
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.

5.5.2.1 Raster controller configuration

Raster_RasterConf

Members	Description
outputFormat	Right aligned or left aligned, TFT or STN data format
intface	The physical data interface with the display
panel	Whether STN or TFT type of panel. For raster It should be TFT
display	If monochrome or colour display is interfaced
bitsPP	The number of bits per pixel
fbContent	If the frame buffer contains frame data, pallete, or both
dataOrder	The order of data is arranged is 'LSB to MSB' or 'MSB to LSB'
nibbleMode	If the nibble mode should be enabled. This is true for bits per pixel less than 8 bits
subPanel	The configuration required for sub-panel, when enabled
timing2	The configuration required for SYNC signals and their polarity control
fifoDmaDelay	The delay after which the raster should generate DMA request to the internal DMA controller
intMask	Interrupts which need to be enabled
hFP	Horizontal front porch length in terms of number of pixel clock cycles
hBP	Horizontal back porch length in terms of number of pixel clock cycles
hSPW	Horizontal sync pulse width in terms of number of pixel clock cycles
pPL	Number of pixels per line
vFP	vertical front porch length in terms of number of line clock cycles
vBP	vertical back porch length in terms of number of line clock cycles
vSPW	vertical sync pulse width in terms of number of line clock cycles
lPP	Number of lines per panel

5.6 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	To disable a specific event

	interrupt mask	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

5.7 Use of RASTER driver through SIO APIs

5.7.1 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;

5.7.2 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

5.7.3 SIO_issue

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

5.7.4 SIO_reclaim

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

5.8 Sources that need re-targeting

5.8.1 ti/pspiom/cslr/soc_C6748.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.

5.9 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.

5.10 Known Issues

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

5.11 Limitations

- The LCDC controller on C6748 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.

5.12 Raster Sample Application

5.12.1.1 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, 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 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).

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.

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 SIO APIs for the creation of 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.

5.12.1.2 Build:

This sample can be built using

<ID>/pspiom/examples/evm6748/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.

5.12.1.3 Setup:

The sample does not need any special setup apart from plugging in the C6748 User Interface module.

5.12.1.4 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

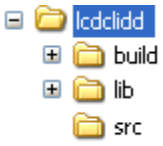
- Multi-instance able, asynchronous and re-entrant driver.
- Each instance operates as a LIDD controller instance of the LCDC
- Supports multiple display types

6.2 Installation

The LCDC LIDD device driver is a part of PSP package for C6748 platform and is installed as part of whole package installation.

6.2.1 LCDC LIDD Component folder

On installation of PSP package for C6748, 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\C6748\ccs3\lcdclidd.pjt. This project file supports the following build configurations.

It can also be built using the CCS v4 project files located at <ID>\packages\ti\pspiom\lcdclidd\build\C6748\ccs4

Debug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6748" to build library for C6748 soc.

iDebug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6748" to build library for C6748 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_C6748" to build library for C6748 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_C6748" to build library for C6748 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_C6748" is used above to build for C6748. Internally this define is used to select a soc specific header file (soc_C6748.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_C6748 or -DCHIP_C6748)"

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 C6748. Currently, only one driver instance for LIDD is supported during driver creation time for the C6748. 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 C6748 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

Members	Description
devConf	The device configuration provided as a Lidd_DeviceConf structure

6.4.1.1 Lidd_DeviceConf

This structure defines the LCDC device setting configuration.

Members	Description
clkFreqHz	MCLK frequency desired
hwiNum	The HWI event number assigned to the group the LCDC CPU event belongs to
numLines	The number of lines in the display.
numCharPerLine	The number of characters on each line in the display.
addressArray	Array of line start addresses for each line incase of character LCD
pscPwrmEnable	Boolean flag to enable (TRUE) or disable (FALSE) any power management in the driver
pllDomain	Pll domain where the current device instance is connected.

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

Members	Description
controller	The controller type to be configured. This should be configured as a Lidd_controller
chanConf	The LIDD controller configuration, given as Lidd_DisplayConf

6.4.2.1 Display Configuration configuration

Lidd_DisplayConf

Members	Description
displayType	The type of display interfaced.
cs0Timing	Strobe signal timong configuration for device connected on CS0 chip select
cs1Timing	Strobe signal timing configuration for device connected on the CS1 chip select
chipSel	This referes to the chip select on which the display device is connected and this channel is created for.

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 type variable.	To turn the cursor blink 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_C	Pointer to ioctlCmdArg	To not show the cursor for

URSOR_OFF	type variable.	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 Raster
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 buffer that holds the transmit data or shall hold the receive 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_C6748.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.

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 C6748 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.tci 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.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.

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 CCS3 or CCS4 interface.

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

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

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

6.11.1.3 Setup:

- The Raster display should be removed from the C6748 Interface Module (UI board)
- The 24x2 character display should be plugged on the UI board.
- The "R104" 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

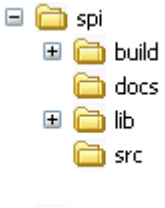
- Multi-instanceable and re-entrant driver
- Each instance can operate as an receiver and or transmitter
- Supports Polled, Interrupt and DMA Interrupt Mode of operation
- Supports using the GPIOs (External to SPI) to be used as additional chipselects.

7.2 Installation

The SPI device driver is a part of PSP package for the C6748 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 C6748, 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\C6748\ccs3\spi.pjt. This project file supports the following build configurations.

It can also be built using the CCS v4 project files located at <ID>\packages\ti\pspiom\spi\build\C6748\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_C6748" to build library for C6748 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_C6748" to build library for C6748 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_C6748" to build library for C6748 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_C6748" to build library for C6748 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_C6748" is used above to build for C6748. Internally this define is used to select a soc specific header file (soc_C6748.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 EVM 6748. 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 Power management Considerations

The SPI driver supports the V/F scaling and sleep mode power management features. The following points should be kept in mind when working with the power management enabled.

- The SPI driver cannot be operated in the salve mode with the power management feature enabled.

For other details on the power management support please refer to Power Management

7.5 Configurations

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

7.5.1 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	PLL domain where the current instance of the device is connected

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.

7.5.2 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.

Please note that the EDMA LLD driver supports multiple instances of the EDMA hardware (2 in case of C6748). The handles to these instances will be valid after calling the edma3init() API. The application should then appropriately pass the EDMA handle via hEdma field above (hEdma[0] or hEdma[1]). If the application is instantiating the driver for device instance number 0 and EDMA event from this device instance are mapped to EDMA controller 0 then the application has to pass hEdma[0].

7.5.3 Spi_DataParam

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

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

7.5.4 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.

7.5.5 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.

7.5.6 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.

7.5.7 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

- (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.

7.6 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`	`Spi_CsPolarity *`	`Configures the CS polarity to High or Low`
`Spi_IOCTL_SET_POLLEDMODETIMEOUT`	`UInt32 *`	`To change the value for polled mode timeout`

7.7 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 SPI specific requirements are covered below.

7.7.1 GIO_create

Parameter Number	Parameter	Specifics to SPI
1	Device Name string	Unique identifier used to identify this driver. Please note the name should be same as specified while creating the driver. (Either through 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)

7.7.2 GIO_control

Parameter Number	Parameter	Specifics to SPI
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

7.7.3 GIO_write/read

Parameter Number	Parameter	Specifics to SPI
1	Channel Handle	Handle returned by GIO_create
2	Pointer to buffer	Should be pointer to variable of type Spi_DataParam.
3	Size	Size of the transaction

7.8 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.

7.9 Sources that need re-targeting

7.9.1 ti/pspiom/cslr/soc_C6748.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.

7.10 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.

7.11 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.

7.11.1 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.

7.12 Known Issues

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

7.13 Limitations

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

7.14 Spi Sample applications

7.14.1 Interrupt mode sample

7.14.1.1 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 EVM 6748. 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.

7.14.1.2 Build:

This sample can be built using

<ID>/packages/ti/pspiom/examples/evm6748/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.

7.14.1.3 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

7.14.1.4 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
.....
```

7.14.2 Dma mode sample

7.14.2.1 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 EVM 6748. 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.

7.14.2.2 Build:

This sample can be built using

<ID>/packages/ti/pspiom/examples/evm6748/spi/edma/build/ccs3/spiSample.pjt

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

7.14.2.3 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

7.14.2.4 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

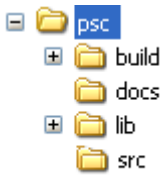
- Does NOT support instances. Simple module level functions.
- Standalone module (driver) ; does not implement IOM interface.

8.2 Installation

The Psc device driver is a part of PSP product for EVM 6748 and would be installed as part of whole package installation.

8.2.1 PSC Component folder

On installation of PSP package for C6748, 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\C6748\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_C6748" to build library for C6748 soc.

iDebug

Release

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6748" to build library for C6748 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_C6748.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.

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).
- Mechanism 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 C6748 McASP Reference Guide

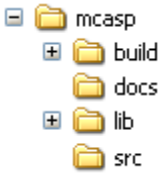
2 TLV320AIC31IRHBRG4_3960631 Stereo Audio Codec Data Manual

9.2 Installation

The Mcasp device driver is a part of PSP product for C6748 and would be installed as part of product installation.

9.2.1 PSP Component folder

On installation of the PSP package for C6748, 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\C6748\ccs3\mcasp.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\mcasp\build\C6748\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_C6748" to build library for C6748 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_C6748" to build library for C6748 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_C6748" to build library for C6748 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_C6748" to build library for C6748 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_C6748" is used above to build for EVM 6748. Internally this define is used to select a soc specific header file (soc_C6748.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 EVM 6748. 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, PSP 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	Interleaved Data Format	Non-interleaved data format

TDM	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[9.5] 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[9.5] 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[9.5] 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[9.5] 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 McaspPSP 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 Power management Considerations

The Mcasp driver supports the V/F scaling and sleep mode power management features. The following points should be kept in mind when working with the power management enabled.

- The McASP driver supports power management features only when the driver is compiled for NON loop job mode.
- Enabling the power management in the loop job mode will result in an error return status from the driver.

For other details on the power management support please refer to Power Management

9.5 IDLE Time Data Patterns

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

9.5.1 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.

9.5.2 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 McaspPSP driver uses driver's internal buffers and length of these NULL buffers is 4 bytes.

9.5.3 IDLE Time buffer size

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

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

If the application does not supply the idle time buffers, the Mcasp driver would use its internal buffer of length 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.

9.6 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

9.7 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.

9.7.1 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.

9.7.2 External Clock

9.7.2.1 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.

9.7.2.2 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 PSP 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)

9.8 Clock Configuration (EVM C6748)

Mcasp drivers sample application that came with this release is configured to use external Clock. The configurations are as explained in section 9.7.1. The sample application demonstrates the audio data capturing through the line in and transmits the same data through the line out Pin.

9.9 Configurations

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

9.9.1 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	This option is used if the driver should take care of validating/invalidating the cache for the buffers provided by the user.
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 in the driver

9.9.2 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

9.9.3 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	Dit Mode support enable disable.

9.9.4 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

9.9.5 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

9.9.6 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 wordwidth in the slots.
isDmaDriven	whether the channel is DMA driven.
userLoopJobBuffer	Buffer to be transferred when the loop job is running.
userLoopJobLength	Number of bytes of the userloopjob buffer for each serializer.
edmaHandle	Handle to PSP EDMA LLD driver

gblCbk	callback required when global error occurs and this must be callable from the ISR context
noOfChannels	No of channels of data to be transmitted. Please refer section 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.

9.9.7 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.

9.10 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.

9.10.1 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`.

9.10.2 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.

9.11 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.

9.12 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_S	Mcasp_HwSet	Re-Configures the channel with new configurations specified. Takes no

ET_FORMAT_CHAN	upData *	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.

9.13 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.

9.13.1 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)

9.13.2 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

9.13.3 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

9.13.4 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

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

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

9.14.1 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 C6748, 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.

9.14.2 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.

9.15 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

9.16 Sources that need re-targeting

9.16.1 ti/pspiom/cslr/soc_C6748.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.

9.17 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.

9.17.1 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.

9.18 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

9.19 Known Issues

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

9.20 Limitations

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

9.21 McASP Sample application

9.21.1.1 Description:

This sample demonstrates the use of the McASP driver in EVM to EVM 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 0 on one EVM connected to the McASP instance 0 on the other evm. The other settings are as described below.

1. *The S7 jumper switch number "2" should be "ON" for both the EVMs.*
2. *The connections for the EVM to EVM are as follows. Refer to the schematics for the PIN number references.*

Master	Slave
ACLKX0(58)	ACLKX0(57)
AFSX0(54)	AFSX0(53)
AXR[9](61)	AXR[9](61)
GND(16)	GND(16)

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 following output will be observed on both the master and slave sides once the application has completed successfully.

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

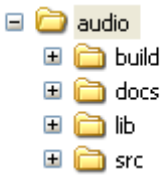
- Multi-instance support and re-entrant driver(10.3.1)
- Each instance can be used to configure a complete receive and transmit section of an audio configuration consisting of an audio device and multiple audio codecs (0).

10.2 Installation

The Audio device driver is a part of PSP product for C6748 and would be installed as part of product installation.

10.2.1 Audio Component folder

On installation of PSP package for C6748, the Audio driver can be found at <ID>\ ti\pspiom\platforms\evm6748\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\evm6748\audio\build\ccs3\audio.pjt. This project file supports the following build configurations.

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

<ID>\packages\ti\pspiom\platforms\evm6748\audio\build\ccs4

Debug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6748" to build library for C6748 soc.

iDebug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6748" to build library for C6748 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_C6748" to build library for C6748 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_C6748" to build library for C6748 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 EVM 6748. 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 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 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 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)

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 to 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_C6748.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.

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_INT7").interruptSelectNumber = 0;
bios.HWI.instance("HWI_INT8").interruptSelectNumber = 1;
bios.HWI.instance("HWI_INT9").interruptSelectNumber = 2;
bios.HWI.instance("HWI_INT10").interruptSelectNumber = 3;
```

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/evm6748/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 EVM 6748. Then connect another audio cable from Line OUT of EVM 6748 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.

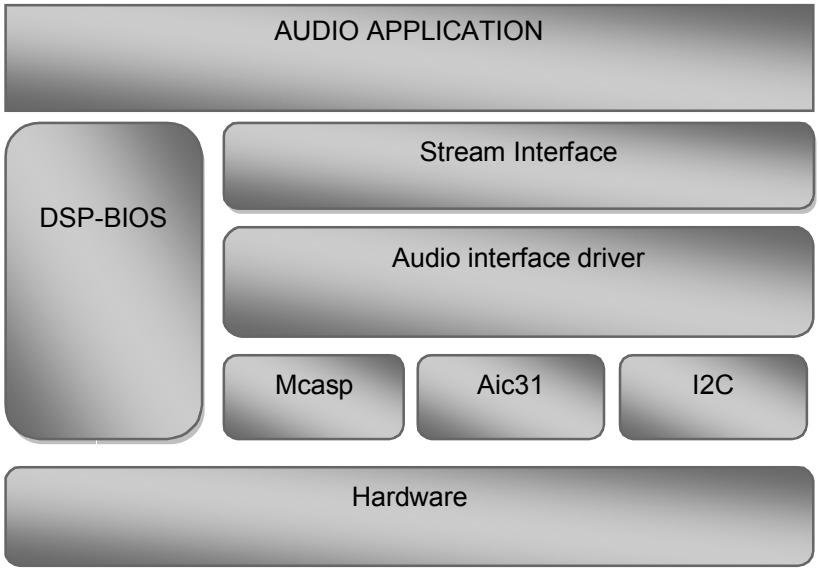
10.12.3 Aic31 Codec Driver

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 applications use the driver typically through APIs provided by SIO layer, to configure the 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

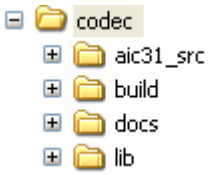
- Multi-instance support and re-entrant driver.
- Each instance can operate as a receiver and or transmitter.
- Interfaces to control the codec specific features like sample rate etc.

11.2 Installation

The Aic31 device driver is a part of PSP product for C6748 and would be installed as part of product installation.

11.2.1 Codec Component folder

On installation of PSP package for C6748, 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\evm6748\codec\build\ccs3\aic31.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.

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_C6748" to build library for C6748 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_C6748" to build library for C6748 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_C6748" is used above to build for the EVM 6748. Internally this define is used to select a soc specific header file (soc_C6748.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 EVM 6748 board. Different instances are specified during driver creation time. Supported instance currently are 0 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	Disables the
Aic31_AC_IOCTL_SET_VOLUME	UInt32 *	Set the required volume for the codec
Aic31_AC_IOCTL_SET_LOOPBACK	None	Not supported
Aic31_AC_IOCTL_SET_SAMPLERATE	UInt32 *	Gets the current sample rate for the audio codec
Aic31_AC__IOCTL_REG_WRITE	Aic31_RegData *	Writes to the specified register
Aic31_AC_IOCTL_REG_READ	Aic31_RegData *	Reads from the specified register
Aic31_AC_IOCTL_REG_WRITE_MULTIPLE	Aic31_RegData *	Writes to the specified number of registers

Aic31_AC_IOCTL_REG_READ_MULTIPLE	Aic31_RegData *	Reads from the specified number of registers
Aic31_AC_IOCTL_SELECT_OUTPUT_SOURCE	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 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.

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_C6748.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.

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 C6748 and would be installed as part of product installation.

12.2.1 Block Media Component folder

On installation of PSP package for the C6748, 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\C6748\ccs3\. 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_C6748" to build library for C6748 soc.

Release:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6748" to build library for C6748 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_C6748" to build library for C6748 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_C6748" to build library for C6748 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_C6748" is used above to build for C6748. Internally this define is used to select a soc specific header file (`soc_C6748.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 = 2	Number of Instances of storage drives 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 individual parameters are provided when the block media driver is initialized via PSP_blkmediaDrvInit(…).

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.

Please note that the EDMA LLD driver supports multiple instances of the EDMA hardware (2 in case of C6748). The handles to these instances will be valid after calling the edma3init() API. The application should then appropriately pass the EDMA handle via hEdma field above (hEdma[0] or hEdma[1]). The block media driver uses free EDMA channels (channels that are not mapped to any device as per the EDMA LLD configuration). These free channels are configured for every instance of the EDMA LDD driver. The application should decide on the EDMA driver instance it will use and pass the EDMA handle appropriately via hEdma. If the application decides to use free channels from EDMA handle 0 then it should pass hEdma[0] and hEdma[1] otherwise.

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	Parameter	Specifics to Block Media

Number		
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_C6748.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

The block media driver uses TWO PaRAM sets. 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.

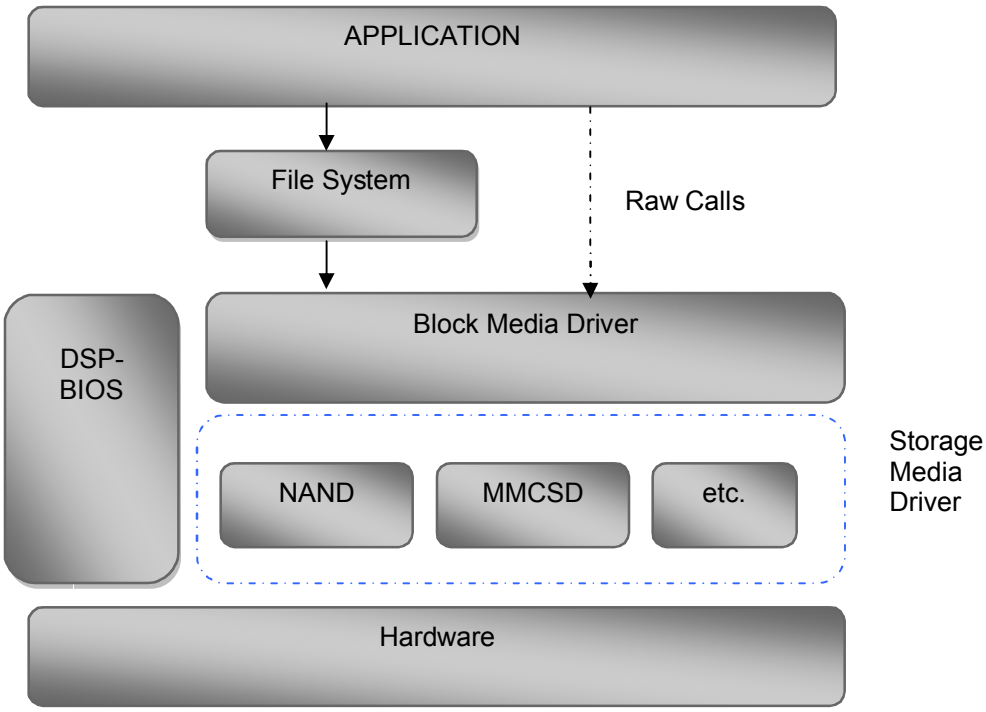
Please refer to the examples section in the File system package for using the file system interface. Please note that the ti.pspiom.blkmedia.filesystem.a674 library needs to be linked for block media to work seamlessly with the ERTFS file system.

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 C6748 and would be installed as part of product installation.

13.2.1 MMCSD Component folder

On installation of PSP package for the C6748, 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\C6748\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_C6748" to build library for C6748 soc.

iDebug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6748" to build library for C6748 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_C6748" to build library for C6748 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_C6748" to build library for C6748 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_C6748" is used above to build for C6748. Internally this define is used to select a soc specific header file (`soc_C6748.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 EVM 6748.

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_MmcsdPllDomain

The PSP_MmcsdPllDomain enumerated data type specifies the PLL domain to the MMCSD device belongs. Following table lists the values of the data type.

Type	Description
`PSP_MMCSD_PLL_DOMAIN_0`	PLL domain 0

PSP_MMCSD_PLL_DOMAIN_1	PLL domain 1

13.4.3 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
pllDomain	Pll domain where the device is

Please note that the EDMA LLD driver supports multiple instances of the EDMA hardware (2 in case of C6748). The handles to these instances will be valid after calling the edma3init() API. The application should then appropriately pass the EDMA handle via hEdma field above (hEdma[0] or hEdma[1]). If the application is instantiating the driver for device instance number 0 and EDMA event from this device instance are mapped to EDMA controller 0 then the application has to pass hEdma[0].

13.4.4 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.

13.4.5 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 Power Management Implementation

13.5.1 DVFS

If there is a request from application for changing the set points (V/F pair), the driver takes care of this and change to the appropriate state. Before calling the set point change event the application should make sure that there is no IO happening inside the driver. If an IO is going on then the driver will not allow set point change. Once the set point is changed the IO's can be submitted again to the driver.

13.5.2 Sleep

If there is a request from application for moving to sleep state (SLEEP/STANDBY/DEEPSLEEP), the driver takes care of these events and change to the appropriate state. Before calling the sleep, the application should make sure that

there is no IO happening in the driver. If an IO is going on then the driver will not allow the sleep change. Once the set point is changed the IO's can be submitted again to the driver.

13.6 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_MmcsdHwEventNotificati`	Pointer to PSP_MmcsdHwEventNotification

	on *	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_CON	Uint32 *	Pointer to Uint32 variable as first

TROLLER_REG		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 *	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.

13.7 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.

13.7.1 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 *

13.7.2 PSP_mmcsdDrvDeInit

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

13.7.3 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

13.8 Sources that need re-targeting

13.8.1 ti/pspiom/cslr/soc_C6748.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.

13.9 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.

13.10 Known Issues

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

13.11 Limitations

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

13.12 MMCSD Sample applications

13.12.1 Dma mode sample

13.12.1.1 Description:

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

The mmcsdSample.tcf file contains the 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 fall in ECM group 0. Here ECM group 0 is mapped to HWI_INT7.

The main() should enable the power of other modules that are used. Sample application calls the mmcsdPscInit() which is defined in the evmInit library.

The echo() task demonstrated the usage of the mmcsd driver. The configureMmcsd() function inside the platform file takes care of configuring the PINMUXes of MMCSD and GPIO (used for interrupt based detection of card insertion).

The init function is mmcsdStorageInit() calls the initialization functions for EDMA3 LLD, block media layer and MMCSD driver. Please refer to the platforms section in this guide for more details.

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 address multiple media, then these APIS should not be used as block media and EDMA initialization is required only once throughout the system

The sample application uses interrupt based detection of card insertion and write protect status via GPIO. 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.

13.12.1.2 Build:

This sample can be built using

<ID>/packages/ti/pspiom/examples/evm6748/mmcsd/edma/build/ccs3/**mmcsdSample.pjt**

13.12.1.3 Setup:

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

13.12.1.4 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

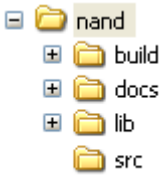
- Supports 512-byte page and 2048-byte page NAND devices
- Supports 8-bit and 16-bit NAND devices
- Error correction using 4-bit ECC mechanism
- Supports wear-leveling and bad-block management functionalities
- Supports protecting a portion of the NAND flash from application access

14.2 Installation

The NAND device driver is a part of PSP product for C6748 and would be installed as part of product installation.

14.2.1 NAND Component folder

On installation of PSP package for the C6748, 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\C6748\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_C6748" to build library for C6748 soc.

iDebug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6748" to build library for C6748 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_C6748" to build library for C6748 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_C6748" to build library for C6748 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_C6748" is used above to build for C6748. Internally this define is used to select a soc specific header file (`soc_C6748.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 EVM 6748.

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 (bypassing the 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

14.4.2.3 PSP_nandPllDomain - The *PSP_nandPllDomain* enumerated data type specifies the PLL domain to the NAND device belongs. Following table lists the values of the data type.

Type	Description
PSP_NAND_PLL_DOMAIN_0	PLL domain 0
PSP_NAND_PLL_DOMAIN_1	PLL domain 1

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

Please note that the EDMA LLD driver supports multiple instances of the EDMA hardware (2 in case of C6748). The handles to these instances will be valid after calling the edma3init() API. The application should then appropriately pass the EDMA handle via hEdma field above (hEdma[0] or hEdma[1]). The NAND driver uses free EDMA channels (channels that are not mapped to any device as per the EDMA LLD configuration). These free channels are configured for every instance of the EDMA LDD driver. The application should decide on the EDMA driver instance it will use and pass the EDMA handle appropriately via hEdma. If the application decides to use free channels from EDMA handle 0 then it should pass hEdma[0] and hEdma[1] otherwise.

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 Power Management Implementation

14.5.1 DVFS

If there is a request from application for changing the set points (V/F pair), the driver takes care of this and change to the appropriate state. Before calling the set point change event the application should make sure that there is no IO happening inside the driver. If an IO is going on then the driver will not allow set point change. Once the set point is changed the IO's can be submitted again to the driver.

14.5.2 Sleep

If there is a request from application for moving to sleep state (SLEEP/STANDBY/DEEPSLEEP), the driver takes care of these events and change to the appropriate state. Before calling the sleep, the application should make sure that there is no IO happening in the driver. If an IO is going on then the driver will not allow the sleep change. Once the set point is changed the IO's can be submitted again to the driver.

14.6 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.

14.7 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.

14.7.1 PSP_nandDrvInit

Parameter Number	Parameter	Specifics to PSP
1	`config`	Configuration parameters of type *PSP_nandConfig* * is passed.

14.7.2 **PSP_nandDrvDeInit**

Parameter Number	Parameter	Specifics to PSP
1	Void	None

14.8 Sources that need re-targeting

14.8.1 **ti/pspiom/cslr/soc_C6748.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.

14.9 EDMA3 Dependency

NAND driver uses ONE PaRAM set. 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.

14.9.1.1 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.

14.10 Known Issues

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

14.11 Limitations

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

14.12 NAND Sample applications

14.12.1 **DMA Interrupt mode sample**

14.12.1.1 Description:

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

The nandSample.tcf file contains the BIOS configuration.

The echo() task exercises the nand driver. The configureNand function inside the platform file takes care of configuring the PINMUXes for NAND.

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.

14.12.1.2 Build:

This sample can be built using

<ID>/packages/ti/pspiom/examples/evm6748/nand/edma/build/ccs3/**nandSample.pjt**

14.12.1.3 Setup:

You need to connect a daughter card having NAND to the EVM 6748.

14.12.1.4 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.

15 McBSP Driver

15.1 Introduction

This document is the reference guide for the Mcbsp 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 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 Mcbsp driver.

15.1.1 Key Features

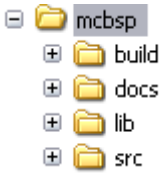
- Multi-instance support and re-entrant driver
- Each instance can operate as a receiver and or transmitter.
- Supports multiple data formats.
- Can be configured to operate in multi-slot TDM, I2S, and DSP. (Used in audio data transfer).
- Mechanisms to transmit desired data (such as NULL tone) when idle

15.2 Installation

The Mcbsp device driver is a part of PSP product for C6748 and would be installed as part of product installation.

15.2.1 PSP Component folder

On installation of the PSP package for C6748, the PSP driver can be found at <ID>\ ti\pspiom\mcbsp



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

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

15.2.2 Build Options

The McBSP library can be built using the CCS v3.3 project file located at <ID>\packages\ti\pspiom\mcbsp\build\C6748\ccs3\mcbsp.pjt. This project file supports the following build configurations.

It can also be built using the CCS4 project files located at the <ID>\packages\ti\pspiom\mcbsp\build\C6748\

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_C6748" to build library for C6748 soc.
- Defines "-DMcbsp_LOOPJOB_ENABLED" to enable loop job mode support in Mcbsp 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_C6748" to build library for C6748 soc.
- Defines "-DMcbsp_LOOPJOB_ENABLED" to enable loop job mode support in Mcbsp driver. It also contains "-i%EDMA3LLD_BIOS5_INSTALLDIR%" to find EDMA3 header files.
- Defines "Mcbsp_DEBUGPRINT_ENABLE to enable Mcbsp driver to LOG debug messages.

Release:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6748" to build library for C6748 soc.
- Defines "-DMcbsp_LOOPJOB_ENABLED" to enable loop job mode support in Mcbsp 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_C6748" to build library for C6748 soc.
- Defines "-DMcbsp_LOOPJOB_ENABLED" to enable loop job mode support in Mcbsp driver. It also contains "-i%EDMA3LLD_BIOS5_INSTALLDIR%" to find EDMA3 header files.
- Defines "Mcbsp_DEBUGPRINT_ENABLE to enable Mcbsp driver to LOG debug messages.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver

15.2.2.1 Required and Optional Pre-defined symbols

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

"-DCHIP_C6748" is used above to build for EVM C6748. Internally this define is used to select a soc specific header file (soc_C6748.h). This header file contains information such as base addresses of mcbsp devices, their event numbers, etc.

The Mcbsp 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 –DMcbsp_LOOPJOB_ENABLE 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.

15.3 Features

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

15.3.1 Multi-Instance

The Mcbsp driver can operate on all the instances of Mcbsp on the EVM C6748. Different instances may be specified during driver creation time, and instances 0 through 1 with corresponding device IDs 0 through 1 are supported, respectively.

These instances can operate simultaneously with configurations supported by the Mcbsp driver. Mcbsp 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 MCBSP instance
2. Dynamic creation – Dynamic creation of an Mcbsp 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. Mcbsp requires this field to be Mcbsp_IOMFXNS.
- initFxn: MCBSP requires that the user call Mcbsp_init() as part of this initFxn. Users can also directly hook in Mcbsp_init().
- device parameters: Mcbsp requires the user to pass an `Mcbsp_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 Mcbsp peripheral.

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

15.3.2 Each Instance as Transmitter and / or receiver

Mcbsp 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"`).

15.3.3 Supported Data Formats

Mcbsp 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 slot or multiple slots. The sections below capture the details of supported data formats.

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

15.3.3.1 Mcbsp_BufferFormat_1SER_1SLOT

This format is used when a single slot is used to transfer the data. 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.

The key configurations are

- `Mcbsp_ChanParams.dataFormat = Mcbsp_BufferFormat_1SER_1SLOT;`
- `Mcbsp_ChanParams. noOfTdmChans = 1;`
- 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 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>`.

15.3.3.2 Mcbsp_BufferFormat_1SER_MULTISLOT_NON_INTERLEAVED

When configured in this mode, it is expected that PSP driver is configured to use multiple slots. The expected data format is as depicted below. When configured to use multiple slots, the samples are expected to be contiguous for a given slot, as 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**>]

The key configurations are

- *Mcbsp_ChanParams.dataFormat= Mcbsp_BufferFormat_1SER_MULTISLOT_NON_INTERLEAVED;*
- Mcbsp_ChanParams. noOfTdmChans = N;
- The size of the IO request is computed as <No of Bytes per Sample> * < No of Samples > * <No of slots>. This value should be given as a size parameter of SIO_submit ()
- Idle Time 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>.

15.3.3.3 Mcbsp_BufferFormat_1SER_MULTISLOT_INTERLEAVED

When configured to use multiple slots and interleaved format. The samples are expected to be interleaved for the slots, as depicted below. The assumption here is no of slots is 2 and no of samples is N

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

The key configurations are

- *Mcbsp_ChanParams.dataFormat= Mcbsp_BufferFormat_1SER_MULTISLOT_INTERLEAVED;*
- Mcbsp_ChanParams. noOfTdmChans = N;
- The size of the IO request is computed as <No of Bytes per Sample> * < No of Samples > * <No of slots>. This value should be given as a size parameter of SIO_submit ()
- Idle Time 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>.

15.3.4 Operational Modes (McBSP, SPI)

15.3.4.1 McBSP

To configure McBSP to work in the normal McBSP mode, configure the mode during the device instance creation as "Mcbsp_OperatingMode_McBSP"

15.3.4.2 SPI

McBSP can be configured to work in the SPI mode of operation. It can operate in either the master mode or the slave mode.To configure McBSP to work in the SPI mode,configure the mode during the operation of the device creation as "Mcbsp_OperatingMode_SPIMASTER" or "Mcbsp_ OperatingMode_SPISLAVE".

Note: The SPI mode of operation is supported only in the SOCs that support operation of the McBSP in SPI mode.The current C6748 SOC does not support the SPI mode of operation.

15.4 Power management Considerations

The Mcbsp driver supports the V/F scaling and sleep mode power management features. The following points should be kept in mind when working with the power management enabled.

- The McBSP driver supports power management features only when the driver is compiled for NON loop job mode.
- Enabling the power management in the loop job mode will result in an error return status from the driver.

For other details on the power management support please refer to Power Management section

15.5 IDLE Time Data Patterns

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

15.5.1 Create Time

On successful creations of SIO instances, the Mcbsp driver starts generating the clock, Frame Sync and data (if configured as source / if configured as sink Mcbsp expects these signals). The data that would be sent out at this point can be configured using `Mcbsp_ChanParams.userLoopJobBuffer` and `Mcbsp_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.

15.5.2 Run Time

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

Mcbsp driver could be configured to send out a know pattern whenever the above situation arises using `Mcbsp_ChanParams.userLoopJobBuffer` and `Mcbsp_ChanParams.userLoopJobLength`. Optionally this could be set NULL and 0x0 respectively, the Mcbsp driver uses driver's internal buffers and length of these NULL buffers is 4 bytes.

15.5.3 IDLE Time buffer size

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

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

If the application does not supply the idle time buffers, the Mcbsp driver would use its internal buffer of length 4 bytes when operating in TDM 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.

Note: This feature can be enabled or disabled by enabling/disabling the "Mcbsp_LOOPJOB_ENABLED" complier switch.

15.6 Clock Configuration (EVM C6748)

McBSP drivers sample applications that came with this release are configured so that the one EVM (slave) uses the bit clock and the frame sync supplied by the other EVM (Master).The configurations are as explained in the following sections. The sample application demonstrates the data transfer between two EVMs. One EVM is

continuously transferring a known pattern of data and the other is continuously capturing the data and comparing the received data with the known pattern.

15.7 Configurations

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

15.7.1 Mcbsp_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
mode	Driver operational mode (i.e. McBSP or SPI) SPI mode support is only available on supported SOCs.
opMode	Driver mode of operation (DMA mode is only supported).
enableCache	whether driver needs to support the cache operations
emulationMode	Emulation mode selection(FREE/SOFT etc)
dlbMode	Loop back mode enable or disable
clkStpMode	Clock stop mode settings.
mcbspSpiFreq	Frequency of the clock when working in SPI mode
srgSetup	Sample rate generator setup.
pscPwrmEnable	option to enable/disable the power management support in the driver
pllDomain	Pll domain where the McBSP device is connected to.

15.7.2 Mcbsp_ChanParams

Members	Description
wordWidth	word width size to be configured.
userLoopJobBuffer	Pointer to user supplied loop job buffer
userLoopJobLength	User supplied buffer length.
gblCbk	global error call back function to be called in case of an error.
edmaHandle	Handle to the EDMA driver
hwiNumber	HWI number to be enabled for this McBSP instance
dataFormat	Format of the data buffer supplied by the application
enableHwFifo	Whether the hardware FIFO is to be enabled or disabled.
chanConfig	configuration for the channel to be created
clkSetup	Clock setup for the channel.

multiChanCtrl	Multi channel control settings.(if required)
chanEnableMask	Channel enable/disable mask

15.8 CACHE Control

McBSP could be configured to FLUSH/INVALIADTE the application supplied buffers while creating the drivers with configuration parameter M**cbsp_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.

15.9 Control Commands

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

Please note that the control commands will be supported only on the basis of the operational mode of the driver(loop job or non loop job mode).

Command	Parameter	Description
Mcbsp_Ioctl_START	NULL	Starts the requested (TX or RX) section.
Mcbsp_Ioctl_STOP	NULL	Stops the requested (TX or RX) section.
Mcbsp_Ioctl_MUTE_ON	NULL	Mutes the TX channel
Mcbsp_Ioctl_MUTE_OFF	NULL	Un-Mutes the TX channel
Mcbsp_Ioctl_PAUSE	NULL	Pauses the selected section (channel)
Mcbsp_Ioctl_RESUME	NULL	Resumes a previously paused channel.
Mcbsp_Ioctl_CHAN_RESET	NULL	Resets the requested channel.
Mcbsp_Ioctl_DEVICE_RESET	NULL	Resets the entire device.
Mcbsp_Ioctl_SRGR_START	NULL	starts the sample rate generator
Mcbsp_Ioctl_SRGR_STOP	NULL	Stops the sample rate generator

Mcbsp_Ioctl_FSGR_START	NULL	starts the frame sync generator
Mcbsp_Ioctl_FSGR_STOP	NULL	stops the frame sync generator

15.10 Use of McBSP driver through SIO APIs

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

15.10.1 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 McBSP requires to received data and "IOM_OUTPUT" when McBSP requires to transmit
3	bufSize	Stream buffer size
4	SIO_Attrs *	Parameters required for the creation of the SIO (e.g. channel parameters)

15.10.2 SIO_ctrl

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

15.10.3 SIO_issue

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

15.10.4 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 that holds the data.
3	Pointer to arg	User argument

15.11 Porting Guide

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

The McBSP Device Driver is based upon the DSP BIOS IOM interface. The driver is tightly coupled with the DSP BIOS operating system.

15.12 Sources that need re-targeting

15.12.1 ti/pspiom/cslr/soc_C6748.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.

15.13 EDMA3 Dependency

Mcbsp 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.

15.13.1 Used Paramset of EDMA 3

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

15.14 Known Issues

1. The audio data support for the McBSP driver is not tested as the EVM does not have the support for the same.
2. Please refer to the top level release notes that came with this release.

15.15 Limitations

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

15.16 Mcbsp Sample application

15.16.1.1 Description:

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

The Mcbsp 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 mcbspSample.tcf file. The required task for the test application and the memory for the heap are also created here.

The mcbspSample.tcf file contains the remaining BIOS configuration like the configuration of the event combiner etc. This helps to map the Mcbsp events to the CPU interrupts.

The "mcbspDemoTask() task exercises the Mcbsp driver. It uses Stream APIS to create mcbsp driver channels and also to perform the IO operations.

15.16.1.2 Build:

This sample can be built using the CCS3 interface 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.

15.16.1.3 Setup:

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

1. *The S7 jumper switch number "2" should be "ON" for both the EVMs.*
2. *The connections for the EVM to EVM are as follows. Refer to the schematics for the PIN number references.*

Master	Slave
CLKX1(65)	CLKR1(17)
CLKR1(17)	CLKX1(65)
DX1(61)	DR1(23)
FSX1(23)	FSR1(13)
FSR1(13)	FSX163)
GND(59)	GND(59)

15.16.1.4 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 following output will be observed on both the master and slave sides once the application has completed successfully.

EDMA intialised

Mcbsp driver primed.

Sample Application completed sucessfully...

16 SATA driver

16.1 Introduction

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

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

16.1.1 Key Features of SATA subsystem

The AHCI compliance SATA Subsystem provides the following features.

- Serial ATA 1.5Gbps and 3Gbps speeds [2]
- Integrated TI SERDES
- Integrated Rx and Tx data buffers
- Supports all SATA power management features
- Internal DMA Engine
- Support one SATA port, hence only one SATA device can be connected.

16.1.2 Features support by driver

- Support ATA/ATAPI protocol
- PWRM Power Management
    - IO based Power Management is supported by driver. Whenever there is no SATA I/O request is pending the SATA Clock is disabled and enabled before start of any IO.
    - Supports PWRM. The PWRM modes supported are
        - DVFS
        - Standby
        - Sleep
        - DeepSleep

16.1.3 Features Not support by driver

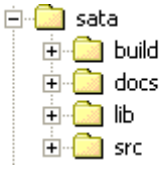
- Port Multiplier support.
- CD/DVD - ATAPI support
- SATA as removable media
- Native command Queueing

16.2 Installation

The SATA device driver is a part of PSP product for C6748 and would be installed as part of product installation.

16.2.1 SATA Component folder

On installation of PSP package for the C6748, the SATA driver can be found at <ID>\ ti\pspiom\sata\



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

- **sata -** The sata folder is the place holder for the entire sata driver. This folder contains `psp_sata.h`, psp_ata_med.h which is the header file included by the application.
- **build –** contains CCS 3.3 / CCS 4 project file to build sata library.
- **docs –** Contains sata driver design document.
- **lib –** Contains sata libraries
- **src –** Contains sata driver's source code.

16.2.2 SATA Dependent components

SATA depends on blkmedia component of BIOSPSP.

16.2.3 Build Options

The SATA library can be built using the CCS v3.3 project file located at <ID>\packages\ti\pspiom\sata\build\C6748\ccs3\sata.pjt. This project file supports the following build configurations.

Debug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6748" and "-DBIOS_PWRM_ENABLE" to build library for C6748 soc.

Release:

- "-o3 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6748" and "-DBIOS_PWRM_ENABLE" to build library for C6748 soc.

16.2.3.1 Required and Optional Pre-defined symbols

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

"-DCHIP_C6748" is used above to build for C6748. Internally this define is used to select a soc specific header file (soc_C6748.h). This header file contains information such as base addresses of sata ahci controller, their event numbers, etc.

"-DBIOS_PWRM_ENABLE" this option will enable PWRM module for power management of the driver. By default this option is enabled by the driver.

16.2.4 Power Management Configuration

The power management for SATA can be disabled by setting sata configuration parameter to Sata Driver initialization routine. Int32 PSP_sataDrvInit(Uint32 inst_id, PlatformResource *platform_res) defined in psp_sata.h. Note that the platform_resource variable of type PlatformResource passed to PSP_sataDrvInit() function should be declared as global variable.

The PlatformResource configuration structure definition is

typedef struct{

Uint32 numRes;

PlatformRes_t res[SATA_CONFIG_MAX_NUM_AHCI_HOST];

Uint32 pscPwrmEnable; // (1 – enable power management, 0 – disable power management)

}PlatformResource;

To enable Power management feature : set one to the pscPwrmEnable member of Sata PlatformResource structure while calling PSP_sataDrvInit() call during initialization. The power management can be controlled by either through PWRM module or through PSC module. To enable PWRM Power management control define BIOS_PWRM_ENABLE option while compiling SATA library. To use PSC controller power management don't define the BIOS_PWRM_ENABLE.

To disable Power management feature: set zero to the pscPwrmEnable member of Sata PlatformResource structure while calling PSP_sataDrvInit() call during initialization. This will disable the power management feature for SATA driver.

16.3 SATA Sample applications

The SATA sample application is not provided as part of BIOSPSP release. Please refer to File system package application example.

Note : While linking the sata library with the application, the user need to create a PRD objects PRD0 and PRD1 in the .tcf associated with SATA application. Please include the following statement in the tcf file.

```
bios.PRD.create("PRD0");
bios.PRD.instance("PRD0").order = 1;
bios.PRD.instance("PRD0").comment = "sata timer0 ";
bios.PRD.instance("PRD0").period = 1000;
bios.PRD.instance("PRD0").mode = "one-shot";
bios.PRD.instance("PRD0").fxn = prog.extern("_sataTimer0_task", "asm");
bios.PRD.create("PRD1");
bios.PRD.instance("PRD1").order = 2;
bios.PRD.instance("PRD1").comment = "sata timer1 task";
bios.PRD.instance("PRD1").period = 1000;
bios.PRD.instance("PRD1").mode = "one-shot";
bios.PRD.instance("PRD1").fxn = prog.extern("_sataTimer1_task", "asm");
bios.PRD.create("PRD2");
bios.PRD.instance("PRD2").order = 2;
bios.PRD.instance("PRD2").comment = "sata timer2 task";
bios.PRD.instance("PRD2").period = 1000;
bios.PRD.instance("PRD2").mode = "one-shot";
bios.PRD.instance("PRD2").fxn = prog.extern("_sataTimer2_task", "asm");
bios.PRD.create("PRD3");
bios.PRD.instance("PRD3").order = 2;
bios.PRD.instance("PRD3").comment = "ata pwrm task";
```

```
bios.PRD.instance("PRD3").period = 100;
bios.PRD.instance("PRD3").mode = "one-shot";
bios.PRD.instance("PRD3").fxn = prog.extern("_ataPwrmTask", "asm");
```

16.4 Known Issues

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

16.5 Limitations

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

17 VPIF driver

17.1 Introduction

This document is the reference guide for Vpif device driver explaining the features and guidelines for using the driver.

DSP/BIOS™ applications use the driver typically through FVID APIs to perform frame video capture and display. FVID was implemented as a simple wrapper on top of the GIO class driver and provides an application-specific interface that has been customized for frame video. For more information on the DSP/BIOS™ device driver model and the GIO class driver, refer to the references section of this document.

The following sections describe in detail, the procedures how to configure and use the driver. It is recommended to go through the sample application to get familiar with initializing and using the Vpif driver.

17.1.1 Key Features

- Supports Multiple VPIF channels (2 capture and 2 display channels are supported on C6748 EVM)
- Supports dual channel 8-bit BT.656 capture and single channel 8, 10 or 12-bit RAW capture.
- Supports dual channel 8-bit BT.656 display.
- External Device Control Interface using EDC driver for seamless integration with different video encoder or decoder devices
- Supports flipping/exchange of multiple frame buffers for seamless capture and display operation
- Easy to maintain & re-target to new platforms

Features supported and verified on EVM:

- SD capture using channel 0 with input interface as Composite
- SD capture using channel 1 with input interface as S-video
- RAW capture using channel 0 with MT9T001 sensor
- SD display using channel 2 with input interfaces as either Composite or S-video but not both at the same time.
- Slice VBI capture and display using Closed Caption service for NTSC.

Features supported but not tested on EVM due to H/W limitation:

- SD display using channel 3
- HD capture
- HD display
- RAW VBI capture/display
- RAW HBI capture/display

Features which are not supported:

- RAW display
- ED capture and display
- Simultaneous RAW and SD capture

17.1.2 Terms and Abbreviations

Term	Description
⚑	This bullet indicates important information. Please read such text carefully.
❑	This bullet indicates additional information.
API	Application Programmer's Interface
CC	Closed Caption
CGMS	Copy generation management system
CSL	TI Chip Support Library – primitive h/w abstraction
EDC	External Device Control
HD	High Definition
IOM	Input / Output Module
IP	Intellectual Property
ISR	Interrupt Service Routine
C6748	TI's digital multi-media processor with C674x core
OS	Operating System
SD	Standard definition
SOC	System on chip
VPIF	Video Port Interface
WSS	Wide screen signaling

17.1.3 References

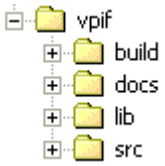
1.	spru403o.pdf	DSP/BIOS™ Driver Developer's Guide
2.	BIOSPSP_VPIF_Driver_Design.doc	VPIF design document
3.	sprugj9.pdf	VPIF H/W Controller
4.	BIOSPSP_vpif.chm	VPIF chm
5.	BIOSPSP_vpifedc.chm	VPIF Edc chm

17.2 Installation

The Vpif device driver is a part of PSP product for C6748 and would be installed as part of product installation.

17.2.1 PSP Component folder

On installation of the PSP package for C6748, the Vpif driver can be found at <ProjectDir>\ti\pspiom\vpif



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

- **vpif –** This top level vpif folder is the place holder for the entire Vpif driver. This folder contains `Vpif.h`, `Edc.h` and `Fvid.h`, the header files included by the application.
- **build –** This folder contains CCS 3.3 / CCS 4 VPIF driver library project file to build Vpif library. The generated driver library shall be included in the application where Vpif driver have to be used.
- **docs –** This folder contains design document and doxygen generated API reference help file. Design document contains the driver details which can be helpful for the developers as well as consumers to understand the driver design.
- **lib –** This folder contains vpif libraries generated in all the configuration modes (debug, idebug, irelease and release).
- **src –** This folder contains Vpif driver source files. It also contains header files that are used by the driver.

17.2.2 EDC Component folder

On installation of the PSP package for C6748, the Edc driver can be found at <ProjectDir>\pspiom\platforms\evmC6748\vpifedc

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

- **vpifedc –** This top level vpifedc folder is the place holder for the EDC driver. This folder contains `Adv7343.h`, `Mt9t001.h` and `Tvp5147.h`, the header files included by the application.
- **build –** This folder contains CCS 3.3 / CCS 4 EDC driver library project file to build EDC library. The generated EDC driver library shall be included in the application where EDC driver have to be used.
- **docs –** This folder contains the doxygen generated API reference help file.
- **lib –** This folder contains EDC libraries generated in all the configuration modes (debug and release).
- **src –** This folder contains EDC driver source files. It also contains header files that are used by the EDC driver. This contains the EDC source code for TVP5147 decoder, MT9T001 sensor and ADV7343 encoder. Codec interface related code is also present here.

17.2.3 Build Options

The Vpif library can be built using the CCS v3.3 project file located at <ProjectDir>\packages\ti\pspiom\vpif\build\C6748\ccs3\vpif.pjt. The EDC library can be built using the CCS v3.3 project file located at <ProjectDir>\packages\ti\pspiom\platforms\evm6748\vpifedc\build\ccs3\vpifedc.pjt.

The project file supports the following build configurations:

IMPORTANT NOTE:

Instrumentation code in iDebug and iRelease pjts is not implemented and is for future implementation. They are same as Debug and Release pjts.

Debug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6748" to build library for C6748 soc.

iDebug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6748" to build library for C6748 soc.

Release:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6748" to build library for C6748 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_C6748" to build library for C6748 soc.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver

17.2.3.1 Required and Optional Pre-defined symbols

This driver does not have any specific build option. The Vpif library must be built with a soc specific pre-defined symbol.

"-DCHIP_C6748" is used above to build for EVM C6748. Internally this define is used to select a soc specific header file (`soc_C6748.h`). This header file contains information such as base addresses of VPIF device, its event numbers, etc.

The Vpif 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.

17.3 Features

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

17.3.1 Overview

Video Port Interface provides a flexible video input/output port which allows the capture and display of digital video streams. This device driver is written in conformance to the DSP/BIOS™ GIO model and handles communication to and from the VPIF device. VPIF has its own internal DMA for data handling.

The following decoders are used for various types of captures:

- Two TVP5147 decoders are connected to both channels via BT.656 interface. One TVP5147 decoder is connected to S-video input which provides BT.656 input to channel 1. The other TVP5147 decoder is connected to composite input which provides BT.656 input to channel 0.
- External MT9T001 sensor is connected to both the channels for RAW data capture.

The following encoder is used for various types of display:

- Single ADV7343 encoder for SD display. Encoder is connected to both S-video output and composite output which provides BT.656 output for channel 2.

17.3.2 Driver Component

The Video driver is constituted of following sub components:

VPIF Driver – application interface, VPIF and DMA handling

EDC (External Device Control) Driver – Configures external Video Decoder and Encoder. VPIF driver library calls EDC Driver APIs for external Decoder and Encoder configurations.

The block diagram below shows the overall system architecture:

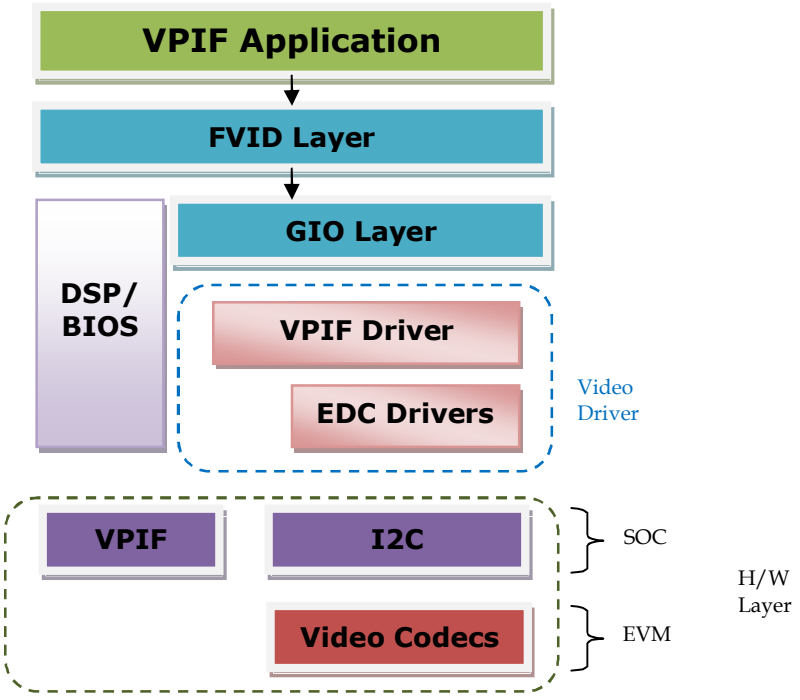


Figure 1. VPIF Driver Architecture

Vpif driver lies below the FVID and GIO layer. The driver uses the DSP BIOS™ APIs for OS services. The main function of the Vpif driver is to program the peripherals, for the display or capture configuration, to move the video data to and from SDRAM

to the VPIF interface. The Vpif driver actually captures and displays the video data. The VPIF channel data format is selectable based on the settings of the specific channel control register (Channels 0-3). The EDC drivers are used to configure the encoders and decoders, using codec interface. The call to EDC drivers is always through the Vpif layer.

All channels can be activated simultaneously for SD mode

- Channels 0 and 1 are prepared only for capture.
- Channels 2 and 3 are prepared only for display.

Display applications can access VPIF channel 2 and channel-3 through software interfaces. Both the channels support SD display. Using EDC interface encoder is configured. Display Driver supports the following standards:

- SD output display: NTSC 480i 30 fps and PAL 576i 25 fps.

Capture applications can access VPIF channel 0 and channel 1 through software interfaces. Both the channels support SD capture but only channel-0 supports RAW capture. Channel 0 and 1 are used simultaneously for raw video capture using sensor device. Using EDC interface decoder and sensor is configured. Capture Driver supports the following standards:

- Raw input capture
- SD input capture: NTSC 480i 30 fps and PAL 576i 25 fps

This driver is not tested for HD because of Hardware constraints. However the driver is designed keeping HD in mind.

The following figure shows the physical connections for TVP5147 decoders on EVM.

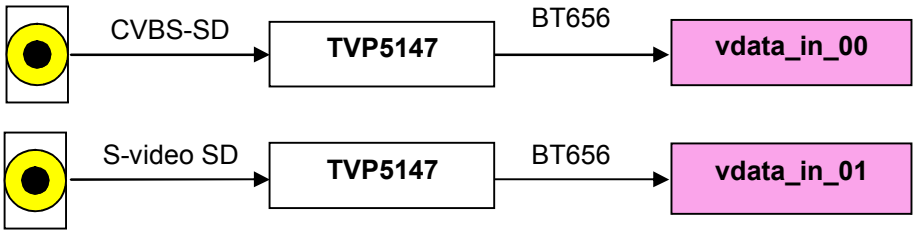


Figure 2. Physical input interface for SD on EVM

The following figure shows the physical connections for ADV7343 encoder on EVM.

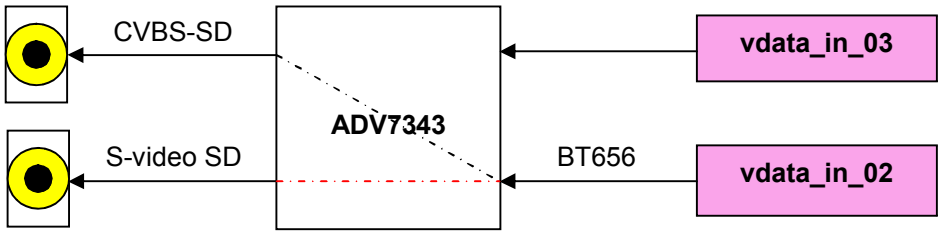


Figure 3. Physical output interface for SD on EVM

17.3.3 Driver Capabilities

Following are some of the capabilities of VPIF driver:

1. The driver conforms to IOM model of DSP/BIOS™ operating system.

2. For field mode, each IO request to the driver would require both fields' data of a frame. For capture, the driver completes the IO request once a frame is captured or both the fields are captured. For display, the driver completes the IO request once a frame is displayed or both the fields are displayed.
3. Supports dynamic switching among input interfaces and various resolutions with some necessary restrictions wherever applicable.
4. The driver will expose 4 software channels. Two capture channel for each of the hardware channel 0 and 1. Two software channels of display for each of the display hardware channels 2 and 3. All the software channels will support SD (BT656) mode but only channel 0 will support RAW capture.
5. The SD capture/display channel will support the following resolutions for BT stream:
    - NTSC 480i at 30fps
    - PAL 576i at 25fps
6. Capture driver
    - Always returns the most recent frame.
    - Cycle through available buffers when application falls behind.
7. Display driver
    - Queues buffers for displaying from application.
    - Keep displaying the same frame when running out of buffers.
    - Returns the IO request/buffer immediately after displaying the content of that IO request, if an IO request is pending.
8. The decoder EDC driver will support runtime change of the following parameters:

    TVP5147: SD BRIGHTNESS, SD CONTRAST, SD SATURATION, SD HUE and SD AUTOGAIN
9. The encoder EDC driver will support runtime change of the following parameters:

    ADV7343: SD BRIGHTNESS, SD HUE, and SD GAMMA.
10. Raw Ancillary data capture/display is supported by VPIF driver provided the same is supported by encoder and decoder. This is not tested due to EVM limitations.
11. VBI capture/display in the slice mode will be provided for closed caption, WSS and CGMS. Decoder TVP5147 and encoder ADV7343, available on EVM, will be used for this purpose.
12. VPIF driver will not allocate frame buffers for driver operations. Applications have to create buffers for this purpose. The API's for buffer allocation will be provided. It is recommended that applications should use the APIs provided with driver for frame buffer allocation/de-allocation purpose.
13. Minimum three buffers are required to be queued inside the Vpif driver before the driver is ready to start capture or display operation. A minimum of 3 frame buffers should be used for proper operation

17.3.4 Driver Limitations

Following are the constraints of the VPIF driver:

1. HD capture will not be supported.

2. HD and RAW display will not be supported.
3. Simultaneous RAW and SD capture would not be supported by the driver.
4. Raw video capture will be supported provided EVM has support for same i.e. there should be sensor (for e.g. external MT9T001 image sensor) to capture RAW data.
5. As SD mode is supported by vpif driver, only SD parameters are configured in the encoder and decoder.
6. Dynamic switching of resolution and dynamic switching of interfaces is not supported when streaming is on.
7. VPIF input/output buffer addresses must be multiple of eight.
8. FVID_EXCHANGE mechanism should be used for exchanging pointers between buffers.
9. Raw VBI and raw HBI is supported by the driver but not tested.

This driver is not tested for HD because of Hardware constraints. However the driver is designed keeping HD in mind.

17.3.5 Capture and / or display operation

Vpif driver can be simultaneously operated as a capture and or display. This could be achieved by creating a channel as an INPUT channel and creating another channel as an OUTPUT channel. The type of channel is specified while creating the channel (using FVID_create() specify "`IOM_OUTPUT`" or "`IOM_INPUT`").

Application can send the mode in which the channel should be opened by making "`dispStdMode`" or "`capStdMode`" member of channel parameters as any of the `Vpif_VideoMode` enum. The driver will look for this mode internally in the lookup table and update the internal `Vpif_ConfigParams` structure. The "`capVideoParams`" or "`dispVideoParams`" member of channel parameter should be NULL. Application can also choose to send these parameters. If the "`capVideoParams`" or "`dispVideoParams`" parameter is not NULL, driver will update the internal `Vpif_ConfigParams` structure using the parameters given by application.

17.4 VPIF Configurations

This section discusses about the initialization details and structures used in the VPIF driver. Please note that for some structure member information/details, the C6748 VPIF peripheral reference guide might need to be referred.

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

Following section document some of the configurable parameter of Vpif. Please refer to `Vpif.h` for complete configurations and explanations.

17.4.1 Initialization details

To use the capture or display channel of Vpif device driver, a device entry must be added and configured in the DSP/BIOS configuration tool.

To have Vpif device driver included in the application, corresponding TCI file have to be included in BIOS TCF i.e. "`vpifSample.tci`" must be included in BIOS TCF file of the application for using VPIF instance 0 of the driver. This file can be found in vpif sample application directory.

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

TCI Configuration Parameters	Description
initFxn - Init Function	Pointer to application function to initialize C6748 VPIF and configure parameters like DMA size and the HWI number. This will override the default parameters inside the driver. VPIF requires that the user call Vpif_init() as part of this initFxn. Users can also directly hook in Vpif_init().
fxnTable - Function Table Pointer	Pointer to IOM function table. Vpif requires this field to be `Vpif_IOMFXNS`. This is a global variable which points to the VPIF driver APIs.
fxnTableType - Function Table Type	IOM_Fxns.
deviceId - Device Id	Specify which VPIF instance to use. For example to use VPIF on C6748 this should be given as 0.
params – Pointer to Port parameter	A pointer to an object of type `Vpif_Params` as defined in the header file `Vpif.h`. This pointer will point to a device parameter structure. In BIOS TCI files, this structure object is passed as an argument. Application should declare and initialize the structure object properly.

The vpif driver initialization in BIOS TCF looks like this:

```
bios.UDEV.create("VPIF0");
bios.UDEV.instance("VPIF0").fxnTableType = "IOM_Fxns";
bios.UDEV.instance("VPIF0").initFxn = prog.extern("userVpif0Init");
bios.UDEV.instance("VPIF0").params = prog.extern("vpifParams");
bios.UDEV.instance("VPIF0").fxnTable = prog.extern("Vpif_IOMFXNS");
bios.UDEV.instance("VPIF0").deviceId = 0x0;
```

Apart from the VPIF driver initialization, I2C driver should also be initialized in the BIOS TCF file. For details on how to initialize I2C driver, refer I2C driver user guide and/or the sample application provided with the package.

17.4.2 Constants & Enumerations

17.4.2.1 Define for Vbi service

```
/* VBI Ancillary Data service: NONE. No Ancillary Data is required */

#define Vpif_VbiServiceType_NONE            0x0

/* VBI Ancillary Data service: Horizontal Ancillary (HANC) - Data between EAV
and SAV (horizontal blanking interval) */

#define Vpif_VbiServiceType_HBI             0x1u

/* VBI Ancillary Data service: Vertical Ancillary (VANC) - Data between SAV
and EAV (horizontal active video area). */

#define Vpif_VbiServiceType_RAW_VBI         0x2u
```

```
/* VBI Ancillary Data service: Specific Ancillary Data. ancillary data that
is not video image data but is VBI data. */

#define Vpif_VbiServiceType_SLICE_VBI       0x4u
```

These are defined for different VBI services supported by VPIF. A valid value for this for a particular channel operation should be passed to channel parameters in the "vbiService" field.

17.4.2.2 Vpif_IOCTL

```
typedef enum Vpif_IOCTL_t

{

    Vpif_IOCTL_CMD_START,

    /**< Start the VPIF channel operation. */

    Vpif_IOCTL_CMD_STOP,

    /**< Stop the VPIF channel operation. */

    Vpif_IOCTL_CMD_GET_NUM_IORQST_PENDING,

    /**< Get number of pending I/O requests in the driver queue. */

    Vpif_IOCTL_CMD_GET_CHANNEL_STD_INFO,

    /**< Get the current configuration parameters of driver. */

     Vpif_IOCTL_CMD_CHANGE_RESOLUTION,

    /**< Change the current resolution of the channel. */

    Vpif_IOCTL_CMD_MAX

    /**< Book-keep - Max ioctl's */

}Vpif_IOCTL;
```

This enum defines the different IOCTL commands used to perform control operation on VPIF. They are common for both capture and display operation. The IOCTL command is passed as second argument in vpifMdControlChan() function when the driver is used directly with the application. These commands are explained in detail during FVID_control() function explanation.

17.4.2.3 Vpif_SdramStorage

```
typedef enum Vpif_SdramStorage_t

{

    Vpif_SdramStorage_FIELD = 0,

    /**< VPIF field format storage: field 1 and field 2 will be stored

     *   separately.*/

    Vpif_SdramStorage_FRAME

    /**< VPIF frame format storage: field 1 and field 2 will be stored in

     *   merged pattern i.e. one line of field 1, one line of field 2.
```

```
     *   CAUTION: For Progressive mode SDRAM storage should be Frame ONLY.*/
}Vpif_SdramStorage;
```

This enum defines the different storage modes of operation. Progressive video must use the frame storage mode, but interlaced video can use either field or frame storage modes.

17.4.2.4 Vpif_VideoMode

```
typedef enum Vpif_VideoMode_t
{
    Vpif_VideoMode_NONE = 0,
    /**< VPIF operation mode: NONE. Used when user wants to send thedifferent
     *   video parameters and do not want to use internal look-up table.*/
    Vpif_VideoMode_NTSC,
    /**< VPIF operation mode: NTSC - 480 I Video Standard*/
    Vpif_VideoMode_PAL,
    /**< VPIF operation mode: PAL - 576 I Video Standard*/
    Vpif_VideoMode_RAW_VGA,
    /**< VPIF operation mode: Raw Mode - Bayer Pattern GrRBGb only*/
    Vpif_VideoMode_RAW_SVGA,
    /**< VPIF operation mode: Raw Mode - Bayer Pattern GrRBGb only*/
    Vpif_VideoMode_RAW_XGA,
    /**< VPIF operation mode: Raw Mode - Bayer Pattern GrRBGb only*/
    Vpif_VideoMode_RAW_SXGA,
    /**< VPIF operation mode: Raw Mode - Bayer Pattern GrRBGb only*/
    Vpif_VideoMode_RAW_UXGA,
    /**< VPIF operation mode: Raw Mode - Bayer Pattern GrRBGb only*/
    Vpif_VideoMode_RAW_QXGA,
    /**< VPIF operation mode: Raw Mode - Bayer Pattern GrRBGb only*/
    Vpif_VideoMode_RAW_480P,
    /**< VPIF operation mode: Raw Mode - Bayer Pattern GrRBGb only*/
    Vpif_VideoMode_RAW_576P,
    /**< VPIF operation mode: Raw Mode - Bayer Pattern GrRBGb only*/
    Vpif_VideoMode_RAW_720P,
    /**< VPIF operation mode: Raw Mode - Bayer Pattern GrRBGb only*/
    Vpif_VideoMode_RAW_1080P
```

```
    /**< VPIF operation mode: Raw Mode - Bayer Pattern GrRBGb only*/
}Vpif_VideoMode;
```

This enum defines the different video modes of operation.

- *Some of the RAW mode may or may not apply, and will depend on the type of image sensor used.*

17.4.2.5 Vpif_RawCapturePinPol

```
typedef enum Vpif_RawCapturePinPol_t
{
    Vpif_RawCapturePinPol_SAME = 0,
    /**< No inversion.                                                */
    Vpif_RawCapturePinPol_INVERT
    /**< Invert incoming signal inside the VPIF.                      */
}Vpif_RawCapturePinPol;
```

This enum defines the polarity of external control signal for raw capture.

17.4.2.6 Vpif_RawCaptureDataWidth

```
typedef enum Vpif_RawCaptureDataWidth_t
{
    Vpif_RawCaptureDataWidth_8BITS = 0,
    /**< 8 bits/pixel                                                */
    Vpif_RawCaptureDataWidth_10BITS,
    /**< 10 bits/pixel                                               */
    Vpif_RawCaptureDataWidth_12BITS
    /**< 12 bits/pixel                                               */
}Vpif_RawCaptureDataWidth;
```

This enum defines the data width for the raw capture mode.

17.4.2.7 Vpif_DmaReqSize

```
typedef enum Vpif_DmaReqSize_t
{
    Vpif_DmaReqSize_32BYTE,
    /**< Request size of 32 bytes                                    */
    Vpif_DmaReqSize_64BYTE,
    /**< Request size of 64 bytes                                    */
    Vpif_DmaReqSize_128BYTE,
    /**< Request size of 128 bytes                                   */
```

```
    Vpif_DmaReqSize_256BYTE
    /**< Request size of 256 bytes                                          */
}Vpif_DmaReqSize;
```

This enum defines the request size settings for DMA transfer.

17.4.2.8 Vpif_FrameFormat

```
typedef enum Vpif_FrameFormat_t
{
    Vpif_FrameFormat_INTERLACED,
    /**< Interlaced frame format                                            */
    Vpif_FrameFormat_PROGRESSIVE
    /**< Progressive frame format                                           */
}Vpif_FrameFormat;
```

This enum keeps track of kind of the frame format. VPIF supports both interlaced and progressive video formats.

17.4.2.9 Vpif_YCMuxed

```
typedef enum Vpif_YCMuxed_t
{
    Vpif_YCMuxed_NO,
    /**< For BT.656 video, luminance (Y) and chrominance (C) values are
     *   multiplexed into a single byte-stream on one channel.             */
    Vpif_YCMuxed_YES
    /**< For BT.1120 video, channels function as a pair without Y/C
     *   multiplexing.                                                     */
}Vpif_YCMuxed;
```

This enum keeps track of Y/C streams are muxed or not.

17.4.2.10 Vpif_CaptureFormat

```
typedef enum Vpif_CaptureFormat_t
{
    Vpif_CaptureFormat_BT,
    /**< BT.xxx  The BT/YC video mode will look for video sync signals that
     *   are embedded within the video byte stream (standard for BT video).*/
    Vpif_CaptureFormat_CCDC
    /**< The CCD/CMOS (Raw Data Capture) mode will look for video syncsignals
     *   on the dedicated VPIF sync pins (common for CCD and CMOS sensors).*/
```

```
}Vpif_CaptureFormat;
```

This enum keeps track of capture format.

17.4.2.11 Vpif_IoMode

```
typedef enum Vpif_IoMode_t
{
    Vpif_IoMode_NONE,
    /**< No operation selected */
    Vpif_IoMode_RAW_CAP,
    /**< Raw mode of Capture */
    Vpif_IoMode_CAP,
    /**< BT mode of Capture */
    Vpif_IoMode_DIS
    /**< Display mode of operation */
}Vpif_IoMode;
```

This enum defines the mode for channel operation. When a channel is opened, this enum defines the IO mode for which the channel is opened.

For display operation "`mode`" parameter passed to FVID_create() is `IOM_OUTPUT` and, only `Vpif_IoMode_DIS` is the I/O mode supported. For capture operation "`mode`" parameter passed to FVID_create() is `IOM_INPUT` and the channel I/O mode can be BT capture or RAW capture decided by `Vpif_IoMode_CAP` and `Vpif_IoMode_RAW_CAP` respectively, passed by application.

17.4.2.12 Vpif_PllDomain

```
typedef enum Vpif_PllDomain_t
{
    Vpif_PllDomain_0 = 0,
    /**< PLL domain 0       */

    Vpif_PllDomain_1 = 1
    /**< PLL domain 1       */
}Vpif_PllDomain;
```

This enum keeps track of the PLL domain where the VPIF device lies.

17.4.3 Data Structures

17.4.3.1 Vpif_RawVbiParams

"`Vpif.h`" file contains `Vpif_RawVbiParams` data structure, which is a part of `Vpif_ConfigParams` structure. This structure will store vpif parameters for raw

vbi/hbi data for capture/display. This is used to calculate the size of raw vbi and raw hbi buffers. The members of this structure are explained below:

Structure Members	Description
samplePerLine	Byte count of valid data within the ancillary blanking region.
countFld0	Line count of valid top field ancillary data.
countFld1	Line count of valid bottom field ancillary data.

17.4.3.2 Vpif_RawSelectiveVbiParams

"*Vpif.h*" file contains *Vpif_RawSelectiveVbiParams* data structure, which is a part of *Vpif_DisChanParams* structure. This structure will store vpif parameters for raw vbi/hbi data when VPIF **SELECTIVELY** wants to display sub-regions in the VBI space. The VPIF **can selectively transmit** sub-regions in the VBI space but **cannot selectively receive** sub-regions in the VBI space.

Note that the user is expected to place valid ancillary data in a memory buffer that is representative of the entire VBI region of interest. However, only the valid ancillary data region needs to be initialized -- the VPIF will automatically transmit blanking data (Y=10h, C=80h) for non-valid ancillary data regions.

The members of this structure are explained below:

Structure Members	Description
vbi0StrtHps	Horizontal start of vbi data for first field. Horizontal position (byte-count) of valid data within the top field horizontal ancillary blanking region. Byte positions are enumerated beginning with 0. The value of HPOS must be a multiple of 8.
vbi0StrtVps	Vertical start of vbi data for first field. Vertical position (line-count) of valid data within the top field horizontal ancillary blanking region. Line positions are enumerated beginning with 1.
vbi0Hsz	Horizontal size of vbi data for first field. Horizontal size (byte-count) of valid top field horizontal ancillary data beginning at vbi0StrtHps. The value of HSIZE must be a multiple of 8.
vbi0Vsz	Vertical size of vbi data for first field. Vertical size (line-count) of valid top field horizontal ancillary data beginning at vbi0StrtVps.
vbi1StrtHps	Horizontal start of vbi data for second field. Horizontal position (byte-count) of valid data within the bottom field horizontal ancillary blanking region. Byte positions are enumerated beginning with 0. The value of HPOS must be a multiple of 8.
vbi1StrtVps	Vertical start of vbi data for second field. Vertical position (line-count) of valid data within the bottom field horizontal ancillary blanking region. Line positions are enumerated beginning with 1.
vbi1Hsz	Horizontal size of vbi data for second field. Horizontal size

	(byte-count) of valid bottom field horizontal ancillary data beginning at vbi1StrtHps. The value of HSIZE must be a multiple of 8.
vbi1Vsz	Vertical size of vbi data for second field. Vertical size (line-count) of valid bottom field horizontal ancillary data beginning at vbi1StrtVps.

17.4.3.3 Vpif_ConfigParams

"*Vpif.h*" file contains *Vpif_ConfigParams* data structure that is passed as a part of channel parameters - *Vpif_CapChanParams* and *Vpif_DisChanParams*. Most members of this structure directly reflect the VPIF register settings. The members of this structure are explained below:

Structure Members	Description
mode	Video Standard mode. Video mode defined by enum *Vpif_VideoMode*. If the mode is not defined in enum *Vpif_VideoMode*, "*mod*e" should be *Vpif_VideoMode_NONE*.
width	Indicates width of the image for this mode
height	Indicates height of the image for this mode. Active lines.
fps	Indicates frames per sec for this mode. This member is not used by Vpif internally and is for information purpose.
frameFmt	Indicates whether this is interlaced or progressive format. This value should be *Vpif_FrameFormat_INTERLACED* or *Vpif_FrameFormat_PROGRESSIVE* depending on required operation.
ycMuxMode	Indicates whether this mode requires single or two channels. This value should be *Vpif_YCMuxed_NO* or *Vpif_YCMuxed_YES* depending on required operation.
eav2sav	The number of bytes in the inactive (EAV2SAV) video regions. The EAV2SAV value must be even.
sav2eav	The number of bytes in the active (SAV2EAV) video regions. The SAV2EAV value must be even.
l1	Enumerated line number for the L1 field position.
l3	Enumerated line number for the L3 field position.
l5	Enumerated line number for the L5 field position.
l7	Enumerated line number for the L7 field position. Note that L7 is not used with the progressive video mode.
l9	Enumerated line number for the L9 field position. Note that L9 is not used with the progressive video mode.
l11	Enumerated line number for the L11 field position. Note that L11 is not used with the progressive video mode.
vsize	Vertical size of the image. Actual lines.
captureFormat	Indicates whether capture format is in BT or in CCD/CMOS. This value should be *Vpif_CaptureFormat_BT* or

	Vpif_CaptureFormat_CCDC depending on required operation.
isVbiSupported	Indicates whether this mode supports capturing vbi or not. Boolean: TRUE = VBI mode is supported by this video mode. FALSE = VBI mode is not supported by this video mode.
isHd	Indicates whether this mode is HD or not. Boolean: TRUE = HD mode. FALSE = not HD mode. Kept for future use.
hancOffset	Offset for the horizontal ancillary data.
rawHbiParams	Raw non selective HBI params.
rawVbiParams	Raw non selective VBI params.

- *For CCDC format many of the members are not used. Please refer to the VPIF peripheral reference guide for detail. Following is an example:*

```
/* RAW parameters for VGA mode */
Vpif_ConfigParams rawParamEx = {Vpif_VideoMode_RAW_VGA, 640, 480, 93, Vpif_FrameFormat_PROGRESSIVE, Vpif_YCMuxed_NO, 0, 0, 0, 0, 0, 0, 0, 0, 0, Vpif_CaptureFormat_CCDC, FALSE, FALSE, 0, {0, 0, 0}, {0, 0, 0}};
```

- *"hancOffset", "rawHbiParams", "rawVbiParams" are valid only if vbi is supported by the video mode and isVbiSupported is set to TRUE.*
- *The driver does not checks the validity of individual parameters*

17.4.3.4 Vpif_StdInfo

"*Vpif.h*" file contains *Vpif_StdInfo* data structure that is passed while *Vpif_IOCTL_CMD_GET_CHANNEL_STD_INFO* call. The members of this structure are explained below:

Structure Members	Description
stdMode	Current video mode of driver. Video mode defined by enum *Vpif_VideoMode*.
activePixels	Same as bytes per line or width
activeLines	Same as height
framePerSec	Frames per second
stdFrameFormat	Frame format – Interlaced or Progressive
stdVbiService	Indicates what all VBI services supported by this mode. Available values for this field are defined in "*Vpif.h*" file with VPIF VBI Ancillary Data service title.
sdramStorage	SDRAM storage mode. This value should be *Vpif_SdramStorage_FIELD* or *Vpif_SdramStorage_FRAME*

	depending on required operation.

17.4.3.5 Vpif_FrameBufferParams

"*Vpif.h*" file contains *Vpif_FrameBufferParams* data structure that is passed as a part of channel parameters - *Vpif_CapChanParams* and *Vpif_DisChanParams*. This structure tells about the alignment of frame buffer and the segment id from which the buffers will be allocated. The members of this structure are explained below:

Structure Members	Description
frmBufAlignment	Frame buffer alignment used by driver while allocating memory for video frame buffer
frmBufSegId	Memory segment ID, used by driver to allocate video frame buffer

17.4.3.6 Vpif_CapChanParams

"*Vpif.h*" file contains *Vpif_CapChanParams* data structure that is passed while FVID_create() call. Applications could use this structure to configure the channel specific configurations. Most members of this structure directly reflect the VPIF register settings. The driver **does not** check the validity of these parameters (Example videoParams, dataSize etc). Please refer VPIF peripheral reference guide for more details. The members of this structure are explained below:

Structure Members	Description
capStdMode	Operation mode title. Video mode defined by enum *Vpif_VideoMode*. If the value of this mode is *Vpif_VideoMode_NONE*, it suggests that user do not want to use internal lookup table for video parameters.
capChannelIoMode	Operation mode for which the channel is opened. Channel IO mode is defined by enum *Vpif_IoMode*.
capFbParams	Frame buffer settings defined by *Vpif_FrameBufferParams*
capStorageMode	Indicates whether it is field or frame based storage mode. This is only applicable for interlaced mode of operation.
*capEdcTbl	Function table of decoder module for the channel. A statically defined EDC function table is passed to the vpifMdCreateChan() function via the channel parameters argument. Refer to External Device Control section for details.
*capVideoParams	Specify the Video parameters if application would like to specify them. This is an optional parameter. If not used, set this element to NULL. If set to NULL, the driver will read the video parameters depending upon the "*capStdMode*" set. If it is not NULL, its value will prevail over whatever mode being set. In this case the mode parameter in "*capVideoParams*" should be *Vpif_VideoMode_NONE*. CAUTION: If wrong parameters are sent, the driver does not verify the validity of these parameters
capVbiService	Indicates what type VBI services are required by this

	mode. Available values for this field are defined in "*Vpif.h*" file with VPIF VBI Ancillary Data service title.
capVbiSliceService	If the VBI type is Slice VBI then what kind of service it is. Valid only if one of the "*capVbiService*" is set as *Vpif_VbiServiceType_SLICE_VBI*. Whatever slice service is set here only that data is captured. Available values for this field are defined in "*Fvid.h*" file with FVID Slice VBI service type title.
capDataSize	The data width bit is only used with the CCD/CMOS data capture mode. Data size defined by enum *Vpif_RawCaptureDataWidth*.
capFieldPol	Field ID polarity inverting control. This value should be *Vpif_RawCapturePinPol_SAME* or *Vpif_RawCapturePinPol_INVERT* depending on required operation.
capVPixPol	Vertical pixel valid signal polarity control. Same as "*capFieldPol*".
capHPixPol	Horizontal pixel valid signal polarity control. Same as "*capFieldPol*".

- *"capDataSize", "capFieldPol", "capVPixPol", "capHPixPol" are only valid for RAW capture mode they are not valid for BT mode.*
- *"capVbiService", "capVbiSliceService" are only valid for BT capture they are not valid for RAW capture mode. Ancillary data is only supported for BT byte streams.*
- *If "capEdcTbl" is passed as NULL, the driver will not throw any error and it is assumed that there is no EDC available for that channel.*
- *Setting "capStdMode" as Vpif_VideoMode_NONE and "videoParams" as NULL in channel parameters will results in error from the driver.*

17.4.3.7 Vpif_DisChanParams

"*Vpif.h*" file contains *Vpif_DisChanParams* data structure that is passed while FVID_create() call. Applications could use this structure to configure the channel specific configurations. Most of the members of this structure directly reflect the VPIF register settings. The driver **does not** check the validity of these parameters (Example *videoParams*, *vVbiParams* etc). Please refer to VPIF peripheral reference guide for more details. The values to be used for most of the members are given in "*Vpif.h*" file. The members of this structure are explained below:

Structure Members	Description
dispStdMode	Operation mode title. Video mode defined by enum *Vpif_VideoMode*. If the value of this mode is *Vpif_VideoMode_NONE*, it suggests that user do not want to use internal lookup table for video parameters.
dispChannelIoMode	Operation mode for which the channel is opened. Channel IO mode is defined by enum *Vpif_IoMode*.
dispFbParams	Frame buffer settings defined by *Vpif_FrameBufferParams*

	structure.
dispStorageMode	Indicates whether it is field or frame based storage mode. This is only applicable for interlaced mode of operation.
*dispEdcTbl	Function table of decoder module for the channel. A statically defined EDC function table is passed to the vpifMdCreateChan() function via the channel parameters argument. Refer to section External Device Control section for details. If NULL is passed then it is assumed that there is no EDC available for that channel.
*dispVideoParams	Specify the Video parameters if application would like to specify them. This is an optional parameter. If not used, set this element to NULL. If set to NULL, the driver will read the video parameters depending upon the "*dispStdMode*" set. If it is not NULL, its value will prevail over whatever mode being set. In this case the mode parameter in "*dispVideoParams*" should be *Vpif_VideoMode_NONE*. CAUTION: If wrong parameters are sent, the driver does not verify the validity of these parameters
dispVbiService	Indicates what type VBI services are required by this mode. Available values for this field are defined in "*Vpif.h*" file with VPIF VBI Ancillary Data service title.
dispVbiSliceService	If the VBI type is Slice VBI then what kind of service it is. Valid only if one of the "*dispVbiService*" is set as *Vpif_VbiServiceType_SLICE_VBI*. Whatever slice service is set here only that data is displayed. Available values for this field are defined in "*Fvid.h*" file with FVID Slice VBI service type title.
*dispVVbiParams	Indicates the parameters for selective Vertical blanking data. Value of NULL suggests that selective sub-regions in the VBI space are not required. For selectively sub-regions in the VBI space this should hold appropriate value. The values are defined by *Vpif_RawSelectiveVbiParams* structure.
*dispHVbiParams	Indicates the parameters for selective Horizontal blanking data. Value of NULL suggests that selective sub-regions in the HBI space are not required. For selectively sub-regions in the VBI space this should hold appropriate value. The values are defined by *Vpif_RawSelectiveVbiParams* structure.

- *"dispVbiService", "dispVbiSliceService" are valid for BT display. Ancillary data is only supported for BT byte streams.*
- *If "dispEdcTbl" is passed as NULL, the driver will not throw any error and it is assumed that there is no EDC available for that channel.*
- *Setting both, "dispStdMode" as Vpif_VideoMode_NONE and "dispVideoParams" as NULL in channel parameters will results in error from the driver.*

17.4.3.8 Vpif_Params

"*Vpif.h*" file contains *Vpif_Params* data structure that is passed during vpifMdBindDev() call which is defined with UDEV VPIF parameters in TCF file of application. This structure defines the device configurations. The members of this structure are explained below:

Structure Members	Description
hwiNumber	HWI number associated with this device event. This is the HWI number application chooses to configure the ECM event (one of 0, 1, 2, 3) that is pertaining to the VPIF DSP interrupt event. The value of this depends on which ECM block the VPIF interrupt fall. Please note that no validation is done by the driver.
dmaReqSize	Request size for DMA data transfer from/to VPIF. Data size is either luminance or chrominance. DMA size defined by enum *Vpif_DmaReqSize*.
pscPwrmEnable	Boolean flag to enable (TRUE) or disable (FALSE) any power management in the driver
pllDomain	Pll domain where the device is

17.4.4 Interface Functions

17.4.4.1 Vpif_init

This function needs to be called as part of BIOS initialization by setting initFxn for at particular UDEV instance or by calling this function as part of user specific initFxn.

17.5 FVID Configurations

This section describes the functions, data structures, enumerations and macros for the FVID module. Please refer to *Fvid.h* for complete configurations and explanations. The following API functions are defined by the FVID module:

Function	Description
FVID_create	Initialize the VPIF channel object
FVID_delete	De-allocate an FVID channel object
FVID_control	Send device-specific control command to the mini-driver
FVID_exchange	Exchange an application-owned buffer for a driver-owned buffer
FVID_dequeue	Get a pointer of the frame buffer from driver to application.
FVID_queue	Relinquish the frame buffer back to the driver from application.
FVID_allocBuffer	Allocate a frame buffer using the driver's memory allocation routines.
FVID_freeBuffer	Free the buffer allocated via FVID_allocBuffer().

17.5.1 Constants & Enumerations

17.5.1.1 Define for IOM_Packet

```
/* IOM user defined command base address */
```

```
#define FVID_BASE          (IOM_USER)

/* Command for FVID_exchange to exchange buffers between Driver and
Application */

#define FVID_EXCHANGE      (FVID_BASE + 0)

/* Command for FVID_queue to submit a video buffer back to video device
driver */

#define FVID_QUEUE         (FVID_BASE + 1)

/* Command for FVID_dequeue to request the video device driver to give
ownership of a data buffer */

#define FVID_DEQUEUE       (FVID_BASE + 2)

/* Command for FVID_allocBuffer to request the video device driver to
allocate one data buffer */

#define FVID_ALLOC_BUFFER  (FVID_BASE + 3)

/* Command for FVID_freeBuffer to request the video device driver to free
memory of given data buffer */

#define FVID_FREE_BUFFER   (FVID_BASE + 4)
```

These are command codes used for FVID to GIO API conversion macros.

17.5.1.2 Define for Slice service

```
/* FVID Slice VBI Service: NONE */

#define Fvid_SLICE_VBI_SERVICES_NONE               0x0

/* FVID Slice VBI Service: Wide screen signaling (WSS) for PAL */

#define Fvid_SLICE_VBI_SERVICES_WSS_PAL            0x1u

/* FVID Slice VBI Service: Copy generation management system (CGMS)for NTSC*/

#define Fvid_SLICE_VBI_SERVICES_CGMS_NTSC          0x2u

/* FVID Slice VBI Service: Closed caption for NTSC */

#define Fvid_SLICE_VBI_SERVICES_CC_NTSC            0x4u

/* FVID Slice VBI Service: MAX */

#define Fvid_SLICE_VBI_SERVICES_MAX                3

/* Maximum data size for FVID Slice VBI data in bytes */

#define FVID_SLICE_VBI_DATA_SIZE_BYTES_MAX         4
```

This enumeration defines the different slice services supported by the VPIF driver.

17.5.1.3 Enum for Color format

```
typedef enum FVID_colorFormat_t

{

    FVID_YCbCr422_INTERLEAVED = 0,

    FVID_YCbCr422_PLANAR,
```

```
    FVID_YCrCb422_INTERLEAVED,
    FVID_YCbCr422_SEMIPLANAR_UV,
    /* YCbCr4:2:2 YC Semi Planar(YUV422UVP) */
    FVID_RGB_888_INTERLEAVED,
    FVID_RGB565_INTERLEAVED,
    FVID_DVD_MODE,
    FVID_CLUT_INDEXED,
    FVID_ATTRIBUTE,
    FVID_BAYER_PATTERN,
    FVID_RAW_FORMAT,
    FVID_COLORFORMAT_INVALID
}FVID_colorFormat;
```

The enumeration string itself is self explanatory of the color format. Only `FVID_YCbCr422_SEMIPLANAR_UV` is supported for BT video data (capture and display) and `FVID_RAW_FORMAT` format is supported for RAW video capture are supported.

- *VPIF supports BT video data in YCbCr 4:2:2 in YC Planar (YUV422UVP) where CbCr are packed. For displaying or capturing* `FVID_YCbCr422_SEMIPLANAR_UV` *enum should be used.* `FVID_YCbCr422_SEMIPLANER_UV` *is the only BT video format supported.*
- *For RAW capture VPIF get the data in Bayer Pattern from the sensor. For capturing RAW data* `FVID_RAW_FORMAT` *should be used*

17.5.1.4 Enum for frame storage format

```
typedef enum FVID_storageFormat_t
{
    FVID_STORAGE_FORMAT_FRAME,
    FVID_STORAGE_FORMAT_FIELD
} FVID_storageFormat;
```

This enumeration is used for specifying the storage format of the frame buffer video data. FIELD and FRAME storage is applicable only for interlaced formats. For progressive formats it is always FRAME mode of storage.

- *For details regarding the data storage please refer to SDRAM frame storage format section.*

17.5.1.5 Enum for VBI service type

```
typedef enum FVID_vbiService_t
{
    FVID_VBI_SERVICE_NONE = 0x0,
    FVID_VBI_SERVICE_HBI = 0x1,
```

```
    FVID_VBI_SERVICE_RAW_VBI = 0x2,
    FVID_VBI_SERVICE_SLICE_VBI = 0x4
}FVID_vbiService;
```

This enumeration defines the different types of VBI services possible. Depending up on the type of VBI service application can see the respective data for that service in the frame buffer.

17.5.1.6 Enum for video interface

```
typedef enum FVID_videoInterface_t
{
    FVID_VI_BT656_8BIT,
    /**< 8-bit BT.656 interface with embedded sync */
    FVID_VI_BT656_10BIT,
    /**< 10-bit BT.656 interface with embedded sync */
    FVID_VI_YC_8BIT_CS,
    /**< 8-bit YC interface with external control sync */
    FVID_VI_YC_10BIT_CS,
    /**< 10-bit YC interface with external control sync */
    FVID_VI_YC_16BIT_ES,
    /**< 16-bit YC interface with embedded sync */
    FVID_VI_YC_16BIT_CS,
    /**< 16-bit YC interface with external control sync */
    FVID_VI_RAW_8BIT_CS,
    /**< 8-bit RAW interface with external control sync */
    FVID_VI_RAW_10BIT_CS,
    /**< 10-bit RAW interface with external control sync */
    FVID_VI_RAW_16BIT_CS,
    /**< 16-bit RAW interface with external control sync */
    FVID_VIDEOINTERFACE_INVALID
}FVID_videoInterface;
```

This enumeration is not used and is for future use.

17.5.1.7 Enum for Field Frame Modes

```
typedef enum FVID_FieldFrame_t
{
    FVID_FIELD_MODE = 0,
```

```
    /**< Interlaced Mode */
    FVID_FRAME_MODE
    /**< Progressive Mode */
}FVID_FieldFrame;
```

This enumeration is not used and is for future use.

17.5.1.8 Enum for Bits per Pixel for different modules

```
typedef enum FVID_bitsPerPixel_t
{
    FVID_BPP_BITS1  = 1,
    FVID_BPP_BITS2  = 2,
    FVID_BPP_BITS4  = 4,
    FVID_BPP_BITS8  = 8,
    FVID_BPP_BITS10 = 10,
    FVID_BPP_BITS12 = 12,
    FVID_BPP_BITS16 = 16,
    FVID_BPP_BITS24 = 24
} FVID_bitsPerPixel;
```

The ENUM string itself is self explanatory of the bits per pixel. The video data is always FVID_BPP_BITS8 for BT capture and display. For raw capture the data width can be 8bpp, 10bpp or 12bpp depending on what is set during channel creation.

17.5.2 **Data Structures**

17.5.2.1 Structure for Interlaced Frame

```
typedef struct FVID_IFrame_t
{
    Char* y1;
    /**< Character pointer for field 1 Y data */
    Char* cb1;
    /**< Character pointer for field 1 CB data */
    Char* cr1;
    /**< Character pointer for field 1 CR data */
    Char* y2;
    /**< Character pointer for field 2 Y data */
    Char* cb2;
    /**< Character pointer for field 2 CB data */
```

```
    Char* cr2;

    /**< Character pointer for field 2 CR data */

} FVID_IFrame;
```

This structure is not used in the current C6748 VPIF driver as it doesn't support separate Cb and Cr components for chrominance. This is meant for future purpose.

17.5.2.2 Structure for Progressive Frame

```
typedef struct FVID_PFrame_t

{

    Char* y;

    /**< Character pointer for frame Y data */

    Char* cb;

    /**< Character pointer for frame CB data */

    Char* cr;

    /**< Character pointer for frame CR data */

} FVID_PFrame;
```

This structure is not used in the current C6748 VPIF driver as it doesn't support separate Cb and Cr components for chrominance. This is meant for future purpose.

17.5.2.3 Structure for Slice frame

```
typedef struct FVID_SliceFrame_t

{

    Uint32                      fvidSliceServiceId;

    /**< Type of Slice service. Available values for this field are defined
with FVID Slice VBI Service title in Fvid.h. */

    Uint8                       fvidField;

    /**< Field for which VBI data is required. 0: first field, 1: second
field*/

    Uint8                       fvidData[FVID_SLICE_VBI_DATA_SIZE_BYTES_MAX];

    /**< Place holder for getting the slice VBI data. */

}FVID_SliceFrame;
```

This structure defines the slice data frame structure. VPIF frame buffer structure contains pointer to this structure for slice data.

17.5.2.4 Structure for Semi Planar Frame

```
typedef struct FVID_SpFrame_t

{

    Uint8 *y1;

    /**< Pointer for top field Y data */
```

```
    Uint8 *c1;
    /**< Pointer for top field CB/CR data */
    Uint8 *y2;
    /**< Pointer for bottom field Y data. Not used for progressive format. */
    Uint8 *c2;
    /**< Pointer for bottom field CB/CR data. Not used for progressive
format.*/
}FVID_SpFrame;
```

This structure is used in the current C6748 VPIF driver. VPIF captures or displays video data in semi planar frame format. This structure will be used during VPIF frame transfer.

Here "1" in the variable name represents field 0 data and "2" represents field 1 data. For example fields named as *y1* and *y2*, where *y1* represents field 0 luminance data and *y2* represents field 1 luminance data. They are not named as y0 and y1 in order to keep it backward compatible with earlier FVID layers.

All the members are valid in case of interlaced mode but for progressive mode only *y1*, *c1* are used.

For progressive video data only use *y1* and *c1*.

For interlaced video data only – frame/field mode use *y1*, *y2*, *c1* and *c2*

The c data is CbCr packed.

- *To know how the data pointers mapped for FIELD and FRAME mode video storage please refer to SDRAM frame storage format section*

17.5.2.5 Structure for VBI Frame

```
typedef struct FVID_VbiFrame_t
{
    Uint8 *h1;
    /**< Pointer for top field RAW HANC data. Not used if RAW HANC data
    is not required */
    Uint8 *h2;
    /**< Pointer for bottom field RAW HANC data. Not used if RAW HANC data
    is not required */
    Uint8 *v1;
    /**< Pointer for top field RAW VANC data. Not used if RAW VANC data
    is not required */
    Uint8 *v2;
    /**< Pointer for bottom field RAW VANC data. Not used if RAW VANC data
    is not required */
```

```
    FVID_SliceFrame     *s1;
    /**< Slice VBI data structure for top field*/
    FVID_SliceFrame     *s2;
    /**< Slice VBI data structure for bottom field*/
}FVID_VbiFrame;
```

This structure is used in the current C6748 VPIF driver for capturing and displaying the VBI data.

Here "1" in the variable name represents field 0 data and "2" represents field 1 data. For example for interlaced *h1*, *h2*, *v1*, *v2*, *s1*, and *s2* are valid but for progressive only *h1*, *v1* and *s1* are valid. *h1* and *h2* are for RAW HBI data. *v1* and *v2* are for RAW VBI data. *s1* and *s2* are for slice VBI data.

All the members are valid in case of interlaced mode but for progressive mode only *h1*, *v1*, *s1* are used.

For raw VBI use *v1* (progressive) and both *v1* and *v2* (interlaced)

For raw HBI use *h1* (progressive) and both *h1* and *h2* (interlaced)

For slice VBI use *s1* (progressive) and *s1* and *s2* (interlaced)

17.5.2.6 Structure for Interlaced Raw Frame

```
typedef struct FVID_RawIFrame_t
{
    Char* buf1;
    /**< Character pointer for field 1 */
    Char* buf2;
    /**< Character pointer for field 2 */
} FVID_RawIFrame;
```

This structure is used to store the raw interlaced video data from vpif driver.

17.5.2.7 Structure for Progressive Raw Frame

```
typedef struct FVID_RawPFrame_t
{
    Char* buf;
    /**< Character pointer for frame */
} FVID_RawPFrame;
```

This structure is used to store the raw progressive data from vpif driver.

17.5.2.8 Structure for FVID frame buffer descriptor

```
typedef struct FVID_Frame_t
{
```

```
QUE_Elem            queElement;
/**< for queuing */

union {
    FVID_IFrame     iFrm;
    /**< y/c frame buffer for interlaced mode         */
    FVID_PFrame     pFrm;
    /**< y/c frame buffer for progressive mode        */
    FVID_RawIFrame  riFrm;
    /**< raw frame buffer for interlaced mode         */
    FVID_RawPFrame  rpFrm;
    /**< raw frame buffer for progressive mode        */
    Ptr             frameBufferPtr;
    FVID_SpFrame    spFrm;
    /**< y/c frame buffer for semi planar data         */
} frame;    /**< \brief union for frame type as used by driver */

Uint32              timeStamp;
/**< Time Stamp for captured or displayed frame */

Uint32              pitch;
/**< Pitch parameters for given plane */

Uint32              lines;
/**< Number of lines per frame */

FVID_bitsPerPixel   bpp;
/**< Number of bits per pixel */

FVID_colorFormat    frameFormat;
/**< Frame Color Format */

FVID_storageFormat  storeFormat;
/**< Storage Format */
```

```
    FVID_VbiFrame       vbiFrm;
    /**< VBI frame */

    FVID_vbiService     vbiService;
    /**< VBI Service */

    Ptr                 userParams;
    /**< In/Out Additional User Parameters per frame */

    Ptr                 misc;
    /**< For future use */
}FVID_Frame;
```

This structure is the descriptor which consolidates the buffer pointers and other useful parameters.

The structure members' *bpp* (bits per pixel), *frameFormat*, *storeFormat*, *vbiService*, *pitch* and *line*s are updated during the time of buffer allocation. The structure member *timestamp*, *queElement* and frame are used in C6748 VPIF driver and applications. They are used/updated for every frame exchange (queue/dequeue) operation. The structure member *misc*, *userParams* are not used by the C6748 driver currently and is meant for future purpose.

C6748 Vpif driver only supports planar 422 formats. Planar format is used for all of the frame types. YUV 422 planar format is used for Y/C frame buffer (vpifFrm). Frame types *riFr*m and *rpFrm* use raw format. "*vbiFrm*" used for VBI data storage.

17.5.3 Interface Functions

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

17.5.3.1 FVID_create

Syntax

FVID_Handle FVID_create(String name, Int mode, Int *status, Ptr optArgs, FVID_Attrs *attrs);

Parameters

name

The name argument is the name specified for the device when it was created in the configuration or at runtime. It is used to find a matching name in the device table.

Strings are case sensitive.

For VPIF driver the string is divided into 5 tokens separated by '/'.

- VPIF driver instance

This identifies the VPIF instance. For capture/display drivers this will be typically "*VPIF0*". This string depends on the device registration string given in BIOS driver TCI file.

- VPIF channel instance

  This identifies the channel to be opened in the VPIF instance. The VPIF instance has four channels – "0", "1", "2" and "3". Capture channel is supported on channel "0" and "1", whereas display is supported on channel "2" and "3". RAW capture is supported only on channel "0".

From here onwards the string is passed as is to the EDC driver and will be used by EDC driver internally. The tokens are typically more dependent on the EVM schematics and external encoders and decoders present in the EVM.

If there is no requirement for EDC driver configuration for a VPIF channel, the token afterwards can be absent.

- Codec string

  This identifies the codec which will be used to program the encoder and decoders. The encoders and decoders on C6748 EVM are connected to instance 0 of I2C and hence "I2C0" string is used. Based upon this string the underlying codec interface driver is opened.

- EDC driver name

  This is the name of the EDC driver to be opened for the channel. This will be used internally by the EDC driver to validate that the open call is for proper EDC driver. In the present C6748 EVM there are two instances of TVP5147. For channel 0 "*TVP5147_1*" string is used and for channel 1 "*TVP5147_0*" string is used. On C6748, for channel 2 "ADV7343" string is used for SD display.

Function pointer for the EDC driver, which is represented by "EDC driver name", should be passed properly during channel creation.

- EDC codec address

  This token tells the EDC driver about the external device address. This address is used by the codec interface to read/write the encoder/decoder registers.

  - *This token is typically more dependent on the EVM schematics and external encoders and decoders present in the EVM. Please refer to the schematics documents for the same.*

The following table shows the typical names for the current C6748 EVM

String Name	Description
"/VPIF0/0/I2C0/TVP5147_1/0x5D"	For VPIF instance 0 and channel no 0, EDC is connected through I2C 0 instance. The EDC device name is TVP5147 #1 which is connected for SD capture having I2C address as 0x5D.
"/VPIF0/1/I2C0/TVP5147_0/0x5C"	For VPIF instance 0 and channel no 1, EDC is connected through I2C 0 instance. The EDC device name is TVP5147 #0 which is connected for SD

	capture having I2C address as 0x5C.
"/VPIF0/2/I2C0/ADV7343/0x2A"	For VPIF instance 0 and channel no 2, EDC is connected through I2C 0 instance. The EDC device name is ADV7343 which is connected for SD display having I2C address as 0x2A.
"/VPIF0/0/I2C0/MT9T001/0x5D"	For VPIF instance 0 and channel no 0, EDC is connected through I2C 0 instance. The external image sensor name is MT9T001 which is connected for RAW capture having I2C address as 0x5D.

mode

The mode argument specifies the mode in which the device is to be opened. This may be *IOM_INPUT* or *IOM_OUTPUT*. *IOM_INPUT* mode is used for capture channel creation and *IOM_OUTPUT* mode is used for display channel creation.

status

The status argument is an output parameter that this function fills with a pointer to the status that was returned by the mini-driver.

optArgs

The optArgs parameter is a pointer that may be used to pass device or domain-specific arguments to the mini-driver. The contents at the specified address are interpreted by the mini-driver in a device-specific manner. The memory segment id for memory allocation is also passed via this parameter.

For Vpif driver, optArgs will be pointer of type *Vpif_CapChanParams* for capture driver SD/raw capture channel creation or *Vpif_DisChanParams* for display channel creation.

VPIF driver doesn't assume any default value for this argument. This is because segment ID (used for frame buffer allocation) is passed to the driver only through this parameter. Hence VPIF driver will return error value if application passes NULL for this parameter.

attrs

The attrs parameter is a pointer to a structure of type FVID_Attrs. This is not supported and NULL should be passed.

Return Value

It returns the handle of type *FVID_Handle* on successful opening of a device. It returns NULL if the device could not be opened.

Description

An application calls FVID_create() to create and initialize a VPIF driver channel. The driver will not allocate frame buffers for FVID_exchange() and other APIs during this call. Applications have to create buffers for this purpose. It is suggested that applications should use the APIs FVID_allocBuffer() and FVID_freeBuffer() provided with driver for frame buffer allocation purpose.

A minimum of 3 frame buffers is required per channel creation for proper operation.

FVID_create() returns a handle to the channel if it is successfully opened. This handle should be used by subsequent FVID module calls on this channel.

Constraints

This function can only be called after the device has been loaded and initialized.

The "*mode*" parameter should be *IOM_INPUT* for channel 0 and 1 and *IOM_OUTPUT* for channel 2 and 3.

Example

The example below shows creation of capture channel 0 for VPIF

```
/* Structure to store each driver channel information   */

typedef struct ChannelInfo_t

{

    FVID_Handle chanHandle;     /* Channel handle  */

    FVID_Frame *frame;          /* Current FVID frame buffer pointer */

}ChannelInfo;
```

```
/* Structure containing display and capture channel information */

ChannelInfo         capChInfo;

Vpif_CapChanParams  vCapParamsChan;

/* Setup Capture Channel 0 -> Composite. Use this capture driver name string
as they are for proper driver creation */

Int8                *vpifCapStrings = "/VPIF0/0/I2C0/TVP5147_1/0x5D";

/* Create and configure capture drivers */

vCapParamsChan.capEdcTbl = &TVP5147_Fxns;

vCapParamsChan.capChannelIoMode = Vpif_IoMode_CAP;

vCapParamsChan.capFbParams.frmBufAlignment = 128u;

vCapParamsChan.capFbParams.frmBufSegId = 0;/* Create from system heap*/

vCapParamsChan.capStdMode = Vpif_VideoMode_NTSC;

vCapParamsChan.capStorageMode = Vpif_SdramStorage_FIELD;

vCapParamsChan.capVideoParams = NULL;

vCapParamsChan.capVbiService = Vpif_VbiServiceType_NONE;

capChInfo.chanHandle = FVID_create(vpifCapStrings,

                           IOM_INPUT,

                           &status,

                           (Ptr)&vCapParamsChan,

                           NULL);

if ((IOM_COMPLETED != status) || (NULL == capChInfo.chanHandle))

{
```

```
    LOG_printf(&trace, "Failed to create capture channel");
}
```

17.5.3.2 FVID_delete

Syntax

Int FVID_delete(FVID_Handle fvidChan);

Parameters

fvidChan

Handle of the vpif driver channel that was created with a call to FVID_create().

Return Value

The function returns *IOM_COMPLETED* on success or negative value if an error occurred. This function is a wrapper above GIO_delete() function. Since GIO_delete() always returns success irrespective of VPIF driver return value, this function always returns *IOM_COMPLETED*.

Description

This function call will close the logical channel associated with fvidChan parameter. It will not free the buffers allocated by driver. It is the applications responsibility to free the already allocated buffers before channel deletion. Please note that, if capture/display operation is started, then *Vpif_IOCTL_CMD_STOP* should be called before calling FVID_delete().

EDC driver associated with the channel is also closed in this function call.

Constraints

This function can only be called after the device has been loaded, initialized and created.

Example

The example below shows deletion of the capture channel already created

```
/* Delete capture driver */
status = FVID_delete(capChInfo.chanHandle);
if (IOM_COMPLETED != status)
{
    LOG_printf(&trace, "Failed to delete capture channel");
}
```

17.5.3.3 FVID_control

Syntax

Int FVID_control(FVID_Handle fvidChan, Int cmd, Ptr args);

Parameters

fvidChan

Handle of the vpif driver channel that was created with a call to FVID_create().

cmd

The cmd argument specifies the control command.

args

The args argument is a pointer to the argument or structure of arguments that are specific to the command being passed.

Return Value

This function returns *IOM_COMPLETED* on success or negative value if an error occurred.

Description

An application calls FVID_control() to send device-specific control commands to the mini-driver.

Below are the supported control commands by C6748 Vpif driver. The following sections explain the commands in detail.

- Vpif_IOCTL_CMD_CHANGE_RESOLUTION

  Reconfigures the resolution of capture or display channel. This command can be used to change the resolution of the operating channel.

- Vpif_IOCTL_CMD_START

  Start display/capture operation.

- Vpif_IOCTL_CMD_STOP

  Stop display/capture operation.

- Vpif_IOCTL_CMD_GET_NUM_IORQST_PENDING

  Gets the number of pending request at driver level

- Vpif_IOCTL_CMD_GET_CHANNEL_STD_INFO

  Get the current channel configuration parameters from driver.

- Default IOCTL

  Configure the external encoders and decoders. Interface will depend on the encoder/decoder drivers.

Constraints

This function can only be called after the device has been loaded, initialized and created. The handle supplied as an argument to this function should have been obtained with a previous call to FVID_create().

This function is not re-entrant for a channel.

Example

The example below shows the start of the capture channel for VPIF

```
/* Start the capture operations */

status = FVID_control(capChInfo.chanHandle, Vpif_IOCTL_CMD_START, NULL);

if (IOM_COMPLETED != status)

{

    LOG_printf(&trace, "Failed to start capture channel device");

}
```

17.5.3.3.1 Vpif_IOCTL_CMD_CHANGE_RESOLUTION

Syntax

Int FVID_control(fvidChan, Vpif_IOCTL_CMD_CHANGE_RESOLUTION, args);

Parameters

fvidChan

Handle of the vpif driver channel that was created with a call to FVID_create().

cmd

Vpif_IOCTL_CMD_CHANGE_RESOLUTION control command.

args

The argument is a pointer to structure containing the new configuration and is of type *Vpif_ConfigParams*. Application can choose to specify the pre-defined modes (enum *Vpif_VideoMode*) in the "mode" parameter or Application can set the "mode" parameter to "*Vpif_VideoMode_NONE*" and provide the filled up *Vpif_ConfigParams* structure.

Return Value

This function returns *IOM_COMPLETED* on success or negative value if an error occurred.

Description

This function call is used to change the resolution for a channel.

Application calls this function when channel is stopped and the driver will reconfigure the resolution parameters but will not start channel. Application has to queue buffers before starting channel again.

It is application's responsibility to free memory for all the buffers before reconfiguring channel.

Constraints

This function can only be called after the device has been stopped. The handle supplied as an argument to this function should have been obtained with a previous call to FVID_create(). Also the buffer the buffers should be freed up, as the buffer requirement changes once the resolution changes.

- *Please note that changing the resolution between SD, HD and RAW mode is not allowed i.e. channel properties cannot be changed (Application may need to close the channel and create channel in that case).*
- *Using this IOCTL the application can switch between different resolutions with in SD (PAL to NTSC) or HD (720P to 1080P) or RAW (VGA to SVGA).*
- *If application sets valid mode in "mode" parameter and also sends the filled structure, the driver would consider the "mode" parameter and update accordingly.*
- *The driver does not check the validity for these parameters when application passes the structure with updated parameters for changed resolution.*

Example

The example below shows changing resolution of a raw capture channel for VPIF

```
Vpif_ConfigParams chResolution;
```

```
chResolution.mode = Vpif_VideoMode_RAW_UXGA;

status = FVID_control(rawChInfo.chanHandle,

             Vpif_IOCTL_CMD_CHANGE_RESOLUTION,

             &chResolution);

if (IOM_COMPLETED != status)

{

    LOG_printf(&trace, "Failed to change the resolution");

}
```

17.5.3.3.2 Vpif_IOCTL_CMD_START

Syntax

Int FVID_control(fvidChan, Vpif_IOCTL_CMD_START, args);

Parameters

fvidChan

Handle of the vpif driver channel that was created with a call to FVID_create().

cmd

Vpif_IOCTL_CMD_START control command.

args

None

Return Value

This function returns *IOM_COMPLETED* on success or negative value if an error occurred.

Description

This function call is used to start capture or display operation.

Constraints

This function can only be called after the device has been loaded, initialized and created. The handle supplied as an argument to this function should have been obtained with a previous call to FVID_create().

This function can be called only after minimum required buffers are queued up.

Example

The example below shows starting a display channel for VPIF

```
/* Start display operation */

status = FVID_control(disChInfo.chanHandle, Vpif_IOCTL_CMD_START, NULL);

if (IOM_COMPLETED != status)

{

    LOG_printf(&trace, "Failed to start display channel device");

}
```

17.5.3.3.3 Vpif_IOCTL_CMD_STOP

Syntax

Int FVID_control(fvidChan, Vpif_IOCTL_CMD_STOP, args);

Parameters

fvidChan

Handle of the vpif driver channel that was created with a call to FVID_create().

cmd

Vpif_IOCTL_CMD_STOP control command.

args

None

Return Value

This function returns *IOM_COMPLETED* on success or negative value if an error occurred.

Description

This function call is used to stop capture or display operation.

Constraints

This function can only be called after the device has been loaded, initialized, created and started. The handle supplied as an argument to this function should have been obtained with a previous call to FVID_create().

This function can be called only after capture or display operation has started.

Example

The example below shows stopping a capture channel for VPIF

```
/* Stop capture operation */

status = FVID_control(capChInfo.chanHandle, Vpif_IOCTL_CMD_STOP, NULL);

if (IOM_COMPLETED != status)

{

    LOG_printf(&trace, "Error in stopping capture operation");

}
```

17.5.3.3.4 Vpif_IOCTL_CMD_GET_NUM_IORQST_PENDING

Syntax

Int FVID_control(fvidChan, Vpif_IOCTL_CMD_GET_NUM_IORQST_PENDING, args);

Parameters

fvidChan

Handle of the vpif driver channel that was created with a call to FVID_create().

cmd

Vpif_IOCTL_CMD_GET_NUM_IORQST_PENDING control command.

args

Pointer to integer

Return Value

This function returns *IOM_COMPLETED* on success or negative value if an error occurred.

Description

This function call will get number of pending requests at driver level. It will provide number of requests yet to be served by driver.

Constraints

This function can only be called after the device has been loaded, initialized and created. The handle supplied as an argument to this function should have been obtained with a previous call to FVID_create().

This function can be called only after minimum required buffers are queued up.

Example

The example below shows getting pending request with the channel for VPIF

```
FVID_Handle chanHandle;

Int numPendingReq;

/* channel creation and queueing should be done here */

/* call to get number of pending requests */

status = FVID_control(capChInfo.chanHandle,

             Vpif_IOCTL_CMD_GET_NUM_IORQST_PENDING,

             &numPendingReq);

if (IOM_COMPLETED != status)

{

    LOG_printf(&trace, "Failed in getting pending requests");

}
```

17.5.3.3.5 Vpif_IOCTL_CMD_GET_CHANNEL_STD_INFO

Syntax

Int FVID_control(fvidChan, Vpif_IOCTL_CMD_GET_CHANNEL_STD_INFO, args);

Parameters

fvidChan

Handle of the vpif driver channel that was created with a call to FVID_create().

cmd

Vpif_IOCTL_CMD_GET_CHANNEL_STD_INFO control command.

args

Pointer to structure of type *Vpif_StdInfo*

Return Value

This function returns IOM_COMPLETED on success or negative value if an error occurred.

Description

This function will provide current channel standard parameters.

Constraints

This function can only be called after the device has been loaded, initialized and created. The handle supplied as an argument to this function should have been obtained with a previous call to FVID_create().

Example

The example below shows how to get the channel parameters for a raw capture channel for VPIF

```
Vpif_StdInfo        rawParams;
status = FVID_control(rawChInfo.chanHandle,
             Vpif_IOCTL_CMD_GET_CHANNEL_STD_INFO,
             &rawParams);
if (IOM_COMPLETED != status)
{
    LOG_printf(&trace, "Failed to get raw capture channel info");
}
```

17.5.3.3.6 Default ioctl

Any other ioctls passed, apart from the above, results in a call to the EDC driver for that channel. This call is only made if the channel parameter "*dispEdcTbl*" or "*capEdcTbl*" is not passed as NULL during channel creation.

To call any EDC specific ioctl application needs to add *Vpif_IOCTL_CMD_MAX* to the EDC ioctl.

Example

The example below shows how to set configuration for a display channel for VPIF

```
Adv7343_ConfParams vDisParamsEncoder =
{
    Adv7343_AnalogFormat_COMPOSITE,   /* AnalogFormat      */
    Adv7343_Std_AUTO,                 /* Video std         */
    Adv7343_InputFormat_YCBCR422,     /* InputFormat       */
    Fvid_SLICE_VBI_SERVICES_NONE     /* slice vbi service */
};
/* Configure ADV7343 */
status = FVID_control(disChInfo.chanHandle,
             Vpif_IOCTL_CMD_MAX + Edc_IOCTL_CONFIG,
             (Ptr)&vDisParamsEncoder);
if (IOM_COMPLETED != status)
{
```

```
    LOG_printf(&trace, "Failed to get raw capture channel info");
}
```

17.5.3.4 FVID_exchange

Syntax

Int FVID_exchange(FVID_Handle fvidChan, Ptr bufp);

Parameters

name

Handle of the vpif driver channel that was created with a call to FVID_create().

bufp

The bufp argument is an in/out parameter that points to the application-owned buffer that is to be relinquished back to the driver. After the call returns successfully, this function fills bufp with a pointer to the structure of type *FVID_Frame* that was exchanged by the device driver.

Return Value

FVID_exchange() returns *IOM_COMPLETED* when it is completed successfully. If an error occurs, a negative value will be returned.

Description

An application calls FVID_exchange() to relinquish a video buffer back to the vpif device driver and take a buffer back from the driver. This function fills bufp with a pointer to the structure of type *FVID_Frame* that is exchanged by the device driver and returned to application. This API function will result in an vpifMdSubmitChan() call being made to the mini-driver.

For capture operation the buffer submitted to the driver is an empty buffer and the buffer returned from the driver is most recent captured frame and for display operation the buffer to be displayed is submitted to the driver and the buffer returned is empty or already displayed.

This operation is similar to calling FVID_queue() and FVID_dequeue() one after the other. Refer corresponding API description for details.

Constraints

This function can only be called after the device has been loaded, initialized and created. Cache coherency of the frame buffer should be taken care by the application.

Example

The example below shows buffer exchange for a capture channel for VPIF

```
/* Invalidate the buffer before giving to capture driver */

BCACHE_inv(capChInfo.frame->frame.vpifFrm.y1, (sizeimage * 2), TRUE);

/* Give the old capture frame buffer back to driver and get the recently
captured frame buffer */

status = FVID_exchange(capChInfo.chanHandle, &(capChInfo.frame));

if (IOM_COMPLETED != status)
```

```
{
    LOG_printf(&trace, "Error in exchanging capture buffer");
}
```

17.5.3.5 FVID_dequeue

Syntax

Int FVID_dequeue(FVID_Handle fvidChan, Ptr bufp);

Parameters

fvidChan

Handle of the vpif driver channel that was created with a call to FVID_create().

bufp

The bufp argument is an out parameter that this function fills with a pointer to the structure of type *FVID_Frame* that was allocated by the device driver.

Return Value

FVID_dequeue() returns *IOM_COMPLETED* when it completes successfully. If an error occurs, a negative value will be returned. If there is no buffer available with driver to return to application, this function will be blocked. But if application calls FVID_dequeue() after calling *Vpif_IOCTL_CMD_STOP* and if there is no buffer available with driver to return to application, then *IOM_ENOPACKETS* code will be returned.

Description

An application will call FVID_dequeue() to request the vpif device driver to give ownership of a data buffer. This API function will result in an vpifMdSubmitChan() call being made to the mini-driver.

For display operation, the driver will return an empty frame buffer which the application can use to fill the next frame data to be displayed. For capture operation, the driver will return the most recently captured frame buffer which can be used by the application for further processing.

After the channel is stopped, this function is used to get all the buffers owned by the driver to free it by calling FVID_freeBuffer() API.

Constraints

This function can only be called after the device has been loaded, initialized and created. Cache coherency of the frame buffer should be taken care by the application.

This function should be called only after queuing minimum number of buffers to the drivers.

Example

The example below shows buffer dequeue for a capture channel for VPIF

```
/* Request a frame buffer from capture driver. Capture buffer will return the
latest captured buffer */

status = FVID_dequeue(capChInfo.chanHandle, &(capChInfo.frame));

if (IOM_COMPLETED != status)
```

```
{
    LOG_printf(&trace, "Failed to dequeue capture channel device");
}
```

17.5.3.6 FVID_queue

Syntax

Int FVID_queue(FVID_Handle fvidChan, Ptr bufp);

Parameters

fvidChan

Handle of the vpif driver channel that was created with a call to FVID_create().

bufp

The bufp argument is a pointer to the structure of type *FVID_Frame* that was previously allocated by the device driver and is not to be relinquished.

Return Value

FVID_queue() returns *IOM_COMPLETED* when it completes successfully. If an error occurs, a negative value will be returned.

Description

An application calls FVID_queue() to submit a video buffer to the vpif device driver. This API function will result in an vpifMdSubmitChan() call being made to the mini-driver.

For display operation, the application gives a filled frame buffer that needs to be displayed next. For capture operation, the application gives an empty buffer to the driver for capturing the next frame data.

Before the channel is started, this function is used to queue the required number of buffers allocated by calling FVID_allocBuffer() API.

Constraints

This function can only be called after the device has been loaded, initialized and created. Cache coherency of the frame buffer should be taken care by the application.

The pointer that is passed as an argument to this call must point to a video buffer of type *FVID_Frame*. This pointer must point to either the buffer newly allocated or the buffer already provided by the driver through a call to FVID_dequeue() or FVID_exchange() or FVID_allocBuffer() calls.

Example

The example below shows buffer queue for a capture channel for VPIF

```
/* Queue the frame buffers for capture */
status = FVID_queue(capChInfo.chanHandle, &(capChInfo.frame));
if (IOM_COMPLETED != status)
{
    LOG_printf(&trace, "Failed to Queue capture buffer");
}
```

17.5.3.7 FVID_allocBuffer

Syntax

Int FVID_allocBuffer(FVID_Handle fvidChan, Ptr bufp);

Parameters

fvidChan

Handle of the vpif driver channel that was created with a call to FVID_create().

bufp

The bufp argument is an out parameter which will contain pointer to the allocated frame buffer from the segment ID provided as a part of channel parameter in FVID_create().

Return Value

FVID_allocBuffer() returns *IOM_COMPLETED* when it completes successfully. *IOM_EALLOC* is returned in case of insufficient memory for buffer allocation else a negative value will be returned in case of other errors.

Description

An application will call FVID_allocBuffer() to request the vpif device driver to allocate one data buffer. This function allocates memory for one frame buffer and one structure variable of type *FVID_Frame*. This function fills buffer pointer in *FVID_Frame* structure variable and assigns its pointer to the structure pointer of type *FVID_Frame* passed as an argument. This API function will result in an vpifMdControlChan() call being made to the mini-driver. The segment ID passed to the driver during FVID_create() will be used for allocation.

It is the responsibility of the application to dequeue the buffer from driver and free it before the channel is deleted.

Constraints

This function can only be called after the device has been loaded, initialized and created.

Example

The example below shows how to allocate and queue the frame buffers in capture channel for VPIF

```
/* Allocate and Queue buffers for capture channel */

/* Allocate Frame buffer for capture driver */

status = FVID_allocBuffer(capChInfo.chanHandle, &(capChInfo.frame));

if (IOM_COMPLETED != status)

{

    LOG_printf(&trace, "Failed to allocate buffer for capture");

}

else

{

    /* After mapping each buffer, it is a good idea to first "zero" them out.
Here it is being set to a mid grey-scale Y=0x80, Cb=0x80, Cr=0x80*/
```

```
    memset((Uint8 *)capChInfo.frame->frame.vpifFrm.y1, 0x80, sizeimage);
    memset((Uint8 *)capChInfo.frame->frame.vpifFrm.c1, 0x80, sizeimage);
    /* Queue the frame buffer for capture */
    status = FVID_queue(capChInfo.chanHandle, &(capChInfo.frame));
    if (IOM_COMPLETED != status)
    {
        LOG_printf(&trace, "Failed to Queue capture buffer");
    }
}
```

17.5.3.8 FVID_freeBuffer

Syntax

Int FVID_freeBuffer(FVID_Handle fvidChan, Ptr bufp);

Parameters

fvidChan

Handle of the vpif driver channel that was created with a call to FVID_create().

bufp

The bufp argument will contain pointer to the frame buffer that is to be released.

Return Value

FVID_freeBuffer() returns *IOM_COMPLETED* when it completes successfully. If an error occurs, a negative value will be returned.

Description

An application will call FVID_freeBuffer() to request the vpif device driver to free memory of one data buffer. Pointer to this data buffer will be passed as an argument to FVID_freeBuffer(). This API call will free memory of one data buffer and one *FVID_Frame* structure variable. This API function will result in an vpifMdControlChan() call being made to the mini-driver.

Constraints

This function can only be called after the device has been loaded, initialized and created. The pointer that is passed as an argument to this call must point to a video buffer of type *FVID_Frame*. This pointer must point to buffer already allocated by the driver through a call to FVID_allocBuffer().

Example

The example below shows how to dequeue and free a frame buffer in capture channel for VPIF

```
/* Dequeue buffers from driver and free them */
status = FVID_dequeue(capChInfo.chanHandle, &(capChInfo.frame));
if (IOM_COMPLETED != status)
{
```

```
    LOG_printf(&trace, "IOM_COMPLETED != status for DQ");
}
status = FVID_freeBuffer(capChInfo.chanHandle, &(capChInfo.frame));
if (IOM_COMPLETED != status)
{
    LOG_printf(&trace, "IOM_COMPLETED != status for free buff");
}
```

17.5.4 Using FVID API's

The following is a simplified example of an application that is capturing data from a video source (e.g. DVD) and displaying the data to a display device (e.g. TV).

```
#include <std.h>
#include "ti/pspiom/vpif/Fvid.h"
#include "ti/pspiom/vpif/Vpif.h"

#define NUM_FRAME_BUFFERS           (3u)
#define MAXLOOPCOUNT                (500u)

/* Structure to store each driver channel information */
typedef struct ChannelInfo_t
{
    FVID_Handle chanHandle;     /* Channel handle */
    FVID_Frame *frame;          /* Current FVID frame buffer pointer */
}ChannelInfo;

Void main()
{
    /* DSP/BIOS scheduler starts at the termination of main() */
}
/* Video processing task */
Void vpifSampleApp(Void)
{
    Vpif_CapChanParams  vCapParamsChan;
    Vpif_DisChanParams  vDisParamsChan;
    /* Structure containing display and capture channel information */
```

```
ChannelInfo         capChInfo;
ChannelInfo         disChInfo;
Int8                *vpifCapStrings = "/VPIF0/0/I2C0/TVP5147_1/0x5D";
Int8                *vpifDisStrings = "/VPIF0/2/I2C0/ADV7343/0x2A";


/* Create and configure capture drivers */
vCapParamsChan.capEdcTbl = &TVP5147_Fxns;
vCapParamsChan.capChannelIoMode = Vpif_IoMode_CAP;
vCapParamsChan.capFbParams.frmBufAlignment = 128u;
vCapParamsChan.capFbParams.frmBufSegId = 0;/* Create from system heap*/
vCapParamsChan.capStdMode = Vpif_VideoMode_NTSC;
vCapParamsChan.capStorageMode = Vpif_SdramStorage_FIELD;
vCapParamsChan.capVideoParams = NULL;
vCapParamsChan.capVbiService = Vpif_VbiServiceType_NONE;
capChInfo.chanHandle = FVID_create(vpifCapStrings,
                           IOM_INPUT,
                           &status,
                           (Ptr)&vCapParamsChan,
                           NULL);
/* Create and configure display driver */
vDisParamsChan.dispEdcTbl = &ADV7343_Fxns;
vDisParamsChan.dispChannelIoMode = Vpif_IoMode_DIS;
vDisParamsChan.dispFbParams.frmBufAlignment = 128u;
vDisParamsChan.dispFbParams.frmBufSegId = 0;/* Create from system heap*/
vDisParamsChan.dispStdMode = Vpif_VideoMode_NTSC;
vDisParamsChan.dispStorageMode = Vpif_SdramStorage_FIELD;
vDisParamsChan.dispVideoParams = NULL;
vDisParamsChan.dispVbiService = Vpif_VbiServiceType_NONE;
vDisParamsChan.dispHVbiParams = NULL;
vDisParamsChan.dispVVbiParams = NULL;
disChInfo.chanHandle = FVID_create(vpifDisStrings,
                           IOM_OUTPUT,
                           &status,
                           (Ptr)&vDisParamsChan,
```

```
                                NULL);
    for (bufCount = 0; bufCount < NUM_FRAME_BUFFERS; bufCount++)
    {
        /* Allocate Frame buffers */
        FVID_allocBuffer(capChInfo.chanHandle, &(capChInfo.frame));
        FVID_allocBuffer(disChInfo.chanHandle, &(disChInfo.frame));
        /* Queue the frame buffers to driver */
        FVID_queue(capChInfo.chanHandle, &(capChInfo.frame));
        FVID_queue(disChInfo.chanHandle, &(disChInfo.frame));
    }
    /* Start display and capture operations */
    FVID_control(disChInfo.chanHandle, Vpif_IOCTL_CMD_START, NULL);
    FVID_control(capChInfo.chanHandle, Vpif_IOCTL_CMD_START, NULL);

    /* Let application have ownership of first frame buffers */
    FVID_dequeue(capChInfo.chanHandle, &(capChInfo.frame));
    FVID_dequeue(disChInfo.chanHandle, &(disChInfo.frame));
    while (counter < MAXLOOPCOUNT)
    {
        /* Invalidate the buffer before giving to capture driver */
        BCACHE_inv(capChInfo.frame->frame.vpifFrm.y1, (sizeimage * 2), TRUE);
        /* Give the old capture frame buffer back to driver and get the
recently captured frame buffer */
        FVID_exchange(capChInfo.chanHandle, &(capChInfo.frame));
        /* Flush and invalidate the processed buffer so that the DMA reads
the processed data */
        BCACHE_wbInv(capChInfo.frame->frame.vpifFrm.y1, (sizeimage * 2),
TRUE);
        /* Give the captured frame buffer to display driver and get a
free frame buffer for next capture */
        FVID_exchange(disChInfo.chanHandle, &(capChInfo.frame));
        counter++;
    }
    /* Stop capture and display operation */
    FVID_control(disChInfo.chanHandle, Vpif_IOCTL_CMD_STOP, NULL);
```

```
    FVID_control(capChInfo.chanHandle, Vpif_IOCTL_CMD_STOP, NULL);

    /* Free the buffer owned by application */

    FVID_freeBuffer(disChInfo.chanHandle, &(disChInfo.frame));

    FVID_freeBuffer(capChInfo.chanHandle, &(capChInfo.frame));

    /* Dequeue buffers from driver and free them */

    for (bufCount = 0; bufCount < (NUM_FRAME_BUFFERS - 1u); bufCount++)

    {

        FVID_dequeue(disChInfo.chanHandle, &(disChInfo.frame));

        FVID_dequeue(capChInfo.chanHandle, &(capChInfo.frame));

        FVID_freeBuffer(disChInfo.chanHandle, &(disChInfo.frame));

        FVID_freeBuffer(capChInfo.chanHandle, &(capChInfo.frame));

    }

    /* Delete capture and display channel */

    FVID_delete(disChInfo.chanHandle);

    FVID_delete(capChInfo.chanHandle);

}
```

17.6 EDC Configurations

This section describes in detail about External Device Control (EDC) mechanism of VPIF driver - EVM or hardware dependent components that are not built inside VPIF module and VPIF has dependency on such peripherals. C6748 vpif driver configures external video decoders and encoders using I2C interface to capture or display video.

This section describes the functions, data structures and enumerations for the EDC module.

Most of the functionality and features supported by the EDC driver depends on the C6748 EVM schematics and VPIF support. Features which are not supported by the current C6748 EVM and VPIF are mentioned as NOT SUPPORTED in the appropriate places. The options which are not supported are given only for future purpose.

- *User should take care of below mentioned points while porting C6748 VPIF driver on different EVM:*
    - *If any encoders and decoders are different than ADV7343, TVP5147 and MT9T001, EDC driver for respective encoder or decoder should be developed. The interface of EDC driver should be same as described in EDC section.*
    - *If encoders and decoders are same as C6748 EVM, but if their hardware interface with VPIF is different than C6748 EVM then corresponding modifications should be done in EDC driver. For example, in some EVM, encoder A is connected with VPIF via encoder B in bypass mode then corresponding modifications should be done in EDC driver.*
    - *If the Codec interface to the encoder or decoder changes other than I2C, then the codec interface for the same should be implemented.*

17.6.1 Interface between VPIF and EDC Driver

Below figure shows interface between VPIF driver and EDC driver when any function is being called from application. Here, EDC Open, EDC Control or EDC Close functions represent corresponding encoder/decoder functions.

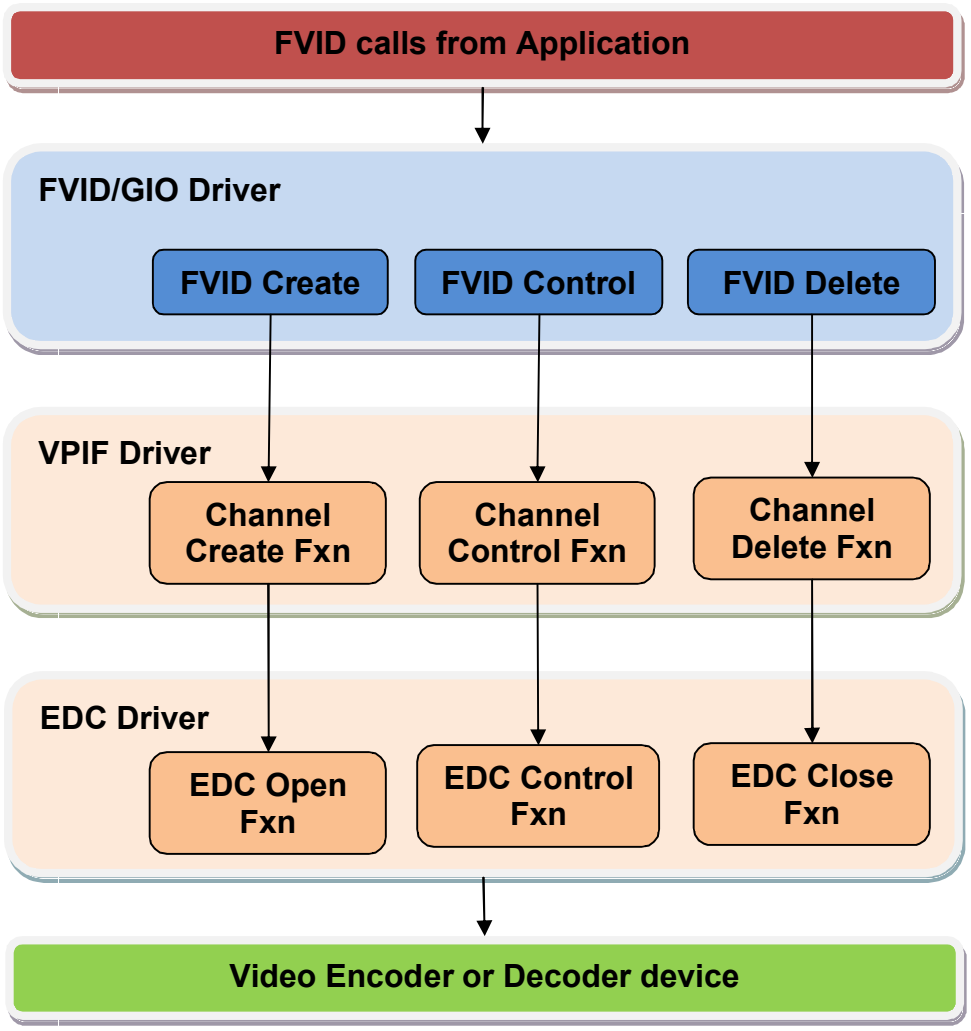


Figure 4. Interaction between VPIF and EDC driver

The EDC driver is associated with each channel of the VPIF driver through the "*capEdcTbl*" or "*dispEdcTbl*" member (of type *EDC_Fxns*) of *Vpif_CapChanParams* or *Vpif_DisChanParams*. This is passed during VPIF driver channel creation call to vpifMdCreateChan(). Each VPIF channel can be associated with one EDC driver.

If edcTbl is NULL in channel parameters, then it is assumed that the channel has no external encoder or decoder attached.

C6748 EVM has following external encoders and decoders. The details of each driver interface are explained in the following section.

- Two TVP5147 Decoders
- One ADV7343 Encoder
- External MT9T001 Sensor

17.6.2 Constants & Enumerations

17.6.2.1 Edc_IOCTL

```
typedef enum Edc_IOCTL_t
{
    Edc_IOCTL_CONFIG = 0,
    /**< EDC configure command */
    Edc_IOCTL_RESET,
    /**< EDC reset command */
    Edc_IOCTL_SET_REG,
    /**< Command to write/set the EDC registers */
    Edc_IOCTL_GET_REG,
    /**< Command to read/get the EDC registers */
    Edc_IOCTL_CMD_MAX
}Edc_IOCTL;
```

This enum defines the different IOCTL commands used to perform control operation on EDC device. They are common for both encoder and decoders operation. The IOCTL command is passed as second argument to ctrl() function pointer of the EDC device function when the driver is used directly with the application.

- *These IOCTL's will be passed to EDC, only if application adds `Vpif_IOCTL_CMD_MAX` to these IOCTL calls from application.*
- *If there is any restriction in implementing them by the encoder/decoder device they should be appropriately noted in the respective encoder and decoder.*

Following table give the type of parameters used by these IOCTLs

Command	Argument	Description
Edc_IOCTL_CONFIG	*Tvp5147_ConfParams * (for example)*	Application has to pass appropriate configuration structure pointer described in the encoder or decoder header file.
Edc_IOCTL_RESET	*None*	This will reset the EDC device.
Edc_IOCTL_SET_REG	*Edc_RegData ***	Command to write/set the EDC registers.
Edc_IOCTL_GET_REG	*Edc_RegData **	Command to read/get the EDC registers.

17.6.2.2 Edc_VideoType

```
typedef enum Edc_VideoType_t
{
    Edc_VideoType_SD = 0,
```

```
    /**< Indicates SD parameters */
    Edc_VideoType_ED,
    /**< Indicates ED parameters - Not supported */
    Edc_VideoType_HD
    /**< Indicates HD parameters - Not supported */
}Edc_VideoType;
```

This enum defines the different video types available by the encoder/decoder device.

- *Enumeration related to ED and HD are not supported by the current driver on C6748*

17.6.2.3 Edc_ControlBusType

```
typedef enum Edc_ControlBusType_t
{
    Edc_ControlBusType_I2C,
    /**< Control Bus for Encoder/Decoder is I2C */
    Edc_ControlBusType_SPI,
    /**< Control Bus for Encoder/Decoder is SPI - Not implemented */
    Edc_ControlBusType_UNKNOWN
    /**< Delimiter Enum */
}Edc_ControlBusType;
```

This enum defines the underlying control bus controlling the read/write to encoder or decoder.

- *Control bus as SPI is not supported by the current driver on C6748.*

17.6.3 Data Structures

17.6.3.1 Edc_RegData

"*Edc.h*" file contains *Edc_RegData* data structure that is passed in *Edc_IOCTL_GET_REG* and *Edc_IOCTL_SET_REG* ioctl for getting and setting the registers of the EDC device. This structure used during read or write to the encoder/decoder registers and specifies the register write or read information. The members of this structure are explained below:

Structure Members	Description
startReg	The starting index of encoder or decoder register
noRegToRW	The total number of registers to read/write. CAUTION: "*noRegToRW*" should be number of CONSECUTIVE registers to be read or written.
value	The register data to be read/written

> *"noRegToRW" should be number of **CONSECUTIVE** registers to be read or written.*

17.6.3.2 EDC_Fxns

"*Edc.h*" file contains EDC function table structure that is passed to the VPIF device during channel creation. Using *Edc_Fxns* structure VPIF calls the open, close and control functions of the respective encoder and decoder.

Below structure definition provides details about the function pointers where-in the external encoder/decoder plugs-in.

```
typedef struct EDC_Fxns_t

{

    EDC_Handle (*open)(String name, Ptr optArg);

    /**< edcOpen() - required, open the device */

    Int32 (*close)(Ptr devHandle);

    /**< edcClose() - required, close the device */

    Int32 (*ctrl)(Ptr devHandle, Uns cmd, Ptr arg);

    /**< edcCtrl() - required, control/query device */

}EDC_Fxns;
```

Every EDC based encoder /decoder/sensor should export its function table pointer through xxx_Fxns global variable.

17.6.4 TVP5147 Decoder

The TVP5147M1 decoder supports the analog-to-digital (A/D) conversion of component YPbPr signals, as well as the A/D conversion and decoding of NTSC, PAL, and SECAM composite and S-video into component YCbCr. This decoder includes two 10-bit 30-MSPS A/D converters (ADCs). A total of ten video input terminals can be configured to a combination of YPbPr, CVBS, or S-video video inputs.

On CVBS and S-video inputs, the user can control video characteristics such as contrast, brightness, saturation, and hue via an I2C host port interface.

The digital data output can be programmed to two formats: 20-bit 4:2:2 with external syncs or 10-bit 4:2:2 with embedded/separate syncs. The TVP5147M1 decoder includes methods for advanced vertical blanking interval (VBI) data retrieval.

The current C6748 EVM contains 2 TVP5147 decoders capable of capturing 2 (1 x 2) SD video channels simultaneously.

TVP5147 input and output interface details are given below:

Analog Input Interface:

- Composite video
- S-video
- Component video (Not supported)

Digital Output Interface:

- 8-bit BT656, With Embedded Sync
- 8-bit BT656, With External Sync (Not supported)

Automatic video standard detection (NTSC/PAL/SECAM) and switching

TVP5147 video decoder is an independent interface which is being configured from the VPIF driver. TVP5147 is I2C slave device. TVP5147 driver configures TVP5147 device using I2C interface.

17.6.4.1 Interface Functions

TVP5147 exports its function table pointer through TVP5147_Fxns global variable as defined below:

```
/* Decoder (TVP5147) driver function table */

extern EDC_Fxns TVP5147_Fxns;

/* TVP5147 EDC function table */

EDC_Fxns TVP5147_Fxns =

{

    &TVP5147_open,

    &TVP5147_close,

    &TVP5147_ctrl

};
```

To use TVP5147, application shall pass this function table pointer as part of channel parameters ("*capEdcTbl*" of *Vpif_CapChanParams*) during channel creation of capture device. This will associate the EDC driver instance with the corresponding channel instance.

As shown in the "Interaction between VPIF and EDC driver", when application calls FVID_create(), VPIF driver will internally call TVP5147_open function. This will power on TVP5147 device, initialize I2C driver for serial communication and configures the decoder for default settings. One of the strings "*/I2C0/TVP5147_1/0x5D*" or "*I2C0/TVP5147_0/0x5C*" should be passed as argument to TVP5147_open function to open the corresponding decoder channel.

> *The string passed should depend on for which VPIF channel the capture device is opened.*

To configure TVP5147, application has to call FVID_control() function with *Vpif_IOCTL_CMD_MAX* + *Edc_IOCTL_CMD_MAX* + TVP5147 IOCTL (as shown in below table) as command. This will internally call TVP5147_ctrl function. Once the application deletes the channel, Vpif driver internally delete the TVP5147 driver instance and close the I2C driver as well using TVP5147_close.

17.6.4.2 Constants & Enumerations

17.6.4.2.1 Tvp5147_OutputFormat

"*Tvp5147.h*" file contains *Tvp5147_OutputFormat* enum that is passed while calling *EDC_IOCTL_CONFIG IOCTL* for TVP5147 from the application. This enum gives available output format of data for Tvp5147 decoder. The members of this enum are explained below:

Enum Members	Description
Tvp5147_OutputFormat_YCBCR422	Interlaced YCbCr 422 output.

17.6.4.2.2 Tvp5147_AnalogFormat

"*Tvp5147*.h" file contains *TVP5147_AnalogFormat* enum that is passed while calling *Edc_IOCTL_CONFIG IOCTL* for TVP5147 from the application. This enum tells about the cable connection from the input device to the EVM. The members of this enum are explained below:

Enum Members	Description
Tvp5147_AnalogFormat_SVIDEO	S-video selection. SVIDEO(Y/C) IN cable used.
Tvp5147_AnalogFormat_COMPOSITE	Composite video input. CVBS IN cable used.

17.6.4.2.3 Tvp5147_Std

"*Tvp5147.h*" file contains *TVP5147_Std* enum that is passed while calling *EDC_IOCTL_CONFIG IOCTL* for TVP5147 from the application. This enum tells about the video standard used. The members of this enum are explained below:

Enum Members	Description
Tvp5147_Std_INVALID	Invalid Input.
Tvp5147_Std_AUTO	Auto switch mode of operation. The standard will be detected automatically
Tvp5147_Std_NTSC720	Analog input standard is NTSC
Tvp5147_Std_PAL720	Analog input standard is PAL

17.6.4.2.4 Tvp5147_ControlId

"*Tvp5147.h*" file contains *Tvp5147_ControlId* enum that is passed as a part of call to *Tvp5147_IOCTL_SET_CONTROL* IOCTL for TVP5147 from the application. This enum is used for control settings for TVP5147. The members of this enum are explained below:

Enum Members	Description
Tvp5147_ControlId_AUTO_GAIN	Gain control. A value of 0 sets Manual gain and value of 1 enables auto gain.
Tvp5147_ControlId_BRIGHTNESS	Brightness control. A value of 255 (bright), 128 (default), 0 (dark). Brightness supported is (0-255).
Tvp5147_ControlId_CONTRAST	Contrast control (Luminance Contrast). A value of 255(maximum contrast), 128 (default), 0 (minimum contrast). Contrast supported is - Contrast: 0 - 255
Tvp5147_ControlId_HUE	Hue control. It can have only 3 values either 0x80(-180 degrees) or 0x7F (+180 degrees) or 0(0 degrees). HUE does not apply to component video.
Tvp5147_ControlId_SATURATION	Saturation (Chrominance Saturation) control. A value of 255 (maximum), 128 (default), 0 (no color) Saturation supported is -Saturation: 0 - 255

17.6.4.2.5 Tvp5147_IOCTL

"*Tvp5147.h*" file contains *Tvp5147_IOCTL* enum that is passed as a part of call to ctrl() for TVP5147 from the application. TVP5147 driver provides support for different IOCTL commands as shown below. Application can call FVID_control() with one of below specified IOCTL command(in a special way) and corresponding argument to configure TVP5147.

TVP5147 IOCTL Command	Argument	Description
Edc_IOCTL_CONFIG	*Tvp5147_ConfParams **	Configure the TVP5147 decoder.
Edc_IOCTL_RESET	*None*	Reset the decoder
Edc_IOCTL_SET_REG	*Edc_RegData**	Write to decoder register
Edc_IOCTL_GET_REG	*Edc_RegData**	Read from decoder register.
Tvp5147_IOCTL_POWERDOWN	*None*	This ioctl will power down the TVP5147 decoder.
Tvp5147_IOCTL_POWERUP	*None*	This ioctl will power up the TVP5147 decoder.
Tvp5147_IOCTL_SET_CONTROL	*Tvp5147_Control **	Set the various control for TVP5147.
Tvp5147_IOCTL_SET_SLICE_VBI_SERVICE	*Uint32 **	Set Slice VBI services for TVP5147. NOTE: This ioctl does not check whether current set standard supports the slice service or not. It just sets them.
Tvp5147_IOCTL_READ_SLICE_VBI_DATA	*FVID_SliceFrame **	Reads Slice VBI data for TVP5147. This IOCTL will be used by VPIF layer to get VBI data and put the data inside the vpif Frame packet

Tvp5147_IOCTL_READ_SLICE_VBI_DATA should only be called from vpif driver and not from application

To configure TVP5147 using generic EDC IOCTL, application has to call FVID_control() function with *Vpif_IOCTL_CMD_MAX* + *Edc_IOCTL_xxxx* as command. Here xxxx is generic EDC IOCTL command.

The example below shows how to use generic EDC IOCTL to write the register of the decoder:

```
Edc_RegData regval;

Uint8 val;

regval.startReg = 0x02u;

regval.noRegToRW = 1u;
```

```
val = 0x01;

regval.value = &val;

status = FVID_control(capChInfo.chanHandle,

             Vpif_IOCTL_CMD_MAX + Edc_IOCTL_SET_REG,

             (Ptr)&regval);

if (IOM_COMPLETED != status)

{

    LOG_printf(&trace, "Failed to set reg. of decoder");

}
```

- *To configure TVP5147 using specific TVP5147 IOCTL, application has to call FVID_control() function with `Vpif_IOCTL_CMD_MAX` + `Edc_IOCTL_CMD_MAX` + `TVP5147_IOCTL_xxxx` as command. Here xxxx is specific TVP5147 IOCTL command.*

The example below shows how to use specific TVP5147 IOCTL to set control parameter (saturation) of the decoder:

```
//Set saturation

Tvp5147_Control control;

control.tvpVidtype = Edc_VideoType_SD;

control.tvpCtrlId = Tvp5147_ControlId_SATURATION;

control.tvpValue = 0;

/* Configure TVP5147 */

status = FVID_control(capChInfo.chanHandle,

             Vpif_IOCTL_CMD_MAX + Edc_IOCTL_CMD_MAX +

             Tvp5147_IOCTL_SET_CONTROL,

             (Ptr)&control);

if (IOM_COMPLETED != status)

{

    LOG_printf(&trace, "Set control for saturation
failed");

}
```

- *For EDC related ioctls, FVID_control() will internally call TVP5147_ctrl function.*

17.6.4.3 Data Structures

This section describes TVP5147 data structures exposed to the application.

17.6.4.3.1 Tvp5147_ConfParams

"*Tvp5147.h*" file contains *Tvp5147_ConfParams* data structure that is passed as an argument while calling *Edc_IOCTL_CONFIG* ioctl for TVP5147 from the application. This structure contains configuration parameters for TVP5147 decoder. The members of this structure are explained below:

Structure Members	Description
tvpAnaFmt	Indicates analog input format for TVP5147. Analog format defined by enum *Tvp5147_AnalogFormat*.
tvpMode	Indicates operation mode (NTSC/PAL) for TVP5147. Operation mode defined by enum *Tvp5147_Std*.
tvpOutFmt	Indicates output format for TVP5147. Output format defined by enum *Tvp5147_OutputFormat*.
tvpServices	Type of Slice VBI service. Available values for this field are defined in "*Fvid.h*" file with FVID Slice VBI Service title. This should be passed appropriately according to the Video standard mode desired. CAUTION: If wrong service is sent, the driver does not verify its validity.

17.6.4.3.2 Tvp5147_Control

"Tvp5147.h" file contains Tvp5147_Control data structure that is passed as an argument while calling Tvp5147_IOCTL_SET_CONTROL ioctl for TVP5147 from the application. This structure contains setting control data structure for TVP5147 decoder. The members of this structure are explained below:

Structure Members	Description
tvpVidtype	Video Type for this control feature. Video type defined by enum *Edc_VideoType*.
tvpCtrlId	Control Id defined for TVP5147. Control id defined by enum *Tvp5147_ControlId*.
tvpValue	Value to be written to the control register.

17.6.5 ADV7343 Encoder

The ADV7343 is a high speed, digital-to-analog video encoder. Six high speed, 3.3 V, 11-bit video DACs provide support for composite (CVBS), S-Video (Y/C), and component (YPrPb/RGB) analog outputs in either standard definition (SD), enhanced definition (ED), or high definition (HD) video formats.

The ADV7343 has a 24-bit pixel input port that can be configured in a variety of ways. SD video formats are supported over a SDR interface and ED/HD video formats are supported over SDR and DDR interfaces. Pixel data can be supplied in either the YCrCb or RGB color spaces.

It also supports embedded EAV/SAV timing codes, external video synchronization signals, and I2C and SPI communication protocols. Cable detection and DAC auto power-down features keep power consumption to a minimum.

On C6748 EVM, ADV7343 encoder is connected to the VPIF for BT.656 display. ADV7343 encoder is used for NTSC/PAL SD resolution displays. The same encoder is connected to both channel 2 and 3 but channel 3 connection on EVM does not allow it to be used for SD display.

ADV7343 input and output interface details are given below:

Analog Output Interface:

- S-video
- Component (RGB/YPrPb) (Not supported)
- Composite

Digital Input Interface:

- Embedded Sync
- External Sync (Not supported)

ADV7343 video encoder is an independent interface which is being configured from the Vpif driver. ADV7343 is I2C slave device. ADV7343 driver configures ADV7343 device using I2C interface.

17.6.5.1 Interface Functions

ADV7343 exports its function table pointer through *ADV7343_Fxns* global variable as defined below:

```
/* Encoder (ADV7343) driver function table */

extern EDC_Fxns ADV7343_Fxns;

/* ADV7343 EDC function table */

EDC_Fxns ADV7343_Fxns =

{

    &ADV7343_open,

    &ADV7343_close,

    &ADV7343_ctrl

};
```

To use ADV7343, application shall pass this function table pointer as part of channel parameters ("*dispEdcTbl*" of *Vpif_DisChanParams*) during channel creation of display device. This will associate the EDC driver instance with the corresponding channel instance.

As shown in the "Interaction between VPIF and EDC driver", when application calls FVID_create(), VPIF driver will internally call ADV7343_open function. This will power on ADV7343 device, initialize I2C driver for serial communication, and configures the encoder for default settings. String of type "*/I2C0/ADV7343/0x2A*" should be passed as argument to ADV7343_open function to open the corresponding encoder channel.

- *The string passed should depend on for which VPIF channel the display device is opened.*
- *VPIF channel 3 cannot be used for SD display as the ADV7343 connection is not available for BT656 display.*

To configure ADV7343, application has to call FVID_control() function with *Vpif_IOCTL_CMD_MAX* + *Edc_IOCTL_CMD_MAX* + ADV7343 IOCTL (as shown in below table) as command. This will internally call ADV7343_ctrl function. Once the application deletes the channel, Vpif driver internally delete the ADV7343 driver instance and close the I2C driver as well using ADV7343_close.

17.6.5.2 Constants & Enumerations

17.6.5.2.1 Adv7343_InputFormat

"*Adv7343.h*" file contains *Adv7343_InputFormat* enum that is passed while calling *EDC_IOCTL_CONFIG* IOCTL for ADV7343 from the application. This enum gives available input data format for ADV7343 encoder. The members of this enum are explained below:

Enum Members	Description
Adv7343_InputFormat_YCBCR422	Interlaced YCbCr 422 input.

17.6.5.2.2 Adv7343_AnalogFormat

"*Adv7343.h*" file contains *ADV7343_AnalogFormat* enum that is passed while calling *EDC_IOCTL_CONFIG* IOCTL for ADV7343 from the application. This enum gives available analog connection from EVM (ADV7343 encoder) to output display device. The members of this enum are explained below:

Enum Members	Description
Adv7343_AnalogFormat_SVIDEO	S-video selection. SVIDEO(Y/C) out cable used.
Adv7343_AnalogFormat_COMPOSITE	Composite video input. CVBS out cable used.

17.6.5.2.3 Adv7343_Std

"*Adv7343.h*" file contains *Adv7343_Std* enum that is passed while calling *EDC_IOCTL_CONFIG* IOCTL for ADV7343 from the application. This enum gives available video operation mode for ADV7343 encoder .The members of this enum are explained below:

Enum Members	Description
Adv7343_Std_INVALID	Invalid Input.
Adv7343_Std_AUTO	Auto switch mode of operation. The standard will be detected automatically
Adv7343_Std_NTSC720	Analog input standard is NTSC
Adv7343_Std_PAL720	Analog input standard is PAL

17.6.5.2.4 Adv7343_ControlId

"*Adv7343.h*" file contains *Adv7343_ControlId* enum that is passed as a part of call to *Adv7343_IOCTL_SET_CONTROL* IOCTL for ADV7343 from the application. This enum is used for control settings for ADV7343. The members of this enum are explained below:

Enum Members	Description
Adv7343_ControlId_BRIGHTNESS	Brightness control. Brightness supported is (0-127); Values in the range of 0x3F to 0x44 could result in an invalid output signal.
Adv7343_ControlId_HUE	Hue control. Hue Supported is - For normal operation (zero adjustment); value is set to 0x80. Values 0xFF and 0x00 represent the

	upper and lower limits, respectively, of the attainable adjustment in NTSC mode. Values 0xFF and 0x01 represent the upper and lower limits, respectively, of the attainable adjustment in PAL mode.

17.6.5.2.5 Adv7343_GammaCurve

"*Adv7343.h*" file contains *Adv7343_GammaCurve* enum that is passed while calling *Adv7343_IOCTL_SET_GAMMA* for ADV7343 from the application. This enum is used to select gamma curve on ADV7343 encoder. The members of this enum are explained below:

Enum Members	Description
Adv7343_GammaCurve_A	Gamma curve A.
Adv7343_GammaCurve_B	Gamma curve B.

17.6.5.2.6 Adv7343_IOCTL

"*Adv7343.h*" file contains *Adv7343_IOCTL* enum that is passed as a part of call to ctrl() for ADV7343 from the application. ADV7343 driver provides support for different IOCTL commands as shown below. Application can call FVID_control() with one of below specified IOCTL command(in a special way) and corresponding argument to configure ADV7343.

ADV7343 IOCTL Command	Argument	Description
Edc_IOCTL_CONFIG	*Adv7343_ConfParams**	Configure the ADV7343 encoder.
Edc_IOCTL_RESET	*None*	Reset the encoder
Edc_IOCTL_SET_REG	*Edc_RegData **	Write the register to encoder
Edc_IOCTL_GET_REG	*Edc_RegData **	Read the register from encoder
Adv7343_IOCTL_POWERDOWN	*None*	This ioctl will power down the ADV7343 encoder.
Adv7343_IOCTL_POWERUP	*None*	This ioctl will power up the ADV7343 encoder.
Adv7343_IOCTL_ENABLE_COLORBAR	*Bool **	This ioctl will enable or disable ADV7343 internal color bar. The value of TRUE - Enables color bar and FALSE - Disables color bar
Adv7343_IOCTL_SET_CONTROL	*Adv7343_Control **	Set control for ADV7343.
Adv7343_IOCTL_SET_GAMMA	*Adv7343_GammaParams **	Set gamma for ADV7343.
Adv7343_IOCTL_SET_SLICE_VBI_SERVICE	*Uint32 **	Set Slice VBI services for ADV7343. NOTE: This ioctl does not check whether current set standard supports the slice

		service or not. It just sets them.
Adv7343_IOCTL_WRITE_SLICE_VBI_DATA	*FVID_SliceFrame **	Writes Slice VBI data for ADV7343. This IOCTL will be used by VPIF layer to get VBI data and put it inside the vpif Frame packet

- *Adv7343_IOCTL_WRITE_SLICE_VBI_DATA should be called from Vpif driver and not from application.*
- *To configure ADV7343 using generic EDC IOCTL, application has to call FVID_control() function with Vpif_IOCTL_CMD_MAX + Edc_IOCTL_xxxx as command. Here xxxx is generic EDC IOCTL command.*

The example below shows how to use generic EDC IOCTL to configure for composite output of the encoder:

```
Adv7343_ConfParams vDisParamsEncoder =

{

    Adv7343_AnalogFormat_COMPOSITE,/* AnalogFormat      */

    Adv7343_Std_AUTO,              /* Video std         */

    Adv7343_InputFormat_YCBCR422,  /* InputFormat       */

    Fvid_SLICE_VBI_SERVICES_NONE   /* slice vbi service */

};

/* Configure ADV7343 */

status = FVID_control(disChInfo.chanHandle,

    Vpif_IOCTL_CMD_MAX + Edc_IOCTL_CONFIG,

    (Ptr)&vDisParamsEncoder);

if (IOM_COMPLETED != status)

{

    LOG_printf(&trace, "Failed to config encoder");

}
```

- *To configure ADV7343 using specific ADV7343 IOCTL, application has to call FVID_control() function with Vpif_IOCTL_CMD_MAX + Edc_IOCTL_CMD_MAX + ADV7343_IOCTL_xxxx as command. Here xxxx is specific ADV7343 IOCTL command.*

The example below shows how to use specific ADV7343 IOCTL to set control parameter (hue) of the encoder:

```
// Set hue

Adv7343_Control control;

control.advVidtype = Edc_VideoType_SD;

control.advCtrlId = Adv7343_ControlId_HUE;
```

```
control.advValue = 0xFF;

/* Configure ADV7343 */

status = FVID_control(disChInfo.chanHandle,

               Vpif_IOCTL_CMD_MAX + Edc_IOCTL_CMD_MAX +

               Adv7343_IOCTL_SET_CONTROL,

               (Ptr)&control);

if (IOM_COMPLETED != status)

{

    LOG_printf(&trace, "Set control for Hue failed");

}
```

The FVID_control() call for the EDC device will internally call Adv7343_ctrl function.

17.6.5.3 Data Structures

This section describes ADV7343 data structures exposed to the application.

17.6.5.3.1 Adv7343_ConfParams

"Adv7343.h" file contains Adv7343_ConfParams data structure that is passed as an argument while calling Edc_IOCTL_CONFIG ioctl for ADV7343 from the application. This structure contains configuration parameters for ADV7343 encoder. The members of this structure are explained below:

Structure Members	Description
advAnaFmt	Indicates analog output format for ADV7343. Analog format defined by enum Adv7343_AnalogFormat.
advMode	Indicates operation mode (NTSC/PAL) for ADV7343. Operation mode defined by enum Adv7343_Std.
advInFmt	Indicates the selection for digital input format for ADV7343. Input format defined by enum Adv7343_InputFormat.
advServices	Type of Slice VBI service. Available values for this field are defined in "Fvid.h" file with FVID Slice VBI Service title. This should be passed appropriately according to the Video standard mode desired. CAUTION : If wrong service is sent, the driver does not verify its validity

17.6.5.3.2 Adv7343_Control

"Adv7343.h" file contains Adv7343_Control data structure that is passed as an argument while calling Adv7343_IOCTL_SET_CONTROL ioctl for ADV7343 from the application. This structure contains setting control data structure for ADV7343 encoder. The members of this structure are explained below:

Structure Members	Description
advVidtype	Video Type for this control feature. Video type defined by enum Edc_VideoType

advCtrlId	Control Id defined for ADV7343. Control id defined by enum Adv7343_ControlId
advValue	Value to be written to the control register

17.6.5.3.3 Adv7343_GammaParams

"Adv7343.h" file contains Adv7343_GammaParams data structure that is passed as an argument while calling Adv7343_IOCTL_SET_GAMMA IOCTL for ADV7343 from the application. This structure contains gamma parameter settings for ADV7343 encoder. The members of this structure are explained below:

Structure Members	Description
type	Video Type for this gamma feature. Video type defined by enum Edc_VideoType
enGamma	Enables/disables gamma correction TRUE: Enable FALSE: Disable
curve	Selects gamma correction curve. Gamma curve defined by Adv7343_GammaCurve.
coeff[ADV7343_MAX_GAMMA_COEFFS]	Gamma correction coefficients.

17.6.6 MT9T001 Image Sensor

The MT9T001 Image sensor is a QXGA-format ½-inchCMOS active-pixel digital image sensor with an active imaging pixel array of 2048H x 1536V. It incorporates sophisticated camera functions on-chip such as windowing; column and row skip mode and snapshot mode. It is a programmable simple two serial wire interface.

The image sensor can be operated in its default mode or programmed by the user for frame size, exposure, gain setting, and other parameters. An on-chip analog-to-digital converter (ADC) provides 10bits per pixel.

The MT9T001 produces extraordinarily clear, sharp digital pictures, and its ability to capture both continuous video and single frames makes it the perfect choice for a wide range of consumer and industrial applications, including digital still cameras, digital video cameras, and PC cameras.

Pixel Data Format

The MT9T001 pixel array is configured as 2,112 columns by 1,568 rows. There are 2,057 columns by 1,545 rows of optically active pixels, which provide a four-pixel boundary around the QXGA (2,048 x 1,536) image to avoid boundary effects during color interpolation and correction.

The MT9T001 uses a Bayer color pattern. The even-numbered rows contain green and red color pixels, and odd-numbered rows contain blue and green color pixels. The even-numbered columns contain green and blue color pixels; odd-numbered columns contain red and green color pixels.

Output Data Format

The MT9T001 image data is read out in a progressive scan. Valid image data is surrounded by horizontal blanking and vertical blanking. The amount of horizontal blanking and vertical blanking is programmable.

MT9T001 image sensor is an independent interface with the vpif driver. MT9T001 image sensor will be configured, through IOCTL of vpif driver. MT9T001 is I2C slave

device. MT9T001 image sensor peripheral registers are configured using I2C driver. I2C will communicate with MT9T001 using the slave address 0x5D. Refer to MT9T001 specs for more detail.

17.6.6.1 Interface Functions

MT9T001 exports its function table pointer through MT9T001_Fxns global variable as defined below:

```
/* External Image Sensor (MT9T001) driver function table */

extern EDC_Fxns MT9T001_Fxns;

/* MT9T001 EDC function table */

EDC_Fxns MT9T001_Fxns =

{

    &MT9T001_open,

    &MT9T001_close,

    &MT9T001_ctrl

};
```

To use MT9T001, application shall pass this function table pointer as part of channel parameters ("capEdcTbl" of Vpif_CapChanParams) during channel creation of raw capture device. This will associate the EDC driver instance with the corresponding channel instance.

Only channel 0 of VPIF driver can be opened as RAW capture device.

MT9T001 image sensor driver is an independent interface which is called from the Vpif driver. As shown in the "Interaction between VPIF and EDC driver", when application calls FVID_create() it will call MT9T001_open function. This will initialize the MT9T001 chip, initialize I2C driver for serial communication and configures the sensor for default configuration. It configures the I2C for further register read and write of MT9T001 image sensor.

String of type "/I2C0/MT9T001/0x5D" should be passed as argument to MT9T001_open function to open the corresponding channel.

To configure MT9T001, application has to call FVID_control() function with Vpif_IOCTL_CMD_MAX + Edc_IOCTL_CMD_MAX + MT9T001 IOCTL (as shown in below table) as command. This will internally call MT9T001_ctrl function.

Once the VPIF driver deletes the channel, it will delete the MT9T001 driver instance and close the I2C driver as well using MT9T001_close function pointer.

17.6.6.2 Constants & Enumerations

17.6.6.2.1 MT9T001_StandardFormat

"Mt9t001.h" file contains MT9T001_StandardFormat enum that is passed while calling EDC_CONFIG IOCTL for MT9T001 from the application. This enum gives available various video format supported by MT9T001 driver. The value can be used to configure the MT9T001 image sensor with specified standard format. The members of this enum are explained below:

```
typedef enum MT9T001_StandardFormat_t
```

```
{
    MT9T001_MODE_VGA,
    MT9T001_MODE_SVGA,
    MT9T001_MODE_XGA,
    MT9T001_MODE_480P,
    MT9T001_MODE_576P,
    MT9T001_MODE_720P,
    MT9T001_MODE_1080P,
    MT9T001_MODE_QXGA,
    MT9T001_MODE_UXGA,
    MT9T001_MODE_SXGA
}MT9T001_StandardFormat;
```

17.6.6.2.2 Generic IOCTL

"Mt9t001.h" file contains Mt9t001_IOCTL enum that is passed as a part of call to ctrl() for MT9T001 from the application. MT9T001 driver provides support for different IOCTL commands as shown below. Application can call FVID_control() with one of below specified IOCTL command and corresponding argument to configure MT9T001.

MT9T001 IOCTL Command	Argument	Description
Edc_IOCTL_CONFIG	*Mt9t001_ConfParams **	Configure the MT9T001 sensor resolution.
Edc_IOCTL_RESET	*None*	Reset the MT9T001 sensor.
Edc_IOCTL_SET_REG	*Edc_RegData **	Write register of MT9T001 sensor. This IOCTL is supported with restriction that only one register can be written to the MT9T001 device also note the way register values should be passed.
Edc_IOCTL_GET_REG	*None*	This IOCTL is not supported by MT9T001 device as some registers when the driver tries to read back and verify it will give errors.

- *There are no specific ioctls supported for MT9T001 device*
- *Edc_IOCTL_GET_REG is not supported by MT9T001 sensor driver.*

The example below shows how to use generic EDC ioctl to change the resolution of the sensor:

```
Vpif_ConfigParams chResolution;
```

```
chResolution.mode = Vpif_VideoMode_RAW_UXGA;
status = FVID_control(rawChInfo.chanHandle,
             Vpif_IOCTL_CMD_CHANGE_RESOLUTION,
             &chResolution);
if (IOM_COMPLETED != status)
{
    LOG_printf(&trace, "Failed to change resolution");
}
```

17.6.6.3 Data Structures

This section describes MT9T001 data structures exposed to the application.

17.6.6.3.1 MT9T001_FormatParams

"*Mt9t001.h*" file contains *Mt9t001_ConfParams* data structure that is part of *Mt9t001_ConfParams* structure. This contains format structure for changing the MT9T001 external image sensor resolution. Most members of these structures directly reflect the MT9T001 sensor register settings. The driver **does not** check the validity of these parameters. The members of this structure are explained below:

Structure Members	Description
columnSize	Value to be written in Col Size Register: 0x04
rowSize	Value to be written in Row Size Register: 0x03
hBlank	Value to be written in Horizontal Blanking Register: 0x05
vBlank	Value to be written in Vertical Blanking Register: 0x06
shutterWidth	Value to be written in Shutter Width Register - lower mask: 0x09 and Shutter Width Register - upper: 0x08
blackLevel	Value to be written in Black Level Register: 0x49
pixelClockCtrl	Value to be written in Pixel Clock Control Register: 0x0A
rowStart	Value to be written in Row Start Register: 0x01
colStart	Value to be written in Col Start Register: 0x02

17.6.6.3.2 Mt9t001_ConfParams

"*Mt9t001.h*" file contains *Mt9t001_Control* data structure that is passed as an argument while calling *Edc_IOCTL_CONFIG* for MT9T001 from the application. This structure contains parameters to change the resolution of MT9T001 sensor. The members of this structure are explained below:

Structure Members	Description
fmtParams	If not NULL, indicates the individual parameters are sent by application and they should be set instead of driver "*stdFormat*" for a standard. If NULL then parameters for "*stdFormat*" inside the driver are set.

stdFormat	Indicates Standard format for MT9T001. This field is not valid if "*fmtParams*" is not NULL.

17.7 Power Management Implementation

17.7.1 DVFS

If there is a request from application for changing the set points (V/F pair), the driver takes care of this and change to the appropriate state. Before calling the set point change event the application should stop VPIF using the IOCTL. Once the set point is changed the operation can be started back.

> *Please note that for changing the set point the VPIF driver should be stopped using the* `Vpif_IOCTL_CMD_STOP` *IOCTL.*

17.7.2 Sleep

If there is a request from application for moving to sleep state (SLEEP/STANDBY/DEEPSLEEP), the driver takes care of these events and change to the appropriate state. Before calling the sleep, the application should stop VPIF using the IOCTL. Once the set point is change the operation can be started back.

> *Please note that for changing the set point the VPIF driver should be stopped using the* `Vpif_IOCTL_CMD_STOP` *IOCTL.*

17.8 EVM Initialization

For the ease of development of application, EVM related code is split and placed inside the platform folder. The header file for VPIF related EVM initialization is placed at *platforms\evm6748\Vpif_evmInit.h*. This section discusses about the initialization details and structures used for EVM initialization.

> *The tci file required for I2C device creation is also defined here and is named as "vpif.tci". Application can choose to use this tci file directly or may define one of its own.*

17.8.1 Enumeration

17.8.1.1 EvmInit_VpifChannel

"*Vpif_evmInit.h*" file contains enum *EvmInit_VpifChannel* that is passed to the EVM configuration API. This enumeration tells for which channel, configuration should be set. The enum string itself is self explanatory of the channel number. Following are the enums exposed:

```
typedef enum EvmInit_VpifChannel_t

{

    EvmInit_VpifChannel_0,

    EvmInit_VpifChannel_1,

    EvmInit_VpifChannel_2,

    EvmInit_VpifChannel_3,

    EvmInit_VpifChannel_BOTHCAPCH,/* For RAW Capture use
both capture channel */

    EvmInit_VpifChannel_BOTHDISPCH/* Not Supported */
```

```
}EvmInit_VpifChannel;
```

Please note that for raw capture VPIF uses both channel 0 and 1, so `EvmInit_VpifChannel_BOTHCAPCH` should be used as a parameter for EVM initialization.

17.8.2 Interface details

17.8.2.1 configureVpif0

Syntax

Void configureVpif0(EvmInit_VpifChannel channelNo, Bool isHd);

Parameters

channelNo

Channel number depending upon the type of usage for which the application is going to open the VPIF channel.

isHd

This parameter should be FALSE and reserved for future use.

Return Value

None

Description

An application will call configureVpif0() to initialize the VPIF device for the required usage. Depending up on the "*channelNo*" passed all EVM related initialization is done. This includes setting up of PINMUXES of VPIF and I2C, enabling clocks and enabling the path of VPIF channel to the encoder or decoder.

Constraints

- This function should be called from task context.
- This function should be called before any call to the VPIF driver is made by the application.

Example

The example below shows the call for configuration related to capture channel 0 of VPIF

```
/* Configure VPIF Input Video Clocks */

configureVpif0(EvmInit_VpifChannel_0, FALSE);
```

17.9 Supporting "NEW" resolution

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

- For adding inside driver:
  - Add an enumeration in *Vpif_VideoMode* defined in *Vpif.h*
  - Define a macro like "*VPIF_SD_PARAMS*" and set the different parameters of type *Vpif_ConfigParams* for the resolution.
  - Add the macro to "*ch**n**Params*"; where **n** is the channel no for which resolution is supported.

- Increase the mode supported by the channel by increasing the value of "*Vpif_CHn_MAX_MODES*", where **n** is the channel no for which resolution is changed.

❖ For changing the resolution from the application, when channel is not created:

- Create the channel by passing the "*capStdMode*" parameter of capture channel or "*dispStdMode*" parameter of display channel, as *Vpif_VideoMode_NONE*.
- Update the desired resolution parameters by filling "*capVideoParams*" member of capture channel parameter or "*dispVideoParams*" member of display channel parameter.

❖ For changing the resolution from the application, when channel is created:

- Stop the channel if already started and free the frame buffers.
- Call the *Vpif_IOCTL_CMD_CHANGE_RESOLUTION* ioctl with "mode" parameter of *Vpif_ConfigParams* structure as *Vpif_VideoMode_NONE*. Update the remaining parameter of the structure as required for the resolution.
- Queue the buffers to the driver and start the channel.

17.10 EDMA3 Dependency

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

17.11 Known Issues

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

17.12 Limitations

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

17.13 Sample Application

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

17.13.1 Writing Applications for Vpif

This section provides guidance to user for writing own application for Vpif capture and display drivers.

17.13.1.1 File Inclusion

To write sample application user has to include following header files in the application:

1. **ti/pspiom/vpif/Fvid.h**

   This file contains FVID layer macros and structures. These macros are wrapper macros specifically for Video above GIO Layer.

2. **ti/pspiom/vpif/Vpif.h**

   This file contains VPIF parameters which are passed to driver at the time of VPIF driver registration with BIOS. This file also contains configuration structures and defines for capture/display channel configuration.

3. **ti/pspiom/vpif/Edc.h**

This file contains EDC specific defines, data types and function pointer table structure.

4. **ti/pspiom/platforms/evm6748/vpifedc/Tvp5147.h**

   This file contains the interfaces, data types and symbolic definitions that are needed by the application to configure the TVP5147 video decoder. This header files needs to be added at the application only if the input to VPIF module is from TVP5147 video decoder.

5. **ti/pspiom/platforms/evm6748/vpifedc/Adv7343.h**

   This file contains the interfaces, data types and symbolic definitions that are needed by the application to configure the ADV7343 video encoder. This header files needs to be added at the application only if the video output is configured from ADV7343 video encoder.

6. **ti/pspiom/platforms/evm6748/vpifedc/Mt9t001.h**

   This file contains the interfaces, data types and symbolic definitions that are needed by the application to configure the external MT9T001 sensor. This header files needs to be added at the application only if the RAW input to VPIF module is from external MT9T001 image sensor.

7. **ti/pspiom/platforms/evm6748/Vpif_evmInit.h**

   This file contains EVM related data type and interfaces required for initialization of different VPIF channels.

17.13.1.2 Buffer Management Strategy

17.13.1.2.1 Capture driver

Capture driver always returns the most recent frame captured and cycle through available buffers when application falls behind.

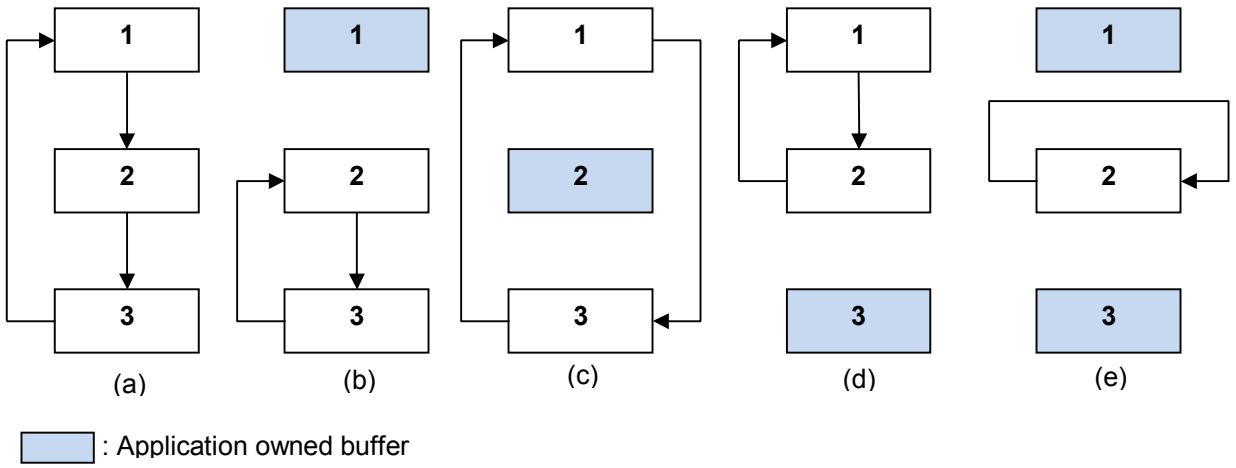


Figure 5. Capture Driver Buffer Management

All buffers are initially in the free queue and the driver cycles through them in a circular fashion. This is illustrated in (a).

When the application calls FVID_dequeue() and grabs the buffer with the most recent data from the driver, the driver then cycles through the rest of buffers. This is illustrated in figure from (a) to (b) and from (b) to (e).

When the application calls FVID_queue(), an empty buffer is returned by the application to the driver's free queue. This is illustrated in figure from (b) to (a) or from (e) to (b).

When the application calls FVID_exchange(), an empty buffer is returned by the application to the driver's free queue, and a buffer with the most recent data is given

17.13.1.2.2 Display driver

Display driver queues buffers for displaying from application and keep displaying the same frame when running out of buffers.

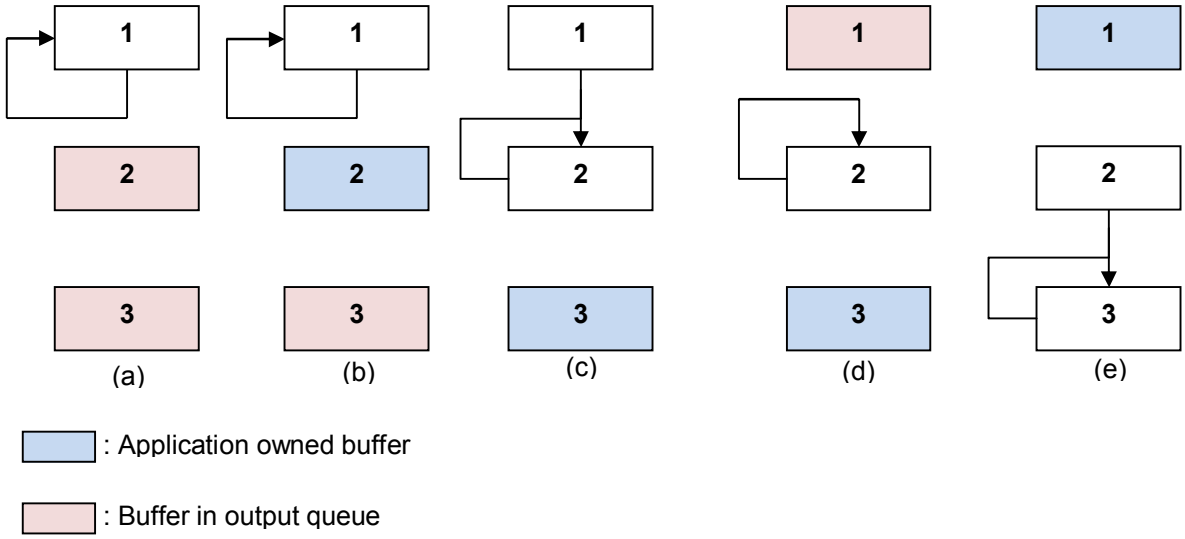


Figure 6. Display Driver Buffer Management

Initially all buffers except one are in the output queue, ready to be grabbed by the application. The driver repeatedly displays the current buffer. This is shown in figure (a).

When the application calls FVID_dequeue(), it gets a buffer from the driver. Application starts to fill data to it while the driver is still displaying its current buffer. This is shown in figure (a) to (b).

When the application calls FVID_queue(), it returns a buffer ready for display back to the driver. The driver, in turn, will set this buffer as its current buffer after it completes displaying the previous one. This is shown in figure (b) to (c) to (d).

When the application calls FVID_exchange(), it returns a buffer ready for display back to the driver and it requires an empty buffer from the driver. This is equivalent to calling FVID_queue() and FVID_dequeue() sequentially, as shown in figure (d) to (e).

17.13.1.3 SDRAM Frame Storage Format

The different ways the buffer can be storage formats that the driver supports are:

- Filed mode storage
- Frame mode storage

In case of FRAME based storage, buffer contains line interleaved top and bottom field data. In the FIELD based storage, top and bottom field data is stored separately in the buffer. The following figures show field and frame mode storages:

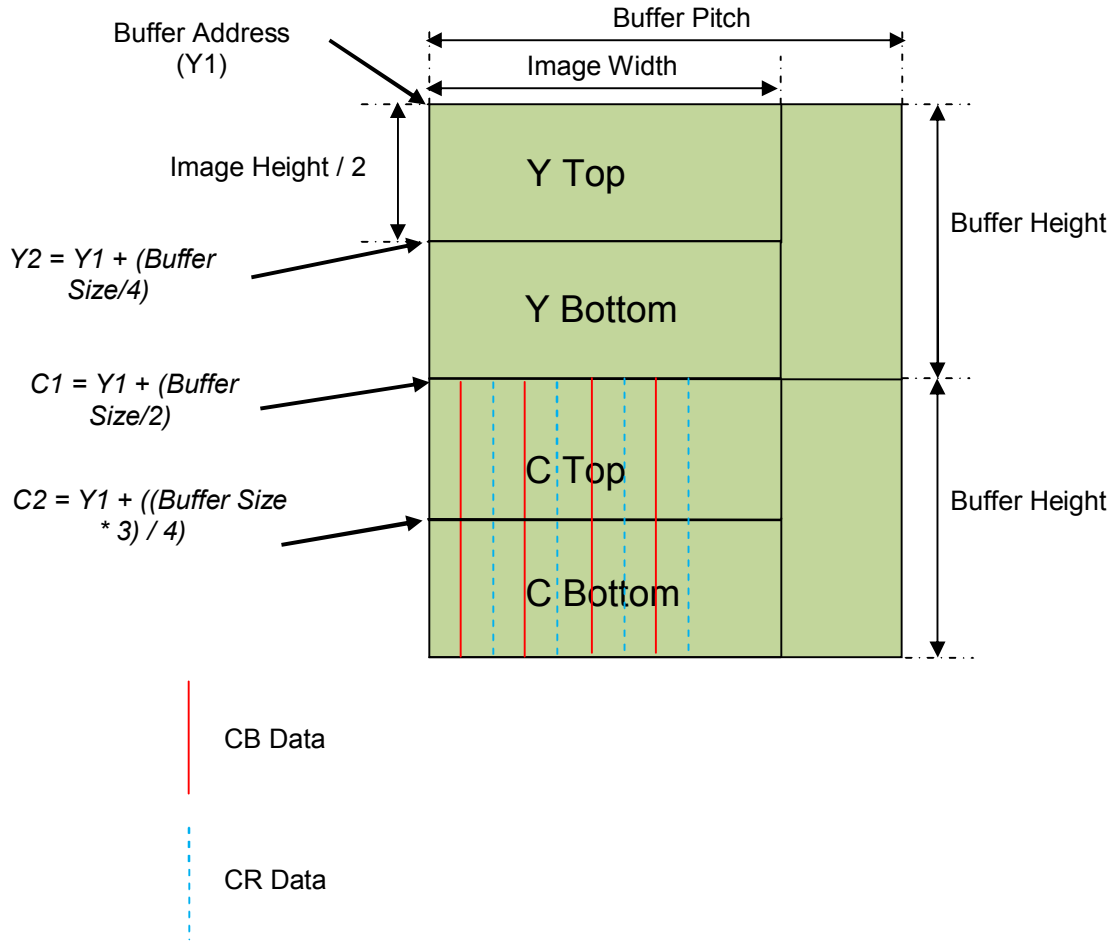


Figure 7. Field Mode Storage

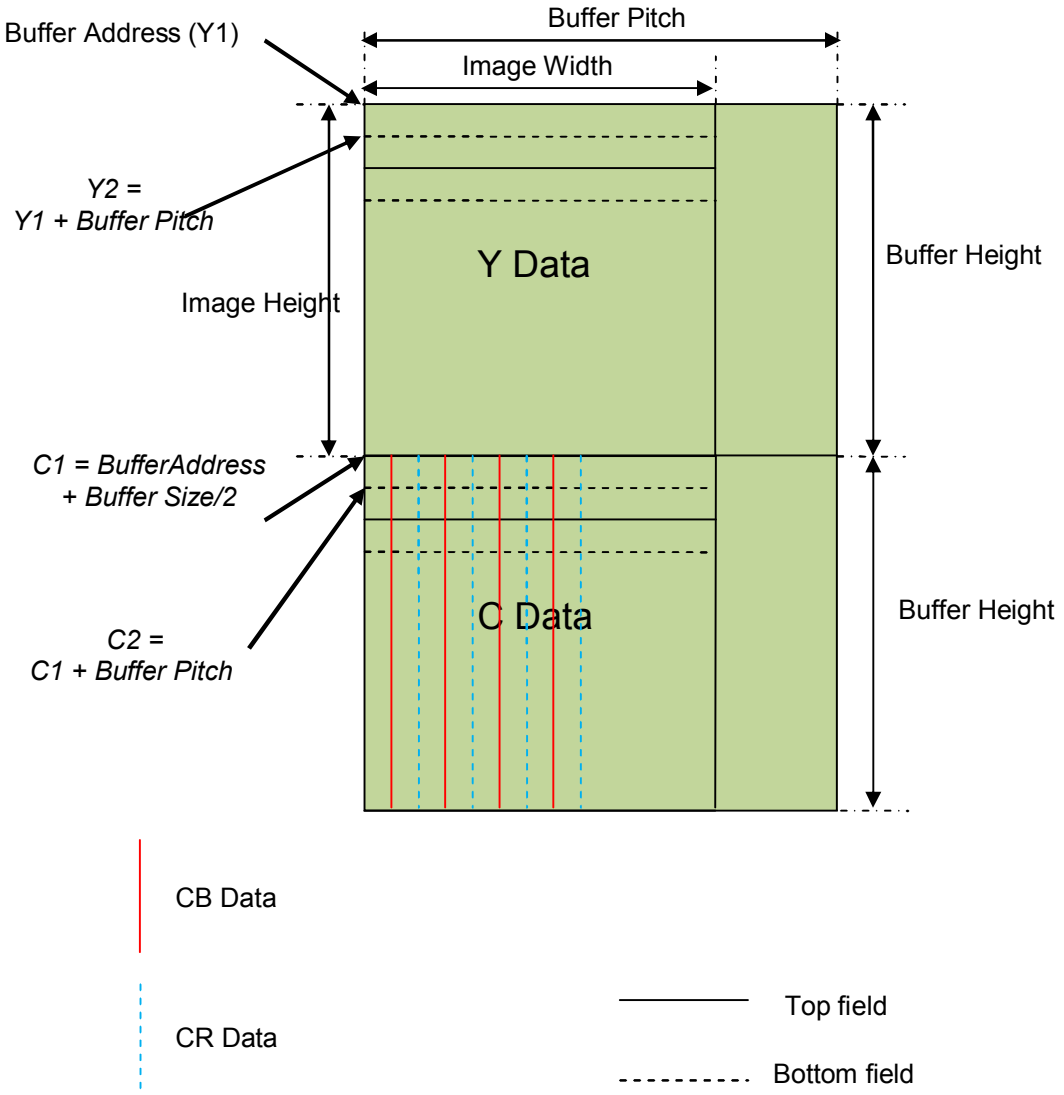


Figure 8. Frame mode storage

17.13.1.4 Slice Buffer Handling

If the slice service is enabled, driver checks whether current standard supports VBI or not. If it does not, driver returns error. It calls underlying decoder/encoder drivers function to set the sliced VBI services in the decoder/encoder device. Decoder or encoder driver checks for parameters validity and sets the services in the decoder/encoder hardware.

Please note that the encoder/decoder driver does not check when the service (CC, CGMS, or WSS) is enabled, the same standard (NTSC, PAL) is set or not. So if a slice service is enabled, driver does not checks whether the current standard supports that slice VBI or not.

Example:

```
/* Configure TVP5147 for closed caption slice service */
status = FVID control(capChInfo.chanHandle,
(Vpif_IOCTL_CMD_MAX + Edc_IOCTL_CMD_MAX +
```

```
Tvp5147_IOCTL_SET_SLICE_VBI_SERVICE),

                    (Ptr)&capSlice);

if (1 == status) /* Returns number of slice services set
*/

{

    status = IOM_COMPLETED;

}
```

FVID_SLICE_VBI_SERVICES_WSS_PAL is only supported PAL capturing/displaying and FVID_SLICE_VBI_SERVICES_CC_NTSC and FVID_SLICE_VBI_SERVICES_CGMS_NTSC are only supported on NTSC capturing/displaying. Size of the WSS, CGMS and CC data is 14 bits, 20 bits and 16 bits per field. They will have to be stored in the buffer as shown in the following figure:

Byte 0			B5	B4	B3	B2	B1	B0
Byte 1	B13	B12	B11	B10	B9	B8	B7	B6
Byte 2			B19	B18	B17	B16	B15	B14

Figure 9. Storage for captured CGMS data

Byte 0	B7	B6	B5	B4	B3	B2	B1	B0
Byte 1			B13	B12	B11	B10	B9	B8

Figure 10. Storage for captured WSS data

Byte 0	B7	B6	B5	B4	B3	B2	B1	B0
Byte 1	B15	B14	B13	B12	B11	B10	B9	B8

Figure 11. Storage for captured CC data for a field

Byte 0	B7	B6	B5	B4	B3	B2	B1	B0
Byte 1	B15	B14	B13	B12	B11	B10	B9	B8
Byte 2					B19	B18	B17	B16

Figure 12. Storage of display CGMS data

Byte 0	B7	B6	B5	B4	B3	B2	B1	B0
Byte 1			B13	B12	B11	B10	B9	B8

Figure 13. Storage of display WSS data

Byte 0	B7	B6	B5	B4	B3	B2	B1	B0
Byte 1	B15	B14	B13	B12	B11	B10	B9	B8

Figure 14. Storage of display CC data for a field

A single call to FVID_EXCHANGE will return all sliced VBI data belonging to one video frame. Application need to make sure that the buffer given to the encoder should be in byte aligned format.

Example:

```
status = FVID_exchange(capChInfo.chanHandle,
&(capChInfo.frame));

if (IOM_COMPLETED != status)

{

    LOG_printf(&trace, "Error in exchanging buffer");

}

else

{

    temp = 0;

    temp = (capChInfo.frame->vbiFrm.s1->fvidData[2] <<
14 | capChInfo.frame->vbiFrm.s1->fvidData[1] << 6 |
capChInfo.frame->vbiFrm.s1->fvidData[0]);

    temp1 = 0;

    temp1 = (capChInfo.frame->vbiFrm.s2->fvidData[2] <<
14 | capChInfo.frame->vbiFrm.s2->fvidData[1] << 6 |
capChInfo.frame->vbiFrm.s2->fvidData[0]);

    disChInfo.frame->vbiFrm.s1->fvidData[0] = temp &
0xFF;

    disChInfo.frame->vbiFrm.s1->fvidData[1] = ((temp >>
8) & 0xFF);

    disChInfo.frame->vbiFrm.s1->fvidData[2] = ((temp >>
16) & 0xFF);

    disChInfo.frame->vbiFrm.s2->fvidData[0] = temp1 &
0xFF;

    disChInfo.frame->vbiFrm.s2->fvidData[1] = ((temp1 >>
8) & 0xFF);

    disChInfo.frame->vbiFrm.s2->fvidData[2] = ((temp1 >>
16) & 0xFF);

    status = FVID_exchange(disChInfo.chanHandle,
&(capChInfo.frame));

    if (IOM_COMPLETED != status)

    {

        LOG printf(&trace, "Error in exchange ");

    }

}
```

17.13.1.5 Cache Coherency

Any buffer used for storing/retrieving data should be cache aligned, since they write/read, to/from SDRAM/DDR. The alignment parameter is passed by application

to the driver using the "*frmBufAlignment*" member of "*dispFbParams*" or "*capFbParams*", which are part of display and capture channel parameters.

Application is responsible to ensure cache coherency of video buffers, as the driver does nothing in this respect. This is because data is typically moved by DMA between fast on-chip RAM and slow off-chip SDRAM for faster CPU access. Furthermore, algorithms can use ping-pong buffer schemes to parallel the DMA transfer and the CPU execution, thus hiding most or all overhead associated with the data movement. If this is the case, cache flush and clean operations can be avoided by aligning the frame buffers to cache line boundaries. However, if the application does access these buffers directly, the application must flush or clean the cache to ensure cache coherency, the DMA accesses external memory directly through the EMIF, while the CPU goes through the cache when accessing the data.

❖ **Recommended Cache Operation in Application:**

In a simple loopback scenario, the application doesn't have to do any cache operations to ensure cache coherency if buffers are exchanged between drivers. But when the application access the video buffers through CPU say to run an algorithm or to copy capture buffer to display buffer using CPU, then the below cache operations are recommended for proper operation.

- Capture driver

  Before providing a buffer to capture driver, the entire buffer should be invalidated. Below code snippet illustrate this.

```
/* Invalidate the buffer before giving to capture driver */
BCACHE_inv(capChInfo.frame->frame.vpifFrm.y1, FRAME_SIZE, TRUE);
/* Give the old capture frame buffer back to driver and get the
recently captured frame buffer */
status = FVID_exchange(chanHandle, &frame);
```

- Display driver

  Before providing a buffer to display driver, the entire buffer should be flushed and invalidated. Below code snippet illustrate this.

```
/* Flush and invalidate the processed buffer so that the DMA reads
the processed data */
BCACHE_wbInv(capChInfo.frame->frame.vpifFrm.y1, FRAME_SIZE, TRUE);
/* Give the captured frame buffer to display driver and get a
free frame buffer for next capture */
status = FVID_exchange(chanHandle, &frame);
```

17.13.2 Sample Applications

17.13.2.1 Introduction

The sample application is a representative test program. They demonstrate the use of the Vpif driver. Initialization of Vpif driver is done by calling initialization function from BIOS.

The Vpif sample application instantiates the I2C driver statically in vpif.tci file, inside platforms\evm6748 folder. I2C driver is required to configure the EVM components, to select routing of signals to VPIF and later configuring the encoder and decoder. This file can be directly imported into an application's tcf script.

The vpifSample.tcf file contains the remaining BIOS configuration like the configuration of the event combiner etc. This helps to map the VPIF events to the CPU interrupts. 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_INT9").interruptSelectNumber = 2;

These lines configure the ECM module and map VPIF events to CPU interrupts. For example the VPIF event number is 95 which fall in ECM group 2. Here ECM group 2 is mapped to HWI_INT9.

The vpifSampleTask() task exercises the vpif driver. The configureVpif0() function inside the platform file takes care of configuring the pinmux (for VPIF, I2C and others, if required) and select the proper routing of Vpif signals to encoder and decoder and configure clocks at proper frequency , if required.

It uses FVID APIs to create VPIF driver channels and also to perform the IO operations.

1. SD Loop back

   The SD loop back application configures capture & display drivers and starts video loop back in NTSC/PAL resolution. By default the sample application captures one channel and displays in **NTSC** resolution. The capture channel is 1 and the display channel is 2. The connection of display is Composite and for capture the connection is S-video.

   Configuration options are provided (macros defined at the start of "*vpifSample_io.c*" file) to change the connection for display or capture and to change loop back for PAL resolution.

2. RAW Capture Loop back

   This sample application illustrates the RAW capture capability of Vpif driver. It captures RAW video from MT9T001 image sensor through VPIF channel 0 and displays the same in VPIF channel 2 in BT656 NTSC format which can be viewed in TV. The sample application does the conversion of Bayer pattern data from MT9T001 image sensor to RGB 888 and then YCbCr 422 so that it can be displayed back using vpif display channel 2. The display connection used is Composite.

   By default the sample application works in 8-bit RAW capture mode for **480P** and display in **480i** resolution.

   Configuration options are provided (macros defined at the start of "*vpifSample_io.c*" file) to change the display connection and change the number of frame buffers.

   - *The conversion is done by treating each 4x4 block of data is as a single pixel. The 2 green pixels are averaged together. The R and B are extracted. This type of processing uses only 1/4 of the captured resolution, i.e., 1/2 the number of pixels / line and 1/2 the lines*
   - *The conversion algorithm when used in release mode results in a jerky image display. This is because of the optimization by compiler.*

Therefore in release mode the file "`vpifSample_conversion.c`", is build with no optimization.

3. SD slice VBI Loop back

   The SD Slice VBI loop back application configures capture & display drivers for closed caption slice VBI service and starts video loop back in NTSC resolution. The application also enables closed caption slice service for both encoder and decoder. By default the sample application captures one channel and displays in NTSC resolution with closed caption enabled. The capture channel is 0 and the display channel is 2. The connection of both capture and display is composite.

   Please note that enabling Slice VBI data results in I2C read and write, and it may happen that I2C may result in some IO error. If an I2C error happens during exchange then the application should close the channels and disable the VBI service and reopen the channel. The error generally happen if there is not enough bandwidth available in the system.

❖ **Build Procedure:**

This sample can be built using following

Open

"<ProjectDir>/packages/ti/pspiom/examples/evm6748/vpifloopback/build/ccs3/**vpifSample.pjt**" for running SD loop back sample application

(OR)

Open

"<ProjectDir>/packages/ti/pspiom/examples/evm6748/vpifraw/build/ccs3/**vpifSample.pjt**" for running RAW capture loop back sample application

(OR)

Open

"<ProjectDir>/packages/ti/pspiom/examples/evm6748/vpifvbiloopback/build/ccs3/**vpifSample.pjt**" for running SD Slice service loop back sample application

This sample can be built using the CCS interface.

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

❖ **EVM Layout:**

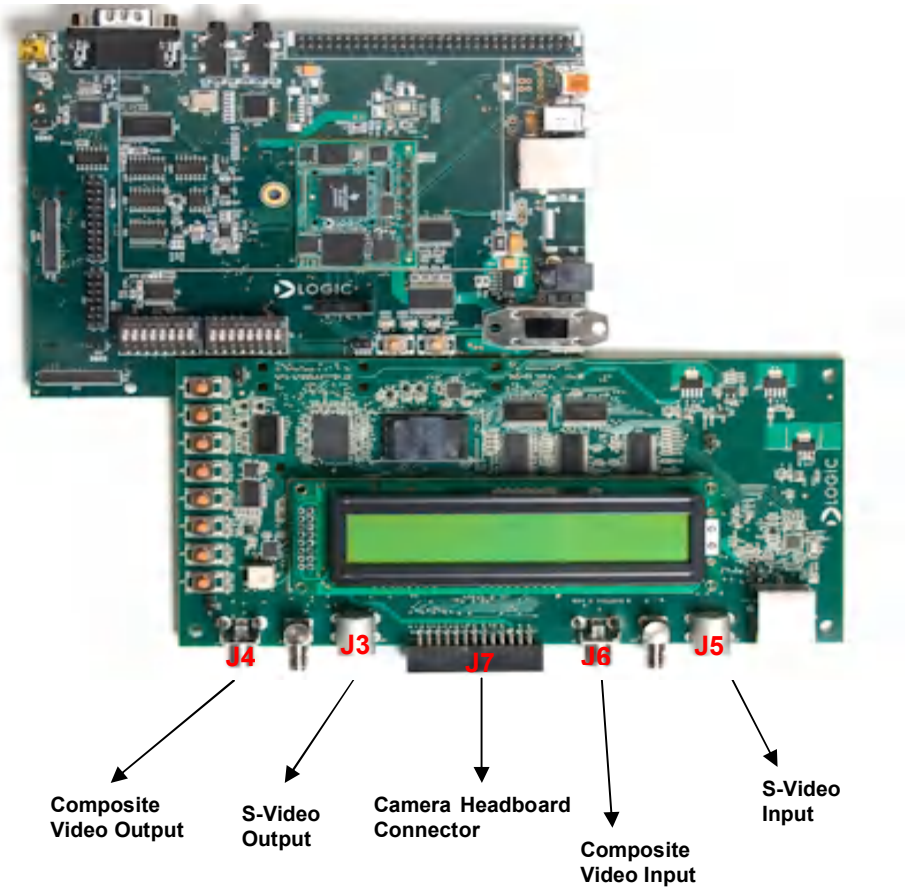


Figure 15. C6748 Video Input/Output connectors Layout

❖ **Hardware setup and connections for SD Loopback**

- Connect the UI card to C6748 EVM experimenter board (**J28** and **J29**).
- Connect RCA video cable from TVP5147 #1 input of C6748 EVM to DVD Player set in NTSC mode. Connect S-video cable from TVP5147 #0 input of C6748 EVM to DVD Player set in NTSC mode. For default application, only one input channel is sufficient.

  Connect the cables in the following sockets
  - Channel 0 – **J6** RCA jack
  - Channel 1 – **J5** S-video jack
- Connect S-video cable from ADV7343 output of C6748 EVM (**J3**) to TV. For composite output from ADV7343 connect RCA cable from **J4** to TV.
- Make sure the Video Clock is set to 27 MHz and the EVM mux are set properly for SD operation.
- Load the generated video ".out" file (**vpifSample.out**) and execute it.
- By default, demo will display video (in Composite format from J4) captured from TVP5154 #0 (in S-video from J5 jack) in NTSC D1 resolution.
- Below are the other configurable options available in this sample application
  - "**VIDEO_MODE**" – Define this to "MODE_PAL" for PAL mode of operation. Default value for this macro is "MODE_NTSC"

- "**NUM_FRAME_BUFFERS**" – The default value of "NUM_FRAME_BUFFERS" is 3 which is the recommended value. It can be increased depending upon the memory availability on the system.
- "**DISPLAY_CONNECTOR**" – The default value of "DISPLAY_CONNECTOR" is "CONN_COMPOSITE". Define this mode to "CONN_SVIDEO" for S-video cable connection. The channel 2 is programmed for composite connection.
- "**MAXLOOPCOUNT**" - This sample application will run for "MAXLOOPCOUNT" amount of frames. After which the application will close. With the current value of 500 frames, the sample application will run for 15 seconds of NTSC video or 20 seconds of PAL video. After which the loop back operation will stop.
- "**CAPTURE_CONNECTOR**" – The default value of "CAPTURE_CONNECTOR" is "CONN_SVIDEO". Define this mode to "CONN_COMPOSITE" for Composite cable connection. If S-video connection is used, vpif channel 0 is used for capture and if Composite connection is used, vpif channel 1 is used for capture.
- "**VIDEO_STORAGE**" – The default value of "VIDEO_STORAGE" is "STORAGE_FRAME". Define this mode to "STORAGE_FIELD" for field based storage. This should be same for both capture and display. If they are not same then proper handling of buffers is required as the data pointed by the capture device and the display device cannot be exchanged straightaway.

❖ **Output:**

When the sample application runs, it will demonstrate the usage of VPIF. In SD loopback the input video data from input device viz. DVD player is displayed to the output device viz. TV and the sample application performs some operations on the same.

❖ **Hardware setup and demo procedure for RAW Capture Loop back**

- Connect the UI card to C6748 EVM experimenter board (**J28** and **J29**).
- Connect MT9T001 camera headboard to **J7** camera headboard connector.
- Connect S-video cable from ADV7343 output of C6748 EVM (**J3**) to TV. For composite output from ADV7343 connect RCA cable from **J4** to TV.
- Make sure the Video Clock is set to 27 MHz and the EVM mux are set properly for SD operation.
- Load the generated video ".out" file (**vpifSample.out**) and execute it.
- By default, demo will display video (in composite format from **J4**) captured from MT9T001 image sensor (**J7**).
- The default resolution for raw capture is 480P and for display is 480I.
- Below are the other configurable options available in this sample application
  - "**DISPLAY_CONNECTOR**" – The default value of "DISPLAY_CONNECTOR" is "CONN_COMPOSITE". Define this mode to "CONN_SVIDEO" for S-video cable connection. The channel 2 is programmed for composite connection.
  - "**MAXLOOPCOUNT**" - This sample application will run for "MAXLOOPCOUNT" amount of frames. After which the application will close. The current value of 500 frames is defined.

- "**NUM_FRAME_BUFFERS**" – The default value of "NUM_FRAME_BUFFERS" is 3 which is the recommended value. It can be increased depending upon the memory availability on the system.

- *Apart from the above there are some more macros defined. They are not for sample application use case but for testing. Using them can stop sample application from working.*
  - *"SELECT_TEST_PATTERN" – When set to 1 output the test pattern on the buffer.*
  - *"SET_GLOBAL_GAIN" – When set to 1 set the global gain to the MT9T001 registers*
  - *"CONFIG_MT9T001" – When set to 1 change the resolution of MT9T001 device to SVGA.*

❖ **Output:**

For RAW loopback the input captured from the sensor is displayed on to the output device viz. TV.

❖ **Hardware setup and connections for SD Slice VBI Loopback**

- Connect the UI card to C6748 EVM experimenter board (**J28** and **J29**).
- Connect RCA video cable from TVP5147 #1 input of C6748 EVM to DVD Player set in NTSC mode.

  Connect the cables in the following sockets
  - Channel 0 – **J6** RCA jack
- For composite output from ADV7343 connect RCA cable from **J4** to TV.
- Make sure the Video Clock is set to 27 MHz and the EVM mux are set properly for SD operation.
- Put a closed caption enabled DVD in to a DVD player detecting Closed Caption. Enable the Closed caption on both DVD player and the TV.
- The Slice service used is `FVID_SLICE_VBI_SERVICES_CC_NTSC`
- Load the generated video ".out" file (**vpifSample.out**) and execute it.
- By default, demo will display video (in Composite format from J4) captured from TVP5154 #1 (in composite from J6 jack) in NTSC D1 resolution.
- Below are the other configurable options available in this sample application
  - "**VIDEO_MODE**" – Define this to "MODE_PAL" for PAL mode of operation. Default value for this macro is "MODE_NTSC"
  - "**NUM_FRAME_BUFFERS**" – The default value of "NUM_FRAME_BUFFERS" is 3 which is the recommended value. It can be increased depending upon the memory availability on the system.
  - "**MAXLOOPCOUNT**" - This sample application will run for "MAXLOOPCOUNT" amount of frames. After which the application will close. With the current value of 500 frames, the sample application will run for 15 seconds of NTSC video or 20 seconds of PAL video. After which the loop back operation will stop.
  - "**VIDEO_STORAGE**" – The default value of "VIDEO_STORAGE" is "STORAGE_FRAME". Define this mode to "STORAGE_FIELD" for field based storage. This should be same for both capture and display. If they are not same then proper handling of buffers is required as the

data pointed by the capture device and the display device cannot be exchanged straightaway.

❖ **Output:**

When the sample application runs, it will demonstrate the usage of slice VBI service. In SD slice VBI loopback the input video data and slice data from input device viz. DVD player is displayed to the output device viz. TV and the sample application performs some operations on the same.

17.13.2.2 Default Configuration Parameters

VPIF driver does not have any default configuration support. Before using the driver, application should configure the driver with valid configurations. In case the driver recognizes invalid configuration parameter it will return the corresponding error code.

All EDC drivers have default configuration. This section describes the default parameters for TVP5147 video decoder chip, ADV7343 video encoder chip and VPIF driver parameters.

❖ **Video Capture Port Default Configuration Parameters**

VPIF instance parameter used during VPIF driver registration with BIOS using TCI files. VPIF instance is configured for 128 bytes DMA transfer. Here is the default setting inside Vpif driver:

```
const Vpif_Params Vpif_PARAMS =

{

    9,                                  /* hwiNumber            */

    Vpif_DmaReqSize_128BYTE,            /* dma request size     */

    FALSE,                              /* driver power aware or not */

    Vpif_PllDomain_0                    /* PLL domain */

};
```

These parameters should be modified by application, if application wants to increase the DMA request size and changing the HWI number.

❖ **Driver naming convention used for Channel creation**

Application calls FVID_create() to create and initialize a VPIF driver channel.

The name argument is the name specified for the device when it was created in the configuration file or at run-time. The name contains five fields for display channel within it like "*/VPIF0/2/I2C0/ADV7343/0x2A*".

1. "VPIF0" - name of the VPIF instance same as UDEV name
2. "2" - channel of selected VPIF. It can be "0", "1", "2" or "3".

   In C6748 for BT capture this can be 0 or 1, for BT display this can be 2 or 3 and for raw capture this can only be 0.
3. "I2C0" – Codec Interface used to communicate with encoder and decoder.

   On C6748 this string is always same, as I2C instance 0 is connected to the encoder and decoder.
4. "ADV7343" – encoder or decoder name.

On C6748 EVM for decoder connected to S-video IN the name is "TVP5147_0", for decoder connected to Composite IN the name is "TVP5147_1", for encoder connected to Composite/S-video OUT the name is "ADV7343" and for external sensor the name is "MT9T001".

5. "0x2A" – I2C slave address.

   On C6748 EVM ADV7343 is connected to the I2C address 0x2A, the TVP5147 #0 is connected to I2C address 0x5C, the TVP5147 #1 is connected to I2C address 0x5D. For MT9T001 external image sensor, please refer to the head board schematic for the I2C address.

❖ **TVP5147 #0 Default Configuration Parameters**

TVP5147 instance 0 decoder is connected to only S-video IN. It is configured for Auto detection of standard. The internal default configuration used by TVP5147 encoder driver for instance 0 during EDC open() call is:

```
static Tvp5147_ConfParams TVP5147_default0 =

{

    Tvp5147_AnalogFormat_SVIDEO, /* only SVIDEO input is connected to the
TVP5147 instance 0*/

    Tvp5147_Std_AUTO, /* Auto standard detection is default */

    Tvp5147_OutputFormat_YCBCR422,

    Fvid_SLICE_VBI_SERVICES_NONE /* slice vbi service default : NONE   */

};
```

❖ **TVP5147 #1 Default Configuration Parameters**

TVP5147 instance 1 decoder is connected to only Composite IN. It is configured for Auto detection of standard. The internal default configuration used by TVP5147 encoder driver for instance 1 during EDC open() call is:

```
static Tvp5147_ConfParams TVP5147_default1 =

{

    Tvp5147_AnalogFormat_COMPOSITE,  /* Only Composite input is connected to
the TVP5147 instance 1 */

    Tvp5147_Std_AUTO,  /* Auto standard detection is default */

    Tvp5147_OutputFormat_YCBCR422,

    Fvid_SLICE_VBI_SERVICES_NONE /* Slice vbi service default : NONE   */

};
```

❖ **ADV7343 Default Configuration Parameters**

ADV7343 video encoder will be configured in Auto detect of standard, 8-bit YUV, S-video output mode. The internal default configuration used by ADV7343 encoder driver during EDC open() call is:

```
/** Default configuration of ADV7343 */

static Adv7343_ConfParams ADV7343_default =

{
```

```
    Adv7343_AnalogFormat_SVIDEO,   /* AnalogFormat         */
    Adv7343_Std_AUTO,              /* Mode                 */
    Adv7343_InputFormat_YCBCR422,  /* InputFormat          */
    Fvid_SLICE_VBI_SERVICES_NONE   /* Slice vbi service    */
};
```

❖ MT9T001 Default Configuration Parameters

The internal default configuration used by MT9T001 image sensor driver during EDC open() call is:

```
/* Default configuration of MT9T001 */
static MT9T001_StandardFormat stdFormat = MT9T001_MODE_480P;
```

18 UPP driver

18.1 Introduction

This section is the reference guide for the UPP 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 (or SIO), to transmit and receive (parallel) data. The following sections describe in detail, procedures to use this driver and configure it. It is recommended to go through the sample application(s) to get familiar with initializing and using the Upp driver.

18.1.1 Key Features

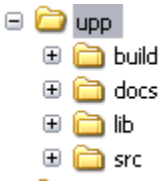
- Multi-instance support and re-entrant driver
- Each instance supports two channels (each channel can be configured as Receive or transmit).
- Support for independent control of each channel.

18.2 Installation

The Upp device driver is a part of BIOSPSP product for C6748 and would be installed as part of product installation.

18.2.1 Upp Component folder

On installation of BIOSPSP package for the C6748, the Upp driver can be found at <ID>\ ti\pspiom\upp\



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

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

18.2.2 Build Options

The Upp library can be built using the CCS v3.3 project file located at <ID>\packages\ti\pspiom\upp\build\C6748\ccs3\upp.pjt. This project file supports the following build configurations.

It can also be built using the CCS v4 project files located at <ID>\packages\ti\pspiom\upp\build\C6748\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_C6748" to build library for C6748 soc.
- Defines "-DUpp_STATISTICS_ENABLE" to enable support for collecting statistics in the driver.

iDebug:

- "-g –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6748" to build library for C6748 soc.
- Defines "-DUpp_STATISTICS_ENABLE" to enable support for collecting statistics in the driver.

Release:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6748" to build library for C6748 soc.
- Defines "-DUpp_STATISTICS_ENABLE" to enable support for collecting statistics in the driver.

iRelease:

- "-o2 –mo –mv6740" compile options used to build library.
- Defines "–DCHIP_C6748" to build library for C6748 soc.
- Defines "-DUpp_STATISTICS_ENABLE" to enable support for collecting statistics in the driver.
- Defines -d"PSP_DISABLE_INPUT_PARAMETER_CHECK" -d"NDEBUG" to eliminate parameter checking code and asserts in driver

18.2.2.1 Required and Optional Pre-defined symbols

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

"-DCHIP_C6748" is used above to build for C6748. Internally this define is used to select a soc specific header file (soc_C6748.h). This header file contains information such as base addresses of upp devices, their event numbers, etc.

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

Use "Upp_STATISTICS_ENABLE" when building library to enable statistics collection in the driver like number of DMA programming errors, internal bus errors etc.

Use "PSP_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 profile by default in the CCS 3.3 and CCS 4 pjts provided.

18.3 Features

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

18.3.1 Multi-Instance

The Upp driver can operate on all the instances of Upp on the EVM C6748. Different instances may be specified during driver creation time, and instance 0 with corresponding device ID 0 is supported.

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

1. Static creation – static creation is done in the "tcf" file of the application; the allocation of device 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 Upp instance
2. Dynamic creation – Dynamic creation of an Upp 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. Upp requires this field to be Upp_IOMFXNS.
- initFxn: Upp requires that the user call Upp_init() as part of this initFxn. Users can also directly hook in Upp_init().
- device parameters: Upp requires the user to pass an `Upp_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 Upp 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 Upp, please refer to the Upp sample application (included with this driver release), and the DSP/BIOS API Reference (spru403o.pdf, included in your DSP/BIOS installation).

18.3.2 Channel control

Each instance of the Upp driver can be used for creating channels for the physical channels A and B available. The channels can be configured to be either both transmit or both receive or either one of the channel configured for transmit with the other configured for receiving data (duplex mode). This could be achieved by opening a stream Channel as an INPUT channel when receiving data and opening a stream Channel as an OUTPUT channel for transmitting data. The type of Channel is specified while creating the channel (using `GIO_create or SIO_create () specify "IOM_OUTPUT" or "IOM_INPUT"`). The configuration parameters are explained in the sections to follow.

18.3.3 Supported IO rates

The Upp peripheral imposes certain constraints on the IO clocks that can configured for the device.

- Neither the module clock nor the transmit clock can be faster than one-half the device CPU clock speed.
- Single data rate restricts the maximum speed of the I/O clock to one-fourth the device CPU clock speed.
- Double data rate restricts the maximum speed of the I/O clock to one-eighth the device CPU clock speed.

Hence depending on the input transmit clock to the Upp, the output IO clock will be restricted to the values mentioned above.

18.4 Configurations

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

18.4.1 Upp_Params

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

During the creation of the device instance these parameters are passed by the application. If the application wants to use default parameters for the driver, the application can use "**Upp_PARAMS**" for the same.

Members	Description
emulationMode	Emulation mode setting to be used.
dlbMode	Loop back mode selection
devMode	Device operating mode (1/2 channels,xmt,rcv etc)
pscPwrmEnable	Power management support is to be enabled or not.
inputFreq	Input clock frequency
hwiNumber	Hwi number where the upp event is configured
instHandlerSel	Interrupt handler to use ECM module or directly use HWI
pllDomain	PLL domain where the Upp source clock is configured to.

Note: inputFreq is the clock input to the module for the generation of the Transmit clock. Once the application has decided on the clock input to be used, the pllDomain also should be specified properly (especially if the power management is to be supported).

18.4.2 Upp_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
enableCache	Option to enable the cache operations on the user supplied buffers.
chanSel	Channel to be configured (A or B).
bitWidth	The wordwdith to be configured for the channel
dataRate	Data rate to be used by the channel(single or double)
chanMode	option to specify special modes like (interleaved etc)
dataPackFmt	Data packing format to be used(left justified, right justified etc)
opFreq	Output bit clock frequency.
idleVal	Idle pattern to be transmitted when no data is available with the Upp
userCbFxnStatus	User specified callback function to be called in case of status interrupts.
userCbArgStatus	User specified argument to be passed to the status callback function.
userCbFxnError	User specified callback function to be called in case of error interrupts.
userCbArgError	User specified argument to be passed to the error callback function.
fifoThreshold	FIFO threshold for this channel
signalCfg	Configuration of the START, WAIT and ENABLE signals.

Note: If the application intends to use more than one channel (two channel mode) then, it should create both the channels before submitting an IO Request on any of the channels. This limitation is because of the fact the Upp device needs to be configured only when the peripheral is disabled. Hence the Upp device remains disabled until both the channels are created and then only it will be enabled. IN single channel mode the Upp is enabled as soon as the first channel is created.

18.4.3 Operational Mode

The Upp driver only supports interrupt mode of operation and utilizes internal DMA for the transfer of data. Hence no support exists for the other modes of operation like polled and DMA.

18.5 DMA support

The Upp device has internal DMA available to support data transfer and reception to the Upp. There are two DMA channels available I and Q. when operating in Non-interleaved mode the DMA channel I services Upp channel A and DMA channel Q services Upp channel B. But if the device is operating in interleaved mode of operation then both the DMA channels I and Q service the Upp channel B.Each DMA channel allows up to two IO requests to be programmed.

18.6 Control Commands

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

Command	Arguments	Description
Upp_Ioctl_START	*None*	Starts the Upp device and makes it ready for transactions.
Upp_Ioctl_STOP	*None*	Stops the Upp device.
Upp_Ioctl_QUERY_STATS	*Upp_devStats **	Retrieves the device statistics from the driver.
Upp_Ioctl_CLEAR_STATS	*None*	Clears all the driver internal statistics.
Upp_Ioctl_SET_TIMEOUT	*Uint32 **	Configures the genric timeout used by the Upp driver to a user supplied value.
Upp_Ioctl_SUSPEND	*None*	Command to suspend the UPP IO transactions
Upp_Ioctl_RESUME	*None*	Command to resume the previously suspended uPP transactions
Upp_Ioctl_CHAN_CONFIG	*None*	Command to change the configuration of the uPP channels

18.7 Use of Upp 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.

18.7.1 GIO_create

Parameter Number	Parameter	Specifics to Upp
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 Upp requires to received data and "IOM_OUTPUT" when Upp requires to transmit
3	Status	Address to place return status from Upp.
4	Channel Params	Pointer to chanParams structure for Upp channel.
5	GIO_Attrs *	Parameters required for the creation of the GIO instance (e.g. channel parameters)

18.7.2 GIO_control

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

18.7.3 GIO_write/read

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

18.8 Sources that need re-targeting

18.8.1 ti/pspiom/cslr/soc_C6748.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.

18.9 EDMA3 Dependency

Upp driver relies on internal DMA to move data from/to application buffers to peripheral.Hence, the Upp peripheral has **NO** dependency on the EDMA3 driver.

18.9.1 Used Paramset of EDMA 3

BIOSPSP Upp driver uses internal DMA. Hence, it will **NOT** be using any EDMA3 paramsets.

18.10 Known Issues

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

18.11 Limitations

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

18.12 Upp Sample applications

18.12.1 EVM to EVM communication sample

18.12.1.1 Description:

This sample demonstrates the use of the Upp driver in EVM to EVM communication mode.

The Upp driver is configured statically in uppSample.tci file. The initFxn and uppParams used in UDEV.create are globals declared in uppSample_main.c.

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

bios.ECM.ENABLE = 1;

bios.HWI.instance("HWI_INT9").interruptSelectNumber = 1;

These lines configure the ECM module and map uPP events to CPU interrupts. For example the uPP event number is 94 which fall in ECM group 3. Here ECM group 3 is mapped to HWI_INT9.

The main () function configures the PINMUX.The uppDemoTask () task exercises the uPP driver.It uses SIO APIS to create uPP channels and read and write to them.

The uppUserInit () calls Upp_init () and initializes the Upp_Params structure.

18.12.1.2 Build:

This sample can be built using

<ID>/packages/ti/pspiom/examples/evm6748/uppEvm/evmA/build/ccs3/uppSampleEvmA.pjt and also using

<ID>/packages/ti/pspiom/examples/evm6748/uppEvm/evmB/build/ccs3/uppSampleEvmB.pjt

18.12.1.3 Setup:

This example needs two EVMs to be connected. The connection settings are as given below. The pin connections are available on J29.Below table shows the configurations and the pin connection to be used for this example to be executed.

EVM1(duplex 0)		EVM2(duplex 1)	
B (Xmt)		B(Rcv)	
PIN NAME	PIN NUMBER	PIN NAME	PIN NUMBER
CH0_START	56	CH0_START	56
CH0_CLK	58	CH0_CLK	58
CH0_ENABLE	44	CH0_ENABLE	44
CH0_WAIT	57	CH0_WAIT	57
D15	66	D15	66
D14	67	D14	67
D13	64	D13	64
D12	65	D12	65
D11	62	D11	62
D10	63	D10	63
D9	60	D9	60
D8	61	D8	61
A(Rcv)		A(Xmt)	
PIN NAME	PIN NUMBER	PIN NAME	PIN NUMBER
CH1_START	45	CH1_START	45
CH1_CLK	47	CH1_CLK	47
CH1_ENABLE	49	CH1_ENABLE	49
CH1_WAIT	51	CH1_WAIT	51
D7	74	D7	74
D6	75	D6	75
D5	72	D5	72
D4	73	D4	73
D3	70	D3	70
D2	71	D2	71
D1	68	D1	68
D0	69	D0	69

18.12.1.4 Output:

When the sample runs, it will output the following messages at the DSP-BIOS message window.

0 Upp Sample Application

1 Requests submitted to the uPP driver.

2 Requests submitted to the uPP driver.

3 *Sample Application completed...*

18.12.2 Loopback sample application

18.12.2.1 Description:

This sample demonstrates the use of the Upp driver in an internal loopback mode.

The uPP driver is configured statically in uppSample.tci file. This file can be directly imported in to an application's tcf script. The initFxn and Upp_Params used in DEV.create are globals declared in uppSample_io.c.

The spiSample.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_INT9").interruptSelectNumber = 1;

These lines configure the ECM module and map uPP events to CPU interrupts. For example the uPP 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 to enable the Upp peripheral.

The uppDemoTask () task exercises the uPP driver. It uses SIO APIS to create uPP channels and reads and writes to them.

The uppUserInit () calls Upp_init () and initializes the Upp_Params structure.

18.12.2.2 Build:

This sample can be built using

<ID>/packages/ti/pspiom/examples/evmOMAPL138/upp/build/ccs3/uppSample.pjt

Alternatively it can be built using ccs4 project files available at

<ID>/packages/ti/pspiom/examples/evmOMAPL138/upp/build/ccs4/

18.12.2.3 Setup:

Connect an EVM OMAPL138 platform to a host PC using a JTAG.load the uppSample.out file and execute the same.

18.12.2.4 Output:

When the sample runs, it will output the following messages at the DSP-BIOS message window.

0 Upp Sample Application

1 Requests submitted to the uPP driver.

2 Data compare passed.

3 Sample Application completed...