DSP/BIOS PSP DM648/C6452

User's Manual

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.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		Applications	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

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

Preface

Read This First

About This Manual

This User's Manual serves as a software programmer's handbook for working with the **DSP/BIOS PSP DM648/C6452 Version 1.10.03.** This manual provides necessary information regarding how to effectively install, build and use **DSP/BIOS PSP DM648/C6452** in user systems and applications.

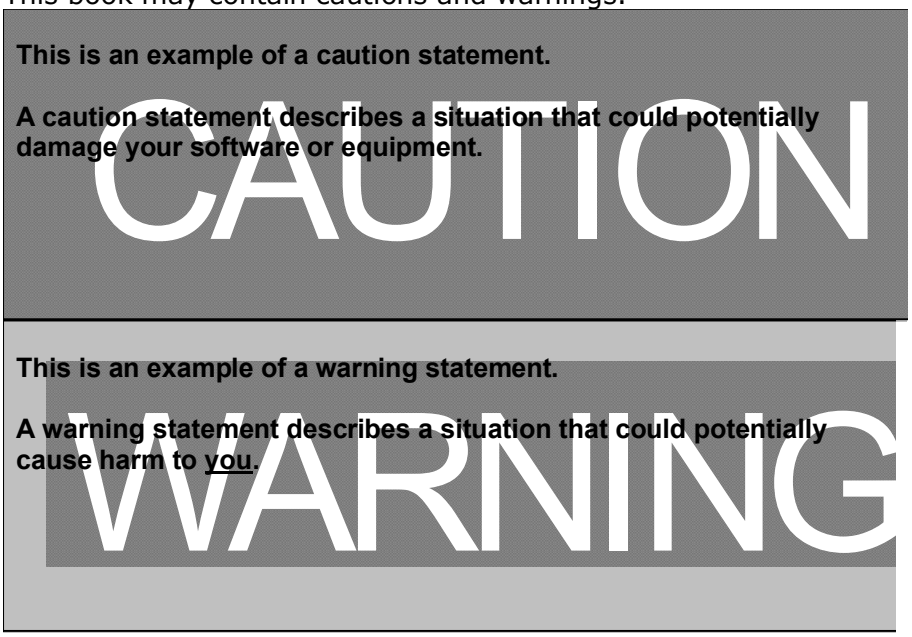
Abbreviations

Table 1-1. Table of Abbreviations

Abbreviation	Description
PSP	Platform Support Package
DSP/BIOS PSP DM648/C6452	This is TI coined name for the product.
API	Application Programming Interface
IOM	Device Driver Adapter
DDC	Device driver core
LLC	Low Level Controller
OS	Operating System
PAL OS	Platform abstraction layer for operation system
SOC	System On Chip

Information About Cautions and Warnings

This book may contain cautions and warnings.



The information in a caution or a warning is provided for your protection. Please read each caution and warning carefully.

Related Documentation

Internal

This is a list of documents that are TI Proprietary and Strictly Private. Exposure to audience outside TI will need due considerations and approvals from TI Legal authorities.

- PSP Framework Architecture (PSPF) 1.0 internal web link: (http://www.india.ti.com/~anant/PSPF1.0)
- DM648 TRM and peripheral documents from PDS

Revision History

Version	Date	Revision History
1.0	19th June, 2006	Pre-silicon release 1.0
1.1	16th August, 2006	Pre-silicon release 1.2
1.2	29th August, 2006	Incremental release Pre-silicon release 0.1.0

1.3	18th September,2006	Incremental release Pre-silicon release 0.1.4
1.4	1st October,2006	Pre-silicon release 0.1.6
1.5	5th October,2006	Pre-silicon release 0.2.0
1.6	1st December, 2006	Post-silicon release 0.3.0
1.7	2nd February, 2007	Changed for DM648
1.8	7th June, 2007	Updated the document for 0.6 releases with changes for i2c, spi, and uart and tsip application usage.
1.9	18th June, 2007	Updated the document for 1st build release.
1.10	13th July 2007	Added SoC analyzer information
1.11	16th July 2007	Updated Gel file information
1.12	28th July 2007	Updated information for Soc
1.13	30th August 2007	Updated as instrumentation build procedure and other minor typo errors.
1.14	19th September 2007	Updated dvsdk environmental variables.
1.15	24th October 2007	Changed the build version to release version (xx.yy.zz)
1.16	May 29, 2008	Updated for release 1.10.01
1.17	22 Dec 2008	Updated for release 1.10.02
1.18	16 April 2009	Updated for release 1.10.03

Contents

Table of tables

Table of figures

Chapter 1

Introduction

This chapter introduces the **DSP/BIOS PSP DM648/C6452** to the user by providing a brief overview of the purpose and construction of the **DSP/BIOS PSP DM648/C6452** along with hardware and software environment specifics in the context of **DSP/BIOS PSP DM648/C6452** deployment.

1.1 Overview

The DM648/C6452 PSP is aimed at providing fundamental software abstractions to DM648/C6452 EVM resources and plugs the same into BIOS operating systems so as to enable and ease application development by providing suitably abstracted interfaces.

1.1.1 Supported Services and features

This release of **DSP/BIOS PSP DM648/C6452** provides the following:

- DM648 PSP
  - UART
  - I2C
  - McASP (with AIC33 support)
  - SPI
  - TSIP (tested in DLB mode)
  - Video port (with Encoder and decoder support) (tested for SD capture and display 8 In + 1 out)
  - VLYNQ
  - PCI
- C6452 PSP
  - UART
  - I2C
  - SPI
  - TSIP (tested in DLB mode)
  - VLYNQ
  - PCI
- Other separate components
  - EDMA driver
  - Ethernet switch driver

1.1.2 System Requirements

The following products are required to be installed for using the DM648 DSP/BIOS PSP:

- CCS 3.3.38.2
- DM648/C6452 EVM (currently tested with DM648 EVM)
- DSP-BIOS 5.31.08 or higher
- CG Tools 6.0.8
- XDC tool 3.00.01

1.2 Driver Software Architecture

This section gives detail on the overall architecture of TI device driver.

1.2.1 Functional Decomposition

The device driver is partitioned into distinct sub-components, consistent with the roles and responsibilities already discussed in section 1.3. In the following sub-sections, each of these functional sub-components of the device driver is further elaborated.

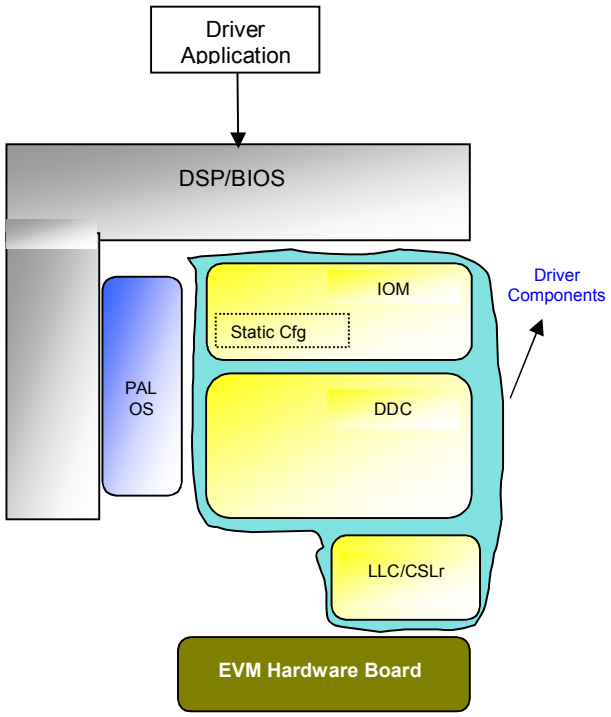


Figure 1 TI Device Driver Functional decomposition

The central portion (IOM-DDC-LLC) shown constitutes the mainline device driver component. The surrounding module PAL-OS constitute the supporting system components that facilitates the interfaces between the OS and the above mentioned device driver components. These modules do not specifically deal with device driver but assist the driver by providing OS abstraction.

1.2.2 H/W Device Specific Layer

CSL Register Layer: This is comprised of a bunch of symbolic constants (#defines) that expose the register bit-field details of the h/w along with assorted other constants such as bit-field masks, shift values and default settings.

LLC Layer: The LLC (low level controller) layer forms the lower most, h/w specific under-pinning of the TI device driver.

This is comprised of a set of functions that exposes functions to set various functionalities supported by hardware. Hardware parameters for example in UART baud-rates, stop bits etc. can be set. This parameter setting is done by writing proper values to the respective hardware register.

A C-structure data type definition is provided as an as-is map of the peripheral device registers in the processor's memory map. This structure is termed the Register Overlay structure. The intended mode of usage is to initialize a pointer variable to base address of the peripheral h/w device and typecast it using structure overlay definition. This way, an otherwise un-adorned pointer assumes a strong C-type thereby making it possible to program the h/w registers by read/write to structure member elements.

It is important to note here that LLC layer *scope is limited to directed access of the underlying h/w device*. It does not depend on any specific OS and does not exploit optimizations specific to a given Compiler. In contrast to a device driver, it does not perform operations that are viewed as management of data movement over the peripheral device. It does not model state machines or protocols. Besides, LLC layer is deployed as a per-processor (single CPU) specific library of services. LLC layer philosophy mandates that services exported for one module (read peripheral h/w device) must not call the services of another module. This orthogonal rule is enforced to allow for true componentization of device drivers.

1.2.3 Device Driver Core functionality (DDC)

The DDC module for device driver, the OS abstracted portion of the driver which provides basic behavior of the driver, modeling the main functionalities and Protocols. The DDC does not directly touch the underlying h/w, it does so via the LLC layer. Likewise,

it does not make direct reference to any OS services, it glues to the OS via a well specified adaptation module termed the IOM.

By mandating certain basic interfaces and extension principles from all implemented device drivers, the DDC helps achieve, uniformity of driver API syntax/semantics across all supported devices and OS platforms.

The objective of a good driver partitionning is to sediment as much of the driver behaviour into the DDC as is practical. This calls for a IOM that is thin and efficient. Reuse and performance improvement efforts can then focus on DDC where bulk of the functionality is realized.

The DDC is modeled after object oriented style of modular software development. It specifies a base set of interfaces that standardizes the common aspects of all device drivers such as – configuration, creation, initialization & startup, termination and teardown. This base set of function can be extended by introducing operations specific to a particular device – IO Access and IO Control. DDC further encourages formation of device classes while extending the basic functionality. This way, device drivers for h/w components that are similar in data transaction models and control semantics will look alike.

The DDC on its own has no existance, it is brought to life by the IOM when the device driver is loaded by the system and removed from the system upon unloading. The IOM has the obligation to pass-on the relevant driver configuration parameters to the DDC during creation phase.

The IOM implements a set of functions that adapt the driver to OS. Likewise DDC implements a set of functions that constitute the driver functional interface.

To improve componentization of device driver, DDC comprises of C functions which makes use of the LLC Layer for implementation of certain functions that support:

- Operations to formally begin/end access to device h/w
- Operations to perform onetime setup of the H/W device such as during device driver initialization
- Operations to program the H/W device registers to change one or more of its configuration parameters
- Operations to query and infer the current state of the H/W device

1.2.4 OS Specific Device Driver Adaptation (IOM)

As discussed above, the DDC is not complete unless it is supplemented by the IOM. The IOM “adapts” the driver core to the specific OS. IOM implements aspects such as – threading model for transaction processing,

Interrupts registration and de-registration, handshaking with OS prescribed upstream/downstream data queues and threads etc., The IOM has full visibility to underlying OS services and is custom-built for a given OS.

While the IOM is primarily intended for presenting an OS manifest to the underlying DDC, it is also possible that the IOM upper-edge interface (user level) imbibes the semantics of any pre-specified Framework, if one exists. This is necessary to prevent undue overheads in system integration.

1.2.5 Platform Abstraction Layer for OS services (PALOS)

It should be clear by now that a device driver is composed of three main sub-components – the HW specific bottom-edge, the OS specific upper-edge and the central device driver core that makes no reference to any particular OS services. Memory footprint Scalability is a key consideration in deploying such a driver. In this section, we give a quick overview of suggested approach followed in TI device driver implementations.

As discussed earlier, both the IOM and DDC publish a table of function pointers for each other to use. However, it must be noticed that, not all the OS specific adaptation services are solicited by DDC in the same context. In addition, it's not practical to abstract all required OS services due to performance reasons. Therefore, we'd usually end up with a residue part of driver that must be *custom built in the IOM* by *directly availing the OS services*. It is this part of the IOM that is exposed through the table of functions to the DDC.

The rest of the OS services such as working with Semaphores, Mutexes, Memory buffer pools etc., are generic in nature. If these services are implemented as static functions, rolled into each IOM, memory foot print bloat occurs when there are multiple instances of device drivers in the system. To mitigate memory footprint bloat and difficulties in reuse, all generic OS abstraction services are pulled out as separate compilation units so that only one copy of these need be loaded to resolve all references across different driver instances.

This common module is termed the PAL OS and is depicted in figure below as green colored units, located inside the library to the right.

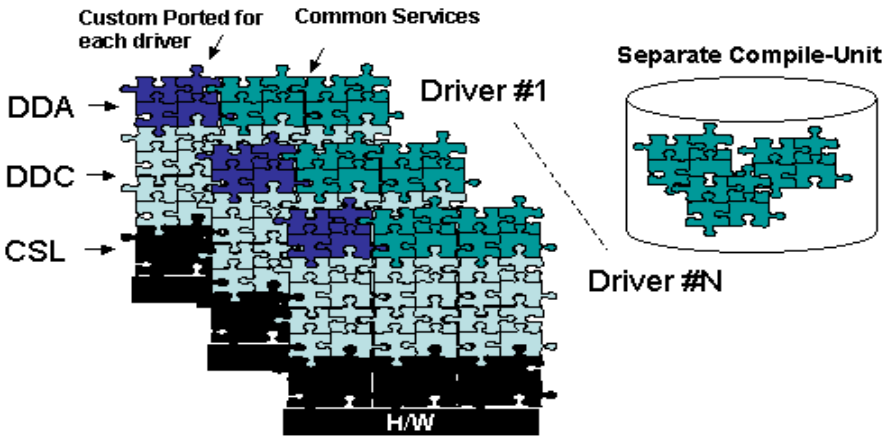


Figure 2 Genesis of PAL-OS

1.3 Design Philosophy of drivers

Central to the philosophy of TI device driver architecture is clarity in separation of roles and responsibilities for the various parts of the device driver. Rather than treat the entire device driver as a monolithic block of code, effort is made to identify the portions of the device driver that are involved in:

- Coupling or handshaking with the specific OS
- Performing primitive, directed read/write access to the h/w device
- Modeling the crux of the driver behavior – protocol, state machine etc this in itself is regardless of any given OS.

With this view, the device driver functionality can be enacted by three key roles defined here under:

- OS Specific Device Driver Adaptation (IOM)
- Device Driver Core, isolated from OS as well as H/W (DDC)
- LLC abstraction providing services to perform primitive access necessary to control/configure/examine status, of the underlying h/w device.

Since there exists a clear separation of roles and responsibilities of the three sub-components of the driver, the prescribed architecture helps in creation of robust device drivers through tested/reusable pieces. Besides, it ensures in maintaining uniform semantics for similar services, supported across different drivers, for different platforms.

The figure below further elucidates the driving philosophy in partitioning the device driver into distinct functional sub-components – IOM, DDC and LLC.

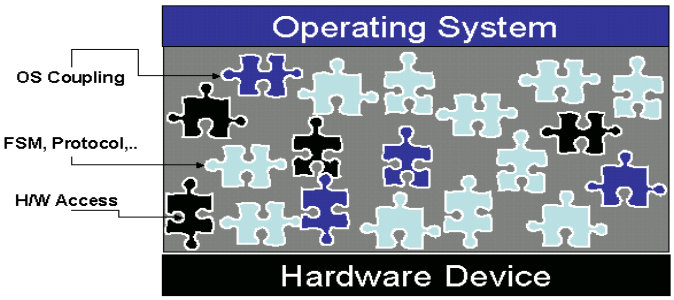


Figure 3 Monolithic Device Driver – Amorphous structure

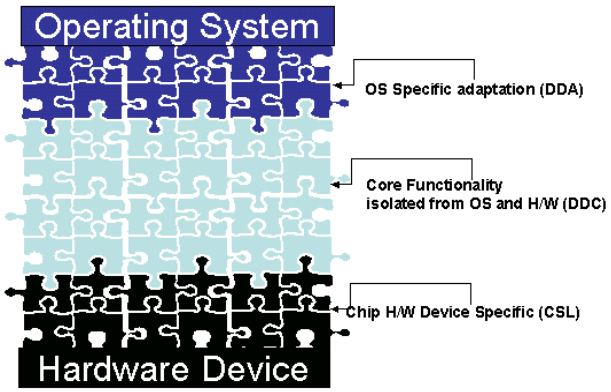


Figure 4 Layered Device Driver – Well defined structure

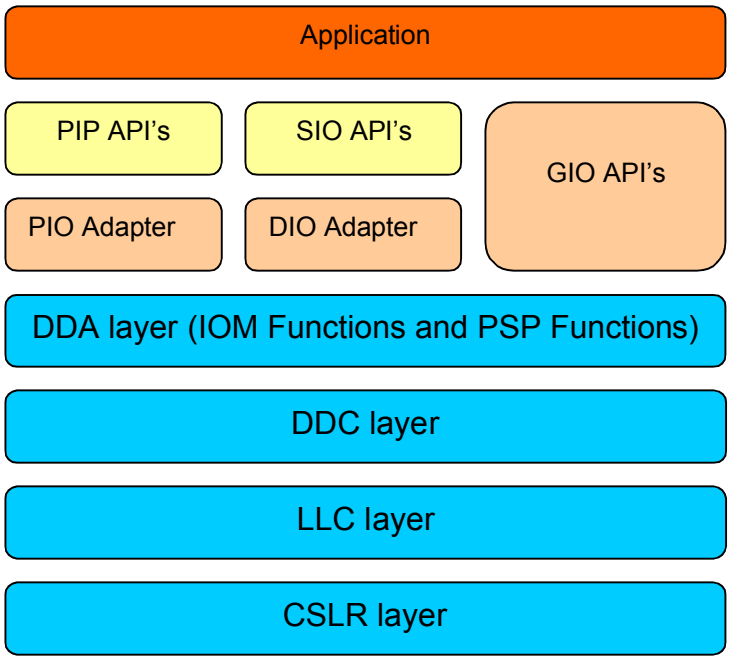


Figure 5 Device driver Architure

1.3.1 Design Goals

The following are the key device driver design goals being factored by proposed TI architecture:

- Uniformly styled drivers, promoting increased reuse and code familiarity to developers
- Minimal overheads in integrating TI device driver into any Software System
- Device driver must support Synchronous as well as Asynchronous interfaces to the user where appropriate
- Device driver must operate both with and without Interrupts capability as appropriate
- Device driver must leverage the H/W DMA capability where available to improve performance of device driver in case of block-oriented transfers.

1.3.2 Assumptions

- It is assumed that TI Device Drivers are required to be fully functional in their native OS (DSP/BIOS in case of DSP side devices). Integration of TI Device Drivers into a given software system is beyond the immediate scope of the proposed architecture. However, effort has been made by TI to ease the integration via IOM interfaces discussed later in this document.

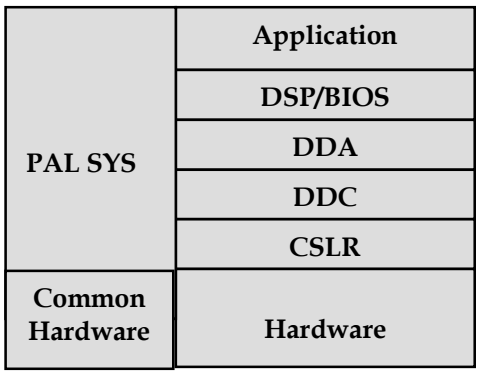
1.3.3 Design Principles

The guiding principles for the TI Device Driver design, drawn in the context of a fore mentioned philosophy, goals and constraints can be enumerated as follows:

- Clear separation of H/W-dependent and H/W-independent parts.
- Clear separation of OS dependent and independent parts
- Consistent interfaces across same class of devices, ensured by design
- Modular design to effectively address multiple device instances – avoids needless code size penalty and also to isolate device specific driver Implementation details from usage policies.

1.4 Design philosophy for PAL SYS drivers

PALSYS driver component abstracts common hardware system functionalities and provides an interface to the application as well as other PSP driver components such as DDA and DDC.



PALSYS can abstract the following functionalities.

- EVM specific Hardware
- Common peripherals
  - LPSC, GPIO, RTC and TIMER utility APIs.
- Bus configuration drivers which normally would not entertain I/O calls
  - VLYNQ
  - PCI

Chapter 2

Installation Guide

This chapter discusses the **DSP/BIOS PSP DM648/C6452** installation, how and what software and hardware components to be availed in order to complete a successful installation (and un-installation) of **DSP/BIOS PSP DM648/C6452**.

2.1 Installation and Usage Procedure

1. Install the above products as per instructions provided along with the products.
2. Install the PSP package by deflating the compressed package file in any drive
3. Configure UART to 115200 BAUD Rate, with 8 bit Data, No Parity, No Flow Control & Stop Bit equal to 1 on PC side.
4. As the EVM has an EVM Controller Processor (ECP), it is required that it to be setup for proper mux selection and reference clock selection. As some drivers (like video drivers) will not come out of reset, it is necessary to configure this muxes properly before we start using PSP drivers.
    a) This can be done by the GEL file available with EVMs (the current version provided by EVM manufacturers setup both audio and video muxes). If any custom change in mux is needed, GEL file can be appropriately modified to program muxes and reference clocks. In non-ccs applications (where the image is loaded from bootloader) the bootloader may use the mux configuration code from GEL file to perform these mux and reference clock configuration.
    b) To help starting up using this package, ccs setup files are available in pspdrivers\docs\Other\ folder and this is not a featured deliverable of PSP package. Appropriate ccs setup files should be availed from respective providers. To download the EVM DM648 board support files/drivers, log on Lyrtech FTP site
        - Open Windows Explorer. (not Internet Explorer)
        - In the Address bar, type ‘’ftp://dm648evm:kC9YxeE9m7c5B1JzVwmu@ftp2.lyrtech.com ‘’
5. For video port sample application/setup, please refer VIDEO port release notes.
6. TCF files in PSP package use HWI 7, 8, 9 and 10 for ECM groups 0,1,2,3. Please perform necessary changes in your customized package.
7. For information on Ethernet package, please refer readme html file in the docs directory of the package. The Ethernet package (tar file) should be copied in packages directory of NDK installation and from there it has to be un-tarred. The tar itself has TI folder in it and untarring would overwrite the TI directory with needed driver files from driver package. Please note that copying the Ethernet driver tar file anywhere else and tarring would create TI folder and copy the contents in that.
8. Please note that if BIOS version 5.31.07 by default uses 891 MHz as clock has to be used with timer patch.

Note:

1. The XDCPATH environmental variable should be set properly to recompile the drivers/sample applications. Ensure that the XDCPATH variable should contain, BIOS packages path, XDC packages path, PSP drivers packages path, EDMA packages path.

Example:

```
z:\pspdrivers\pspdrivers_\packages;
C:\CCStudio_v3.3\bios_5_33_01\packages;
C:\Program Files\Texas Instruments\xdctools_3_10_02\packages;
C:\Program Files\Texas Instruments\edma3 lld 01 05 00\packages;
```

2. psp_xdcpaths_common.dat should contain the information about the PSP package installation directory and the EDMA installation directory.

    a. The *"pspRootDir"* string should point to the directory where the PSP package is installed. For example if the PSP packge is installed as *"z:\pspdrivers\pspdrivers_1_10_01\packages"* then the *pspRootDir* should be equal to *"z:/pspdrivers/".* Please note the forward slashes in the path.

    b. The *"pspversion"* string should contain the version number of the current PSP installation. For example, if the version of PSP package installed is 1.10.01, then *pspversion* should be equal to "1_10_01".

    c. *edma3InstallDir* should be set to point to the installation directory of the EDMA package. If EDMA driver package is installed at *"c:\ProgramFiles\TexasInstruments\",* then *edma3InstallDir* should be set to *"c:/ProgramFiles/TexasInstruments/".* Please note that forward slashes should be used here for directory paths.

    d. The first section –Section 1- should be uncommented in case of non DVSDK users and the second section –Section 2- should be commented. If users are using PSP package from inside a DVSDK installation, then the second section – Section 2- should be uncommented and first section – Section 1- should be commented.

Steps to run sample applications of EDMA, UART, I2C, SPI and VLYNQ on DM648/C6452 EVM:

1. For hardware setup, connect the UART cable to DM648 EVM. Connect XDS 510 USB emulator to the JTAG connector on board. Switch on the power supply for board.

Note1: To run VLYNQ sample, hardware setup will consist of two back-to-back connected daughter cards plugged into DM648 EVM. (If the EVM we are connecting is other than DM648 EVM, then corresponding VLYNQ connector along with VLYNQ cable to be used to make the setup)

Note2: To run McASP sample, connections to Audio input device (typically PC headphones out) and audio output device (typically, headphones or connection to amplified speakers) is a prerequisite.

2. Open CCS 3.3.38.4 setup. Select EVMDM648_XDS510USB processor.

3. Connect to DM648 processor.
4. Open \pspdrivers\drivers\<device>\sample\build\<device dependent>.pjt Build the image and download it to the platform using CCS.
5. Sample applications for the included driver shall automatically start following this logo.

Steps for running sample application of PCI driver on DM648/C6452 board

1. Set the switch settings in the EVM, for PCI boot mode. Please refer the EVM document for complete details of switch settings

   (To download the DVEVM DM648 board support files/drivers, log on Lyrtech FTP site

   i. Open Windows Explorer. (not Internet Explorer)

   ii. In the Address bar, type ‘’ftp://dm648evm:kC9YxeE9m7c5B1JzVwmu@ftp2.lyrtech.com ‘’)
2. Fit EVM DM648/C6452 card into the PCI slot of a Linux machine having linuxkernel 2.6.9(RHEL 4 AS).
3. Connect USB emulator to JTAG port of DM648 EVM. Connect the USB cable to other PC having CCS on that.
4. Power ON the Linux machine. BIOS of Linux machine will start booting up.
5. Open CCS 3.3 on the PC and try to connect to DM648 EVM.
6. Press Ctrl+Alt+F1 to go to console in Linux machine.
7. Enter username and password for "root" user.
8. Copy Makefile and pcidrv.c from "pspdrivers\pal_sys\pci\docs\linuxapp" folder to "/home" directory of Linux machine.
9. Go to "/home" directory by giving "cd /home" command.
10. Give "make" command. This will generate the object file of a PCI host driver for Linux. Driver will be generated by module name "pcidriver.ko" in same directory.
11. Give "insmod pcidriver.ko" command. This will insert the driver as a module.
12. This will give some print messages on Linux machine. If print messages of Linux driver for PCI are not visible on the console then give "dmesg" command on Linux console to see the print messages.
13. There will be a print message saying "Address to write from DSP = 0x518000". Note this address (0x518000).
14. Build CCS project and run the out file.
15. When asked for input (by CCS scanf window) enter the address value got from step 12.

16. Out file will generate a host interrupt on Linux machine. On generation of host interrupt the will be print messages saying "Interrupt received from DSP". This indicates that DSP to Host interrupt is generated.
17. Sample application loaded on EVM DM648/C6452 through CCS will also perform read/write test and print the message of success or failure on STDOUT

2.2 Un-Installation

1. Delete the <Installation Path>:\pspdrivers_<Version> directory to remove DM648/C6452 PSP package.
2. Un-install the products (listed in system requirements) as per instructions provided with the product.
3. pcidriver.ko can be un-installed from Linux machine by giving "rmmod pcidriver.ko" command on console of Linux machine. (applicable of PCI setup only)

2.3 PSP Component Folder

This section details the files and directory structure of the installed **DSP/BIOS PSP DM648/C6452** in the system. A viewgraph of the actual directory tree (as seen in the final deployed environment is inserted here for clarity.

2.3.1 Top level PSP Directory structure:

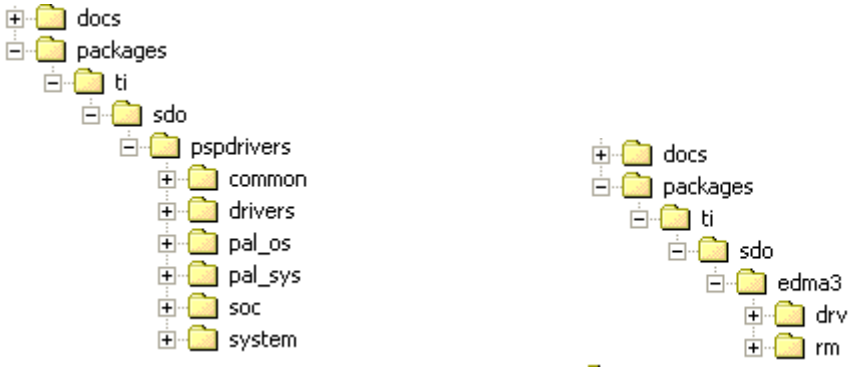


Figure 6 PSP Top level directory structure

The directories of interest (for this release) are:

\edma3\drv – EDMA3 driver
\edma3\rm –EDMA3 resource manager
\docs – Documents
\pspdrivers\drivers - Drivers
\pspdrivers\common – Include files used by all the drivers
\pspdrivers\pal_os - PAL OS component
\pspdrivers\pal_sys – PAL SYS component
\pspdrivers\system – EVM dependent drivers (e.g. audio) and sample Application
\pspdrivers\soc – CSLRs for DM648 SOC

2.3.2 PAL OS for BIOS Directory structure:

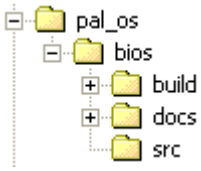


Figure 7 PAL OS directory structure

..\pal_os\bios\build - PAL OS build directory
..\pal_os\bios\src - PAL OS BIOS source directory
..\pal_os\bios\docs – documents for PAL OS
..\pal_os\bios\ - Include files

2.4 CCS Projects

The following CCS projects are provided as part of the PSP package:

DM648 Sample project for BIOS Sample Application (Executable build):

- \system\DM648\bios\evmdm648\build\evmDM648_dvr_pimux_combo1_bios_sample_out.pjt

  This project builds the sample code for UART, I2C, McASP and VPORT.

- \system\DM648\bios\evmdm648\build\evmDM648_dvr_pimux_combo2_bios_sample_out.pjt

  This project builds the sample code for SPI and VLYNQ.

- \pspdrivers\pal_sys\pci\sample\psp_pci_bios_sample_dm648.pjt

  This project builds the sample code for PCI

- \pspdrivers\drivers\tsip\sample\c6452_evm_tsip_st_sample.pjt

  This project builds the sample code for TSIP

Chapter 3

Integration Guide

This chapter discusses the **DSP/BIOS PSP DM648/C6452** package usage.

3.1 Application Usage of PSP

3.1.1 Demo Application

As part of the PSP package, a demo application is provided to do the following:

1. Display the PSP logo on UART Console
2. Run the sample code for the drivers

NOTE: In case if cache is enabled and driver is configured in EDMA mode or driver uses EDMA, application must provide cache line size (current configuration of cache line size is 64 – bytes) aligned address to the driver.

3.1.2 I2C Driver usage example

Sample code is provided to demonstrate the usage of I2C driver. This sample demonstrates usage of I2C driver by blinking LEDs on DM648 EVM through MSP430.

3.1.3 SPI Driver usage example

Sample code is provided to demonstrate the usage of SPI driver. This sample demonstrates usage of SPI driver by performing a read/write operation on EEPROM.

3.1.4 UART Driver usage example

UART driver sample code demonstrates read/write operations of 1000-bytes.

3.1.5 McASP Driver usage example

Sample code of McASP demonstrates basic audio record and playback functionality.

3.1.6 EDMA3 Driver usage example

Sample code is provided to demonstrate the usage of EDMA driver. EDMA has been tested in simple memory to memory copy on DM648 EVM board. The application demonstrates the copying of data through DMA channel and QDMA channel.

3.1.7 PAL SYS VLYNQ Driver usage example

VLYNQ sample application demonstrates the read/write operation on back to back connect DM648 EVMs through VLYNQ connectors. The sample first detects the VLYNQ link and than writes data on to the peer SOC and read data from the peer SOC.

3.1.8 PAL SYS PCI driver usage example

PCI driver sample application demonstrates read/write operation across PCI bus. This application has to be run on a DM648 EVM fit in PCI slot of LINUX machine. By running a same program on LINUX machine a memory mapped area on the host side can be obtained. This memory mapped address can be used by the sample application of DM648 EVM for read/write operation.

3.2 Instrumentation Tool - SoC Analyzer usage

3.2.1 Pre-requist installations for instrumentation

1. Install "SOC analyzer" by running dvt.exe of SocAnalyzer_1.0.0.1.1.

2. Install **DVSDK** targeted for DM6437 to setup the log server using the installable DM648_DVSDK_setupwin32_1_10_00_02.exe.

3. Set following as environment variables. (Assumption is dvsdk in installed in
   BIOS_INSTALL_DIR - D:/CCStudio_v3.3/bios_5_31_07
   BIOSUTILS_INSTALL_DIR:%BIOSDVSDK_INSTALL_DIR%\biosutils_1_00_02\
   BIOSDVSDK_INSTALL_DIR: C:\dvsdk_1_10_00_02

3.2.2 INSTRUMENATION IN SAMPLE PROJECT

1. Include the following files in application project that uses driver library.

   - %BIOSUTILS_INSTALL_DIR%\packages\ti\bios\log\ndk\examples\common\logserverstacksetup.c
   - %NDK_INSTALL_DIR%\packages\ti\ndk\example\network\cfgdemo\evmdm648\evmdm648_init.c
   - %NDK_INSTALL_DIR%\packages\ti\ndk\example\network\cfgdemo\evmdm648\cpsw3g_switchApp.c

2. Include the following Libraries in sample project of driver.

- %NDK_INSTALL_DIR%\packages\ti\ndk\lib\c64plus\hal\hal_ser_stub.lib
- %NDK_INSTALL_DIR%\packages\ti\ndk\lib\c64plus\hal\hal_timer_bios.lib
- %NDK_INSTALL_DIR%\packages\ti\ndk\lib\c64plus\hal\hal_userled_stub.lib
- %NDK_INSTALL_DIR%\packages\ti\ndk\lib\c64plus\miniPrintf.lib
- %NDK_INSTALL_DIR%\packages\ti\ndk\lib\c64plus\netctrl.lib
- %NDK_INSTALL_DIR%\packages\ti\ndk\lib\c64plus\nettool.lib
- %NDK_INSTALL_DIR%\packages\ti\ndk\lib\c64plus\os.lib
- %NDK_INSTALL_DIR%\packages\ti\ndk\lib\c64plus\stack.lib
- %NDK_INSTALL_DIR%\packages\ti\ndk\lib\hal\evmdm648\hal_eth_dm648.lib

3. Change "Debug" build configuration as per the following.

   Add the following compiler options, include file paths and include library paths and DSP-BIOS builder settings.

   Compiler options:

   -pds238

   Include paths:

   Open Build options of sample project. Go to "Compiler" tab. Select the "Preprocessor" category. Add the following paths in "include search path(-i)" field.

   -i"%NDK_INSTALL_DIR%\packages\ti\ndk\inc"
   -i"%NDK_INSTALL_DIR%\packages\ti\ndk\example\tools"
   -i"%BIOSUTILS_INSTALL_DIR%\packages\ti\bios\log\ndk"
   -i"%BIOSUTILS_INSTALL_DIR%\packages\ti\bios\log\support"
   -i"%NDK_INSTALL_DIR%\packages\ti\ndk\inc"
   -i"%NDK_INSTALL_DIR%\packages\ti\ndk\src\hal\evmdm648\ethss_dm648"
   -i"%BIOS_INSTALL_DIR%\packages"
   -i"%BIOSUTILS_INSTALL_DIR%\packages"

   "DspBiosBuilder" Settings

   Open Build options of sample project. Go to "DspBiosBuilder" tab. Select the "Basic" category. Add the following paths in "import path" field.

   %NDK_INSTALL_DIR%/packages/ti/ndk/inc/tci;
   %BIOSUTILS_INSTALL_DIR%\packages\ti\bios\utils;
   %BIOSUTILS_INSTALL_DIR%\packages\ti\bios\log\support;
   %BIOSUTILS_INSTALL_DIR%\packages\ti\bios\log\ndk\examples\common;

"XDC" Settings(This section is valid only when application project is RTSC compliant)

Open Build options of sample project. Go to "XDC" tab. Select the "Basic" category. Add the following paths in "XDC path" field.

%NDK_INSTALL_DIR%/packages/ti/ndk/inc/tci;
%BIOSUTILS_INSTALL_DIR%\packages\ti\bios\utils;
%BIOSUTILS_INSTALL_DIR%\packages\ti\bios\log\support;
%BIOSUTILS_INSTALL_DIR%\packages\ti\bios\log\ndk\examples\common;

"Linker" Settings:

Open Build options of sample project. Go to "Linker" tab. Select the "Library" category. Add the following paths in "Search Path(-i)" field.

-i"%BIOS_INSTALL_DIR%\packages\ti\rtdx\lib\c6000"
–i"%BIOSUTILS_INSTALL_DIR%\packages\ti\bios\log\ndk\lib"
-i"%BIOSUTILS_INSTALL_DIR%\packages\ti\bios\log\support\lib"
-i"%BIOSUTILS_INSTALL_DIR%\packages\ti\bios\utils\lib"

Add the following paths in "Incl. Libraries(-l)" field.

-l"logsupport.a64P"
-l"logservercgi.a64P"
-l"utils.a64P"

4. Create "iDebug" build configuration. Keep the compiler options, include file paths, include library paths and DSP-BIOS builder settings same as "Debug" configuration.

   Add the following lines in pjt of sample project to create "iDebug" configuration.

   Config="iDebug"
   ["Compiler" Settings: "iDebug"]

5. Repeat step 3 and 4 for release mode to generate iRelease configuration.

6. Do the following changes in TCF file of application project.

   - Remove "trace" object of LOG to avoid multiple definitions.
   - Add following lines in application project's TCF file:

     bios.GBL.CALLUSERINITFXN = 1;
     bios.GBL.USERINITFXN = prog.extern("evmdm648_init");

     This will set "evmdm648_init" as a global initialization function. This function will do the initialization required for network stack setup and read MAC address of EVM DM648 board

   - Add the following lines at the start of application project TCF file.

```
utils.importFile('ndk.tci');
utils.importFile('Load.tci');
utils.importFile('LogTrack.tci');
utils.importFile('logserverexample.tci');
```

- Increase the buffer size of "LOG_system" buffer = 262144, "DVTEvent_Log" buffer = 65536 and change the segment of both these buffers to DDR2 instead of IRAM setting following values in the TCF file:

  ```
  bios.LOG.instance("LOG_system").bufLen = 262144;
  bios.LOG.instance("LOG_system").bufSeg = prog.get("DDR2");
  ```

  Do the same for "DVTEvent_Log".

  ```
  bios.LOG.instance("DVTEvent_Log").bufSeg = prog.get("DDR2");
  bios.LOG.instance("DVTEvent_Log").bufLen = 1048576;
  ```

7. Do the following changes in main sample application of project.

- Add the following code in the main function(starting point) of application project. This is for initializing log server.

  ```
  LogAux_init();
  LogTrack_init();
  TRC_disable(TRC_LOGCLK); /* This is required only with BIOS 5.31.03 or less */
  ```

- Include the following files in main sample application of driver.

  ```
  #include <LogServerCgi.h>
  #include <LogAux.h>
  #include <LogTrack.h>
  ```

- Add the following code in file containing main function of application project.

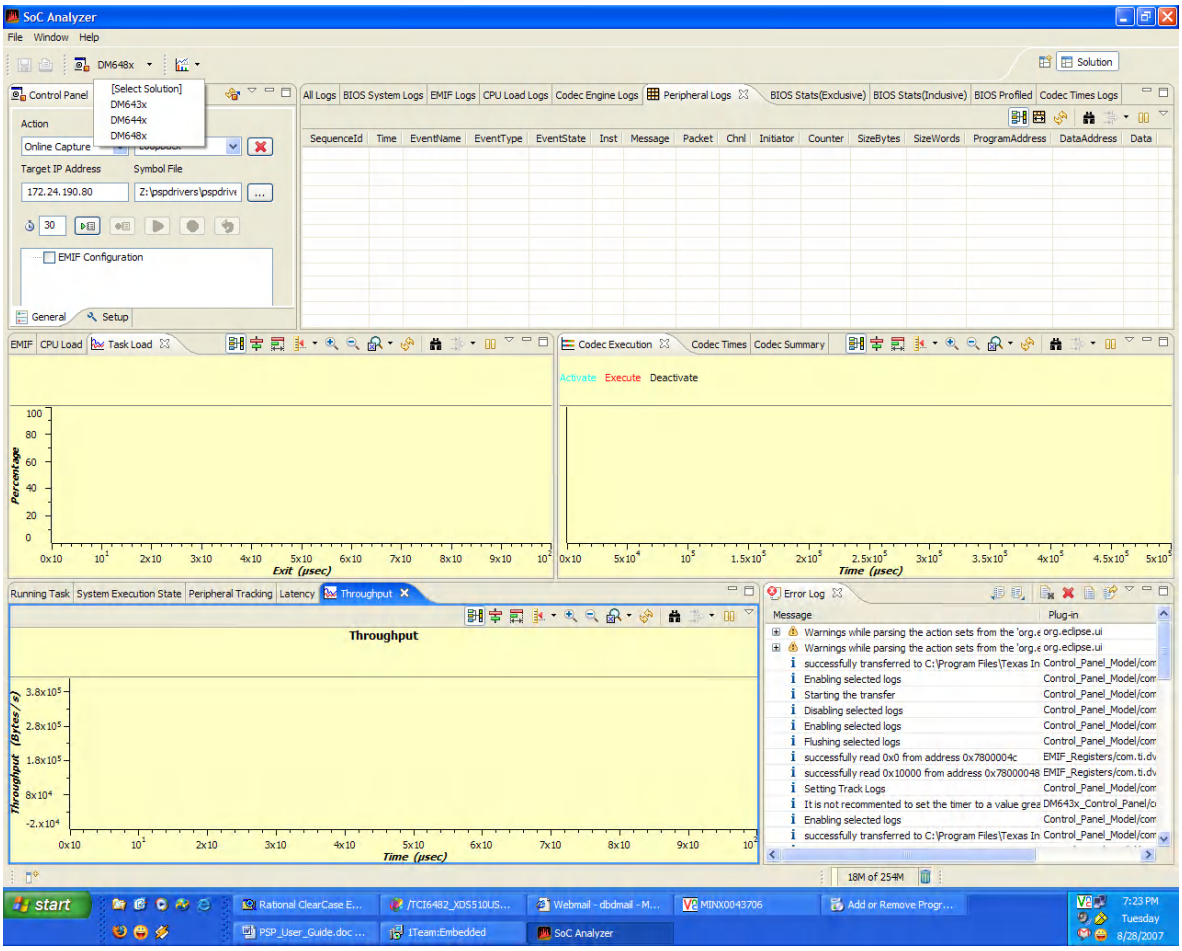
  ```
  extern LOG_Obj logTrace;
  extern LOG_Obj logTest;

  /*
   *  logTracePrd
   *
   *  This fxn runs periodically and continuously updates the
   *  logTrace log.
   *
   */
  Void logTracePrdfxn()
  {
     LOG_printf4(&logTrace, "logTrace data: %c %c %c %c\n", 'p', 'q', 'r', 's');
  }

  /*
   *  logTestPrd
   *
  ```

```
     *  This fxn runs periodically and continuously updates the
     *  logTest log.
     *
     */
    Void logTestPrdfxn()
    {
        LOG_printf4(&logTest, "logTest data: %c %c %c %c\n", 'h', 'i', 'j', 'k');
    }
/*  ======== task ========
 */
Void task(Arg id_arg, Arg time_arg)
{
    Int     id = ArgToInt (id_arg);
    Int     time = ArgToInt (time_arg);
    LgUns   currTime;
    LgUns   startTime;

    /* this outer while loop runs once per task run */
    while (1) {
        LOG_printf(&trace, "Task %d  starting!\n", id);
        startTime = CLK_getltime(); /* time when task begins */

        do {
            currTime = CLK_getltime();
        } while (currTime - startTime < time);

        LOG_printf(&trace, "Task %d going to sleep ...\n", id);
        TSK_sleep(300);
    }
}
```

8. Build the application project in iDebug or iRelease mode and run the out file. This will print the IP address of EVM DM648. Make a note of this IP address. Make sure that network cable is plugged in the EVM and connected to proper LAN port.
9. Make sure that there are enough logs(samples) generated so that SoC analyzer is able to plot the value

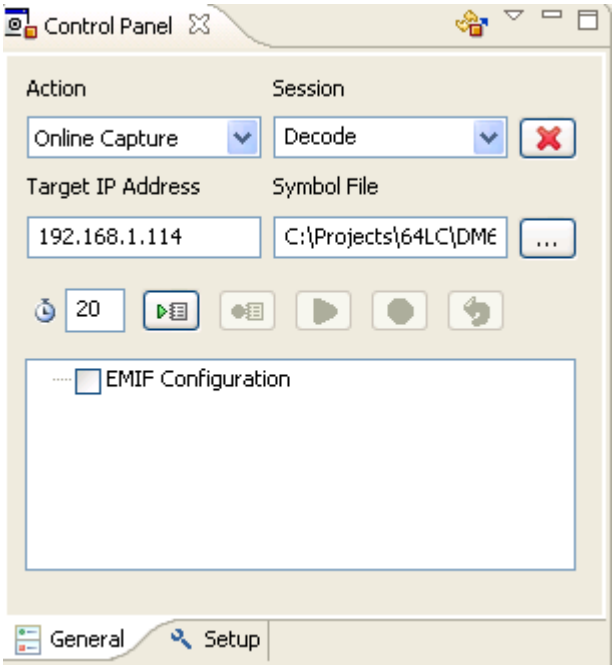
3.2.3 ANALYZING INFORMATION

1. Open SOC Analyzer. At the first use, the SoC Analyzer will start-up with a blank screen. In order to capture and visualize data, certain settings have to be done for SOC Analyzer tool. The settings have to be done in Solution control of SOC Analyzer. To open Solution control of SOC Analyzer, On the menu select solution control as shown in the picture below.(Please note that absolute values of throughput and latency are not proper, and this information is given to SoC tool team to look into the issue)



This will open following menu box. Select “DM648x”. This will open the control panel and solution for DM648.

2. In control panel tab, set “Action” to “Online Catpure” and set “Session” to “loopback”. Set the IP address note while running the application out file in “Target IP Address” field and set the path of OUT file that is running on EVM DM648 in “Symbol File” field as show in the following figure. Set the “Number of Seconds to Capture” filed to “60” which is set to “30” in the following figure.



3. Click on the "Start Capturing Logs on Target"  button. By pressing this button SOC analyzer will start capturing data.

4. To see various graphs go to Window->Open Views select the appropriate option.

3.3 Adding instance of the device driver

To have device driver included in the application, it is required to add the instance of the device driver in the TCF file of the application.

A TCI file can be used for adding the device driver instance in the TCF file. It contains following field:

1. **init function:** This function will be called before the initialization of drivers. Initializations that have to be done before driver initialization are done in this function

2. **function table ptr:** This is a pointer to a structure that contains all the function pointers to all the driver functions. This pointer should be initialized by a structure that contains function pointers to the functions the peripheral driver

3. **function table type:** Function table type which can be IOM_Fxns or DEV_Fxns

4. **device Id:** Device driver instance number

5. **device params ptr:** Pointer to a structure that will be used while initialization of drivers

The following is the example of the UART driver depicts the way a TCI file shall be made. A TCI file shall consists of the following detail:

```
bios.UDEV.create("UART0");
bios.UDEV.instance("UART0").deviceId = 0;
bios.UDEV.instance("UART0").fxnTableType = "IOM_Fxns";
bios.UDEV.instance("UART0").initFxn = prog.extern("uartAppInit");
bios.UDEV.instance("UART0").fxnTable = prog.extern("UARTMD_FXNS");
bios.UDEV.instance("UART0").params = prog.extern("Uart_DevParams");
```

Here

Field name	Description
UART0	The name of device driver instance. This name shall be given as input parameter while calling GIO_create for corresponding driver.
deviceId = 0	Refers to instance of device driver for UART0
IOM_Fxns	Type of function table
uartAppInit	Function that will be called to do the initialization required for UART driver
UARTMD_FXNS	Pointer to structure containing mini-driver functions for UART driver
UART_DevParams	the configuration parameter for device driver of UART0

This TCI file, containing the driver instance, need to be imported in the TCF file in the following way:

```
utils.importFile("dm648_uart0.tci");
```

This can be added in the TCF file by opening it text edit mode and then writing the above line in the starting. Here "dm648_uart0.tci" is the name of the respective TCI file of the device driver that needs to be included.

For this file to be imported, its path needs to be listed in the projects build option. The import path can be added by adding the path of the TCI file in project's build option-DspBiosBuilder-Basic.

3.4 Interrupt configurations in TCF file

Following lines need to be there in tcf file for enabling hardware interrupt.

bios.ECM.ENABLE = 1;

ECM configuration – following settings needs to be made manually to reflect these settings available in soc.h.

```
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
```