

SC-MCSDK 2.0 User Guide

Small Cell Multicore Software Development Kit

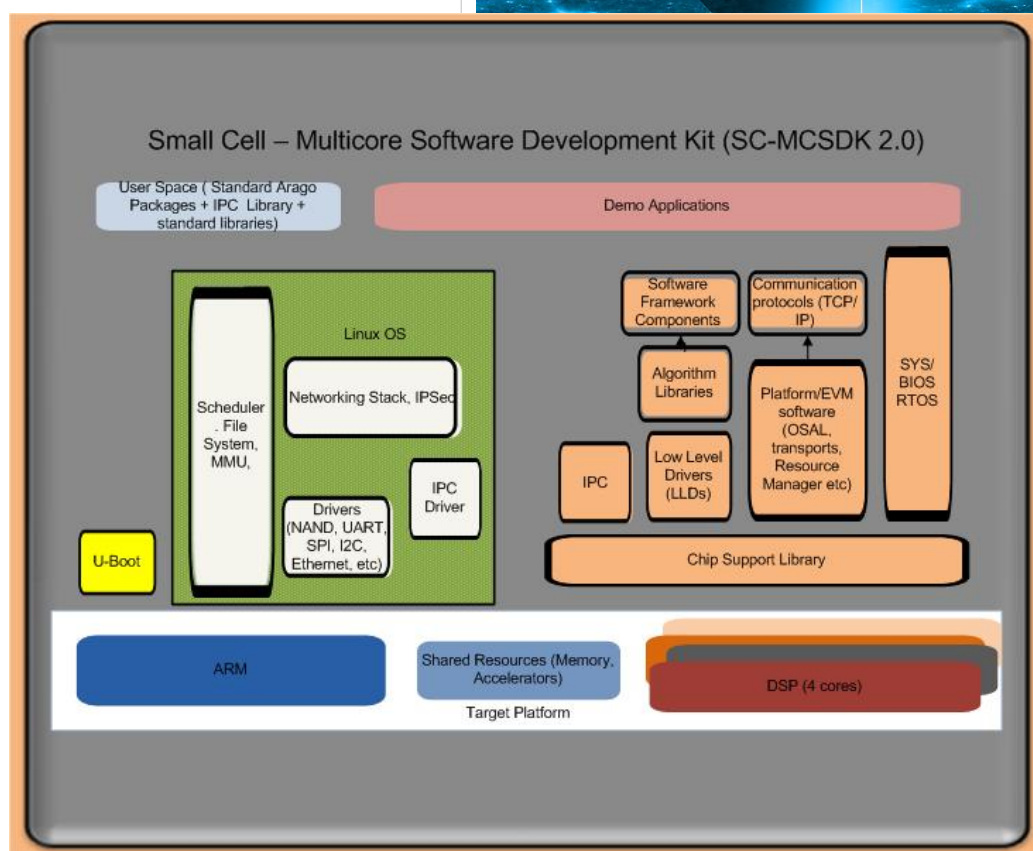
Version 2.x

User's Guide

Last updated: //

Overview

SC-MCSDK provides software components and tools to help develop softwares for the Small Cell Multi core target platforms. This User Guide provides the details of each of the components delivered in this SDK. The architecture of the SC-MCSDK software is given in the diagram below and consists of software components running on ARM and DSP cores that inter operate through the inter process communication (IPC) framework provided in the SDK.



After installing CCS and MCSDK, the components in the picture above will be located as follows:

Software Element	Location
CSL and Low Level Drivers	
Chip Support Library	pd_k_<platform>_w_xx_yy_zz/packages/ti/csl/
All LLD (except EDMA3)	pd_k_<platform>_w_xx_yy_zz/packages/ti/drv/ - If the driver is supported for a given platform it will be located in the drv/ directory
EDMA3 LLD	edma3_lld_w_xx_yy_zz/
Algorithm Libraries	
DSPLIB	dsplib_<proc_type>_w_x_y_z/
IMGLIB	imglib_<proc_type>_w_x_y_z/
MATHLIB	mathlib_<proc_type>_w_x_y_z/
Platform/EVM Software	
Platform Library	pd_k_<platform>_w_xx_yy_zz/packages/ti/platform/<device>/platform_lib
Resource Manager	pd_k_<platform>_w_xx_yy_zz/packages/ti/platform/resource_mgr.h (Note: There is also a RM LLD provided for resource management)
Platform OSAL	pd_k_<platform>_w_xx_yy_zz/packages/ti/platform/platform.h
Transports	pd_k_<platform>_w_xx_yy_zz/packages/ti/transport/ipc/qmss/ - QMSS IPC Transport
	pd_k_<platform>_w_xx_yy_zz/packages/ti/transport/ipc/srio/ - SRIO IPC Transport
	pd_k_<platform>_w_xx_yy_zz/packages/ti/transport/ndk - NDK Transport
POST	mcsdk_w_xx_yy_zz/tools/post/
Bootloader	mcsdk_w_xx_yy_zz/tools/boot_loader/
Target Software Components	
SYS/BIOS RTOS	bios_w_xx_yy_zz/
Interprocessor Communication	ipc_w_xx_yy_zz/
Network Developer's Kit (NDK) Package	ndk_w_xx_yy_zz/
System Library	syslib_w_xx_yy_zz/
Demonstration Applications	
Image Processing	mcsdk_w_xx_yy_zz/demos/image_processing/

The release will be delivered as installjammer created self installer for Windows and Linux.

Please refer to *Getting Started Guide* for release directory structure.

When installjammer is used, choose the release folder when the GUI prompt for the same. All release documents will be available in the top level release folder.

The document is arranged into two sections :- ARM and DSP to describe the software run on the respective CPUs.

Acronyms and Definitions

The following acronyms are used throughout this document.

Acronym	Meaning
AMC	Advanced Mezzanine Card
CCS	Texas Instruments Code Composer Studio
CSL	Texas Instruments Chip Support Library
DDR	Double Data Rate
DHCP	Dynamic Host Configuration Protocol
DSP	Digital Signal Processor
DVT	Texas Instruments Data Analysis and Visualization Technology
EDMA	Enhanced Direct Memory Access
EEPROM	Electrically Erasable Programmable Read-Only Memory
EVM	Evaluation Module, hardware platform containing the Texas Instruments DSP
HUA	High Performance Digital Signal Processor Utility Application
HTTP	HyperText Transfer Protocol
IP	Internet Protocol
IPC	Texas Instruments Inter-Processor Communication Development Kit
JTAG	Joint Test Action Group
MCSA	Texas Instruments Multi-Core System Analyzer
SC-MCSDK	Texas Instruments Small Cell Multi-Core Software Development Kit
NDK	Texas Instruments Network Development Kit (IP Stack)
NIMU	Network Interface Management Unit
PDK	Texas Instruments Platform Development Kit
RAM	Random Access Memory
RTSC	Eclipse Real-Time Software Components
SRIO	Serial Rapid IO
TCP	Transmission Control Protocol
TI	Texas Instruments
UART	Universal Asynchronous Receiver/Transmitter
UDP	User Datagram Protocol
UIA	Texas Instruments Unified Instrumentation Architecture
USB	Universal Serial Bus

Note: We use the abbreviation TMS when referring to a specific TI device (processor) and the abbreviation TMD when referring to a specific platform that the processor is on. For example, TMS320TCI664 refers to the TCI6614 SoC and TMDXEVM6614 refers to the actual hardware EVM that the processor is on.

Supported Devices/Platforms

The latest SC-MCSDK Release supports the following Texas Instrument devices/platforms:

Platform Development Kit	Supported Devices	Supported EVM
TCI6614	TMS320CTCI6614	TMDXEVM6614

Other Resources

Training

This section provides a collection links to training resources relevant to this release.

Link	Description
MCSDK Overview Online [1]	This video training module provides an overview of the multicore SoC software for C66x devices. This module introduces the optimized software components that enable the rapid development of multicore applications and accelerate time to market using foundational software in the MCSDK. The MCSDK also enables developers to evaluate the hardware and software capabilities using the C66x evaluation module.
KeyStone Architecture Wiki [2]	KeyStone Architecture Overview Mediawiki
KeyStone Architecture Online [3]	C66x Multicore SOC Online Training for KeyStone Devices
SYS/BIOS Online [4]	SYS/BIOS Online Training
SYS/BIOS 1.5 Day [5]	SYS/BIOS 1.5-DAY Workshop
MCSA Online [6]	Multicore System Analyzer Online Tutorial

MCSDK Information

The following resources are a good place to start for basic information on the Multicore Software Development Kit.

Document	Description
MCSDK White Paper [7]	This paper introduces TI's Multicore Software Development Kit (MCSDK) by outlining the various software packages available, along with utilities and tool chains that can aid programmers in development for high-level operating systems such as Linux, and the real time operating system SYS/BIOS.
BIOS-MCSDK Short Video [8]	This short video describes what the BIOS Multicore Software Development Kit is and how it helps customers get to market faster.

Getting Started Guides

The getting started guides walk you through setting up your EVM and running the "Out of Box" Demonstration application. This is where you should start after receiving your EVM.

Document	Description
SC-MCSDK Release Notes	Contains latest information on the release including what's changed, known issues and compatibility information. Each foundational component will have individual release notes as well.
SC-MCSDK Getting Started Guide	Discusses how to install the SC-MCSDK and access the demonstration application.
TMDXEVM6614 Quick Setup Guide	Quick Setup Guides showing how to set up the EVM and run the Out of Box demonstration application from flash. These documents can be found in the links provided below for <i>Hardware - EVM Overview</i> .

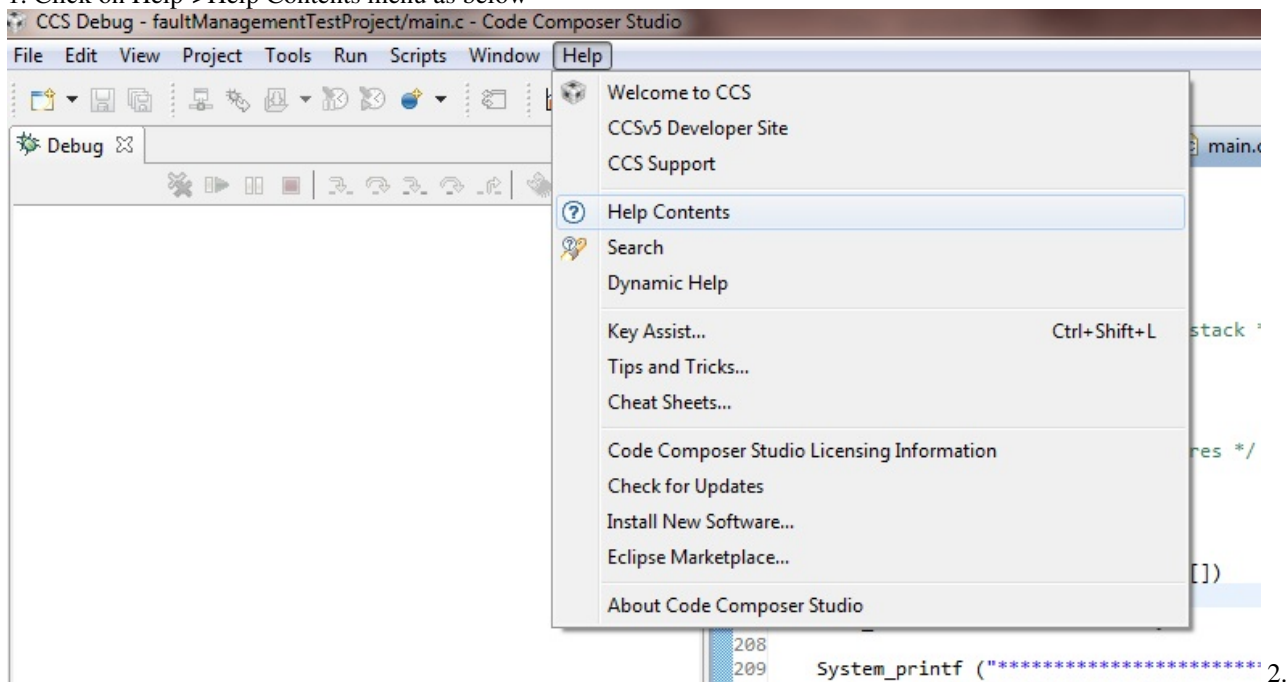
Document map

Code composer Studio provides the document map for the components that were recognized by CCS via eclipse. The below example shows how to get to System Analyzer's documentation/API reference using CCS.

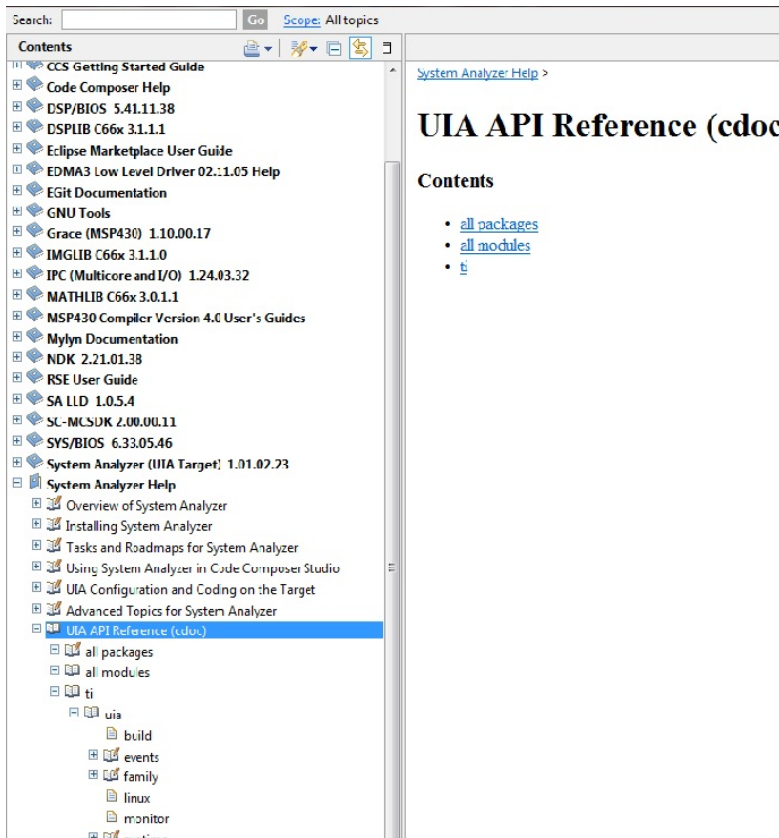


Note: Please note that every component provides the documentation on its own way

1. Click on Help->Help Contents menu as below



You should see below help and expand on System analyzer for API reference help as well as documentation on UIA.



ARM

Introduction

This section provides information on the features, functions, delivery package and compile tools for the Linux software release for TCI6614 Simulator and EVM platforms. This document describes how to install and work with Texas Instruments' Linux software release for the TCI6614 Simulator and EVM platforms. The release provides a fundamental Linux based software platform for development, deployment and execution on the ARM processor on the TCI6614 platforms. In this context, the document contains instructions to:

- Install the release, and
- Build the sources contained in the release
- Run software on the target platform
- Detailed description of various software components delivered in the release

Installation Guide

Prerequisites for Installation

Before you begin with the installation of this package please make sure you have met the following system requirements:

- **Windows Host:** Windows host is used for running Tera Term or Hyper Link for connecting to UART. This is also used for running CCSv5 to connect to EVM and run Linux.
- **Linux Host:** The Linux host is necessary for build and development. The recommended Linux Host OS is Ubuntu 10.04. This host is also used as servers for NFS, TFTP, etc. The Linux host can also be used for running CCSv5, connecting to the Simulator platform and run Linux.

Note: Unix shell commands shown in this document assume a bourne style shell (sh, bash, dash, etc.). These commands may not operate as expected if executed in a different shell environment (e.g. C-shell).

Toolchain Installation

To build the Linux kernel or U-Boot, you will need to download and install a ARM tool chain. To build with CodeSourcery ARM tool chain, install CodeSourcery version 2009-Q1 (<https://sourcery.mentor.com/sgpp/portal/release858>). Use "chmod +x <installer-executable>" to make this installer executable. Execute the installer and step through the GUI to install the toolchain on your system. A newer toolchain (Linaro 2012.03 based version toolchain) can be used to build with thumb2 mode enabled in the kernel. To download the installer, do:

```
wget https://launchpad.net/linaro-toolchain-binaries/trunk/2012.03/+download/gcc-linaro-arm-linux-gnueabi-2012.03-20120326_linux.tar.bz2
tar -jxf gcc-linaro-arm-linux-gnueabi-2012.03-20120326_linux.tar.bz2
```

For latest releases of SC-MCSDK (02.02.x), another latest Linaro toolchain 2013.03 is being used. This can be downloaded with:

```
wget https://launchpad.net/linaro-toolchain-binaries/trunk/2013.03/+download/gcc-linaro-arm-linux-gnueabi-4.7-2013.03-20130313_linux.tar.bz2
tar -jxf gcc-linaro-arm-linux-gnueabi-4.7-2013.03-20130313_linux.tar.bz2
```

Texas Instruments Code Composer Studio (CCS) version 5.1 will also need to be installed on the Linux host machine. CCS is used to run the TCI6614 Simulator. Please follow the website link provided in the SC-MCSDK release notes to download and install CCSv5. To help you get started quickly, the Linux software release comes with pre-built binaries. However, after making any changes to the Linux Kernel you need to cross-compile them using the CodeSourcery toolchain and use the new binaries that are generated

Linux Software Installation

The Linux kernel and u-boot pre-built binaries provided in the release under the folder `sc_mcsdk_linux_<version>`.

The following is a more detailed listing of the contents under this folder:

```
release_folder
  ti
    sc_mcsdk_linux_<version>
      bin
      boot
      docs
      host-tools
      loadlin
```

```

u-boot
images
linux-devkit

```

- host-tools/loadlin
 - contains dss scripts and collaterals for loading and running Linux kernel on the Simulator
- loadlin.xsl
- appleton.ccxml - simualtor configuration file for ccs
- tci6614-evm.ccxml- EVM configuration file for ccs
- ti-scmcsdk-rootfs-tci6614-evm.cpio- canned min-root file system for EVM - used by initrd during kernel boot up
- loadlin-sim.js - dss script that loads and runs Linux kernel on simulator.
- loadlin-evm.js - dss script that loads and runs Linux kernel on the EVN.
- tracelog.xml - trace file
- images
 - zImage-tci6614-evm.bin
 - precompiled Linux kernel image suitable for loading on to the EVM through CCS.
 - uImage-tci6614-evm.bin
 - A precompiled Linux kernel image suitable for loading through u-boot.
 - vmlinux-tci6614-evm
 - vmlinux that has symbols and may be loaded to EVM for source level debugging
 - tci6614-evm.dtb
 - dtb image for evm.
 - u-boot-tci6614-evm.bin
 - A precompiled U-Boot image in blob binary format suitable for loading on EVM through CCS
 - u-boot_hdr_tlr.bin
 - u-boot image formatted to flash to NAND
 - tci6614-evm-ubifs.ubi
 - UBI image to flash to ubifs partition from u-boot or from Linux console
 - ubinize.cfg
 - ubinize config file used for creating tci6614-evm-ubifs.ubi
 - ti-scmcsdk-rootfs-tci6614-evm-20130718024830.rootfs.ubifs
 - Full rootfs ubifs image with sc-mcsdk applications. Use this to write to the rootfs volume.
 - ti-scmcsdk-rootfs-tci6614-evm-20130718024830.rootfs.tar.gz
 - Full rootfs tar file.
 - ti-scmcsdk-recoveryfs-tci6614-evm-20130718032327.rootfs.ubifs
 - Recovery rootfs ubifs image. Use this to write to rootfs-recovery volume.
 - ti-scmcsdk-recoveryfs-tci6614-evm-20130718032327.rootfs.tar.gz
 - Recovery rootfs tar file.
- host-tools
 - evmc6614lxe_<version>.gel - Gel file used for booting Linux through CCS.

Linux Kernel/U-Boot Build Instructions

Source Repositories

The Linux Kernel and U-Boot source files are not provided in the release, but the source is available in the public domain.

Component	Location
Linux Kernel	[9]
U-Boot	[10]

Note: Arago infrastructure ^[11] is being used to build the SC-MCSDK images. To build the SC-MCSDK release please follow the instructions in the section to configure and set up build environment.

Build Prerequisites

- Install Linaro toolchain as per the "Toolchain Installation" section above.
- Configure environment variables for toolchain and architecture. These may be added to your .bashrc or shell initialization script:

```
export PATH=<path-to-arm-toolchain-bin>:$PATH
export CROSS_COMPILE=arm-linux-gnueabi-
export ARCH=arm
```

- Install and configure git in the Ubuntu machine. Use the following command to install git:

```
$ apt-get install git-core
```

- To configure git please refer here[12]. If your network is behind a proxy, those settings need to be configured as well.

- Packages needed at build-time can be fetched with a simple command on Ubuntu 12.04:

```
$ sudo apt-get install build-essential subversion ccache sed wget cvs coreutils unzip texinfo
docbook-utils gawk help2man diffstat file g++ texi2html bison flex htmldoc chrpath libxext-dev
xserver-xorg-dev doxygen bitbake
```

If you are running a distribution other than Ubuntu 12.04, please refer to your distribution documentation for instructions on installing these required packages

Proxy Setup

If your network is behind a firewall/proxy additional settings are needed for bitbake to be able to download source code repositories for various open source projects. Some of these configuration items are:

- wgetrc: A ".wgetrc" needs to be created under the \$HOME directory. A sample wgetrc can be found here[13]. Please update configuration variables http_proxy, https_proxy and ftp_proxy as per your network environment.
- Set proxy environment variables. These may be added to your .bashrc or shell initialization script:

```
export http_proxy="http://<your_proxy>:<port>"
export ftp_proxy="http://<your_proxy>:<port>"
export https_proxy="http://<your_proxy>:<port>"
```

- \$HOME/.subversion/servers needs to be updated if the network is behind a proxy. The following lines need to be modified as per settings for your network:

```
http-proxy-exceptions = "exceptions"
http-proxy-host = "proxy-host-for-your-network"
http-proxy-port = 80
```

Setting up Git Repositories

The only repository needed to build SC-MCSDK images through Yocto build infrastructure is oe-layerssetup-mcsdk.git.

- Create the SC-MCSDK repository directory:


```
$ mkdir $HOME/sc-mcsdk;
cd $HOME/sc-mcsdk
```
- Clone Yocto build base repository


```
$ git clone git://arago-project.org/git/people/hzhang/oe-layerssetup-mcsdk.git
```

Configuring Yocto

Yocto needs to be configured for the local environment. The following steps set up a default configuration that may be customized as needed:

- Clone the sample environment settings


```
$ cd $HOME/sc-mcsdk/oe-layerssetup-mcsdk
$ ./oe-layertool-setup.sh -f configs/sc-mcsdk/02.02.00.00-config.txt
```
- If needed, edit the bitbake configuration file oe-layerssetup-mcsdk/conf/local.conf to customize your bitbake build. In particular, enabling parallel task execution substantially speeds up the Yocto build process by fetching package sources in parallel with other build steps. In order to do so, please enable and edit the following parameter in your bitbake configuration `BB_NUMBER_THREADS = "4"` **Note:** The `BB_NUMBER_THREADS` definition is not the number of CPUs on your SMP machine. The value of 4 appears to work quite well on a single core system, and may be adjusted upwards on SMP systems.

Building Linux Kernel, u-boot and file systems using Yocto build environment

Assuming all the above instructions worked well, the \$HOME/sc-mcsdk is now ready for building Linux kernel through Yocto.

Type in the following commands to start the build for TCI6614 EVM:

```
$ cd $HOME/sc-mcsdk/oe-layerssetup-mcsdk/build
$ source conf/setenv
$ MACHINE=tc6614-evm TOOLCHAIN_BRAND=linaro bitbake ti-scmcsdk-rootfs-image
```

To create sc-mcsdk-recovery-image do

```
$ MACHINE=tc6614-evm TOOLCHAIN_BRAND=linaro bitbake ti-scmcsdk-recoveryfs-image
```

Once build is complete, the images for Linux kernel and file system can be located at

```
$Home/sc-mcsdk/oe-layers/setup-mcsdk/build/arago-tmp-external-linaro-toolchain/deploy/images/
```

To create the ubi image, please follow the steps in the section on UBI/UBIFS

Building Linux Kernel outside Yocto

- Clone linux-tci6614 git tree
\$ git clone git://arago-project.org/git/projects/linux-tci6614.git
- Build Linux Kernel

This section assumes that the tool chain for ARM (Code sourcery or Linaro toolchain) is installed and environment variables are set up as per instructions given under Build Prerequisites

```
$ cd linux-tci6614
$ git reset --hard <release tag>
```

Where <release tag> is the tag used for a release and can be obtained from the SC-MCSDK release note.

To build Linux kernel for EVM with newer (linaro) toolchain (that can support thumb2 mode in kernel), do

```
$ export CROSS_COMPILE=arm-linux-gnueabi-
$ make tci6614-evm-thumb2-defconfig
$ make uImage KALLSYMS_EXTRA_PASS=1 EXTRAVERSION=`echo ${SRCREV} | sed
's/DEV.SC-MCSDK//g' | sed 's/P/-p/g`
```

NOTE: Here SRCREV is the release tag, for example: DEV.SC-MCSDK-02.00.00.12. The need to do this is because: when the filesystem is built from yocto, this flag is passed onto any kernel modules that are being built. So, if a user uses the kernel delivered from platform release, everything will match. But if the user wants to build the kernel separately - this step is needed.

To build dtb (device tree blob)

```
$ make tci6614-evm.dtb
```

The uImage, Image, zImage and tci6614-evm.dtb will be available under arch/arm/boot directory.

NOTE: Need to build and use latest tci6614-evm.dtb on target to make sure this is in sync with the latest Linux kernel .

Building U-Boot outside Yocto

- \$ git clone git://arago-project.org/git/projects/u-boot-tci6614.git
- \$ cd u-boot-tci6614
- \$ git reset --hard <release tag>

Where <release tag> is the tag used for a release and can be obtained from the SC-MCSDK release note.

This section assumes that the Code Sourcery tool chain for ARM is installed and environment variables are set up as per instructions given under Build Prerequisites

To build u-boot execute the following commands:-

```
$ make tci6614-evm-config
$ make
```

Running Linux Kernel on Simulator

Note: this section should be skipped if the purpose is to run the kernel on the EVM. No need to install the simulator.

Installing Simulator

The SC-MCSDK 2.0 uses release 0.9.1 of the Simulator for TCI6614.

- Download and install CCS v5.1 on Ubuntu Linux 10.04 or on Windows Host. Installing CCS 5.1 is explained in section "Installing Code Composer Studio" above. Use appropriate installer based on your Host.
- Install the Appleton simulator as per instructions at [14]. Install the Appleton simulator in the CCSv5 path, which is to be selected when the self-installer prompts for the CCS installation path. **IMPORTANT** Please note that the CCSv5 path should be <CCSV5_INSTALLATION_PATH>\ccsv5\ccs_base\

On selecting the correct CCSv5 path, the Appleton simulator binaries would be installed by the Appleton self-installer.

Currently there are issues when enabling cache. So as a work around, add a line for NEW_MIF OFF (shown with arrow below) in ccsv5/ccs_base/simulation_csp_aptn/bin/configurations/tisim_tci6614_pv.cfg

tisim_tci6614_pv.cfg

```
MODULE ARM_CORE_HOST_A8;
    MODULE ARM_CORE;
        SIM_DLL;
        .....
    END ARM_CORE;

    MODULE MEMORY_MODULE;
        ....
        MEMORY_TYPE          CORTEXA8_FUNC;
        NEW_MIF OFF; <==== Add this line
        USE_SSI 0;
        SEVERITY              LOW;
    END MEMORY_
```

Loading and running Linux Kernel on Simulator

The dss script, loadlin-sim.js is used for loading and running Linux kernel on the simulator. The script is located in the linux release folder release-folder/sc-mcsdk-<version>/host-tools/loadlin

```
$cd <release-folder/sc-mcsdk-<version>/linux/>/host-tools/loadlin
```

- Edit the loadlin-sim.js script and change following variables

CFG_LOADER_BASE = <= Give path of your loadlin directory here. So to use the loadlin from release folder, you can point this to loadlin folder under linux/host-tools/loadline directory

CFG_SRC_BASE = <= Give path of your vmlinux and Image binaries. For using the images from the release, you can point this to linux/images directory from release folder.

If you are building your own kernel, then point CFG_SRC_BASE to the kernel tree and change following variables in the script as shown:-

```
var pathKernel = CFG_SRC_BASE + "/arch/arm/boot/Image";
```

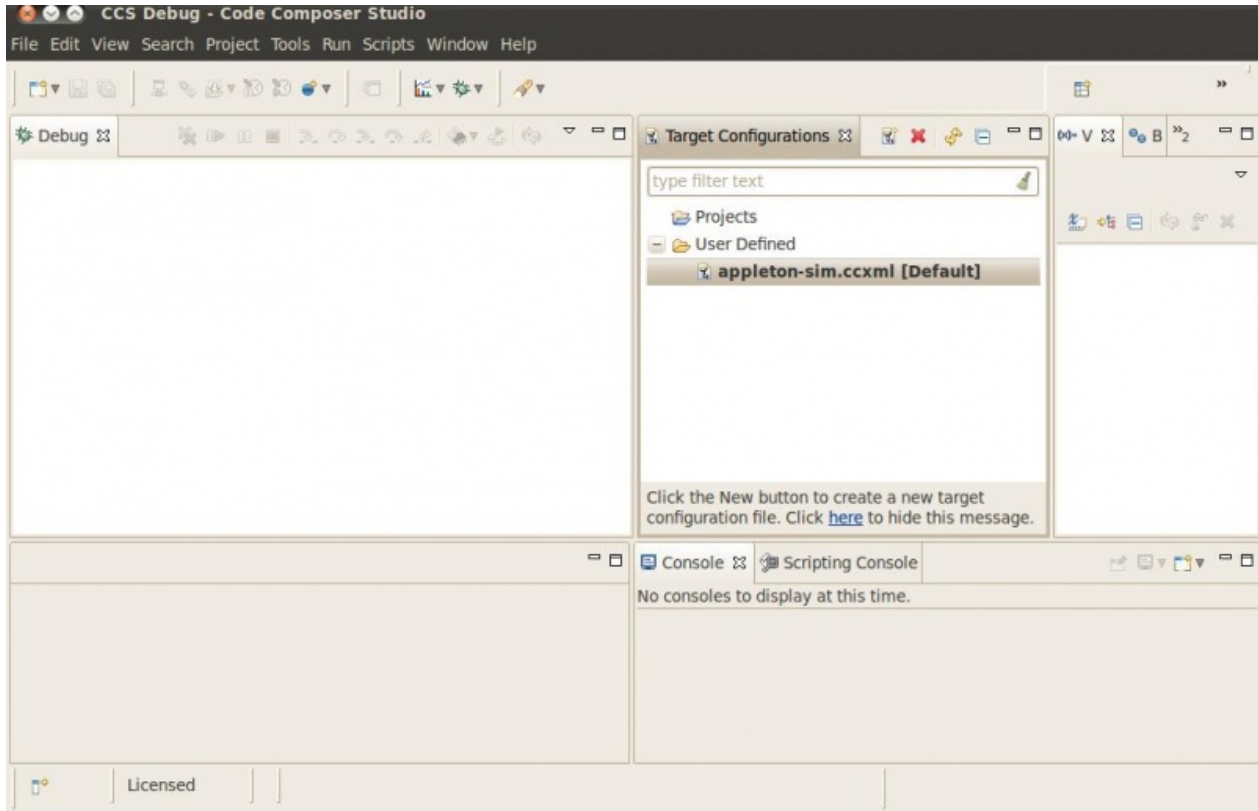
```
var pathVmlinux = CFG_SRC_BASE + "/vmlinux";
```

This will pick the Image and vmlinux from the kernel tree.

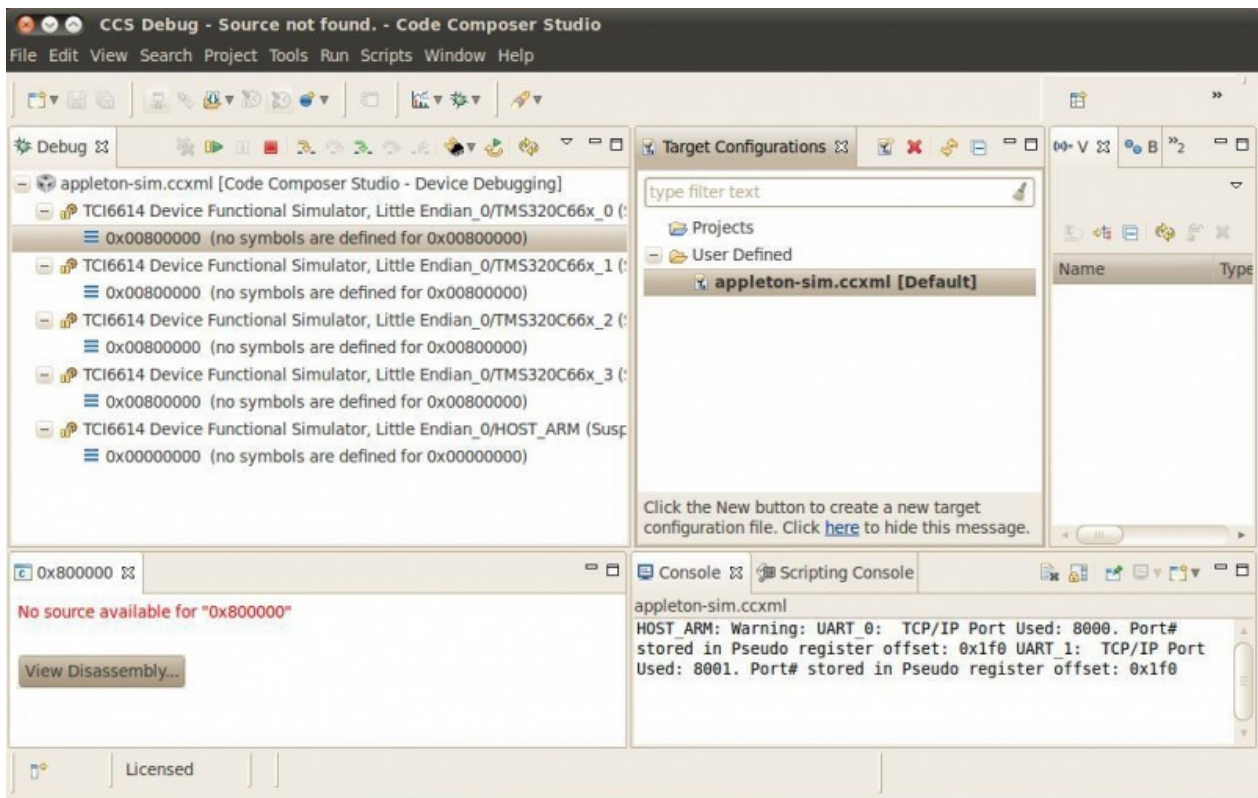
copy the file appleton.ccxml from loadlin folder to ~/user/CCSTargetConfiguration folder where the CCS saves the user specific configuration file. Rename the file to

appleton-sim.ccxml

Start CCS by clicking on to the CCS desktop icon. Click View -> Target Configuration. Under Target configuration window, you will find appleton-sim.ccxml. Right click and launch Selected configuration.



UART simulation is done through telnet. So after launching the configuration, the UART0 and UART1 telnet ports appear on the console window as shown:- . In the picture below, port 8000 is used for UART0 and 8001 for UART1. Telnet to port 8000 as



\$telnet localhost 8000

The Simulator alternate between 8000 and 8001. So if you can't connect to 8000, try connecting 8001

The telnet session will show as below

```

a0868495@uda0868495: ~/work/LoadLin
File Edit View Terminal Help

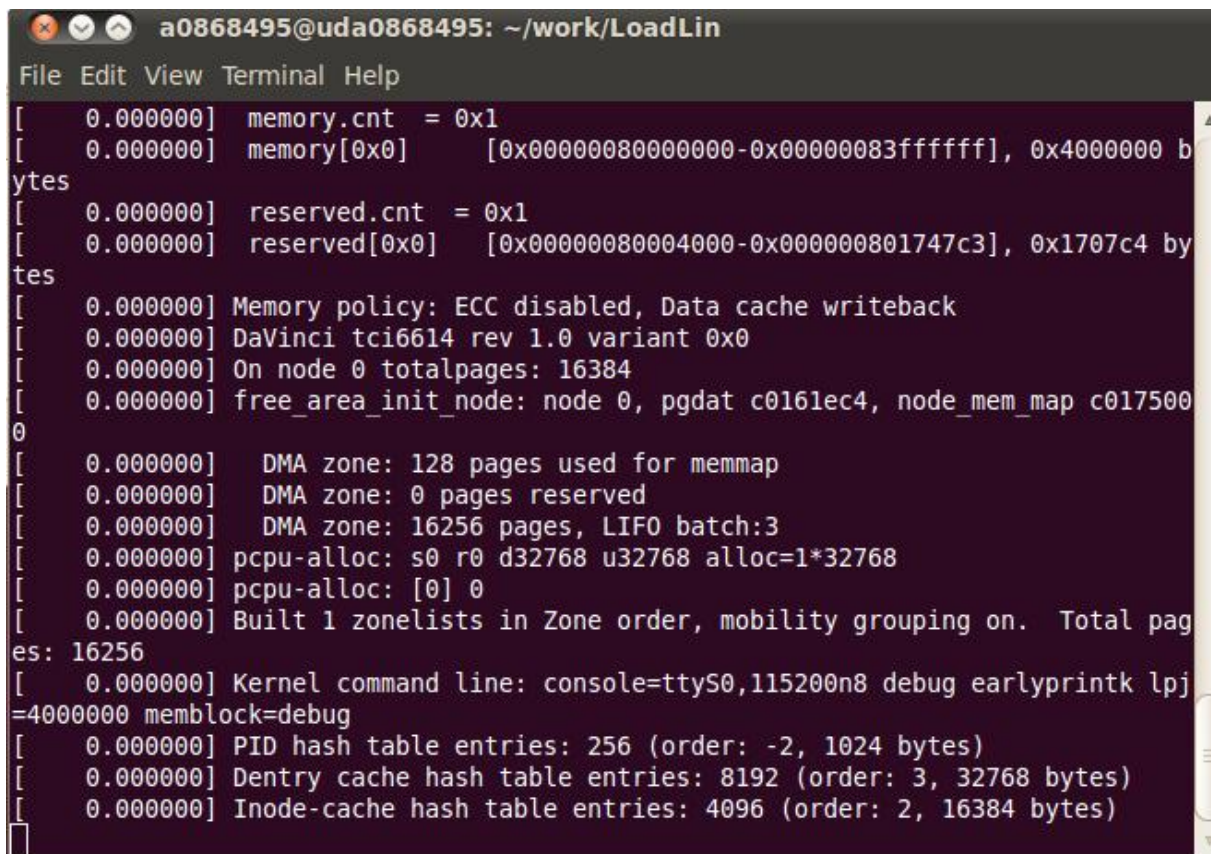
LoadLin
mcsdk-beta2
Simulation-APPLETON-0.8.0-Linux-x86-Install
syslink.commits
tci6614-mega-patch.txt
a0868495@uda0868495:~/work$ cd LoadLin/
a0868495@uda0868495:~/work/LoadLin$ ls
appleton.ccxml  Image-0810  loadlin.js  tracelog.xml  vmlinux-0810
Image           Image.old   loadlin.xsl vmlinux        vmlinux.old
a0868495@uda0868495:~/work/LoadLin$ pwd
/home/a0868495/work/LoadLin
a0868495@uda0868495:~/work/LoadLin$ /home/a0868495/work/LoadLin/loadlin.js^C
a0868495@uda0868495:~/work/LoadLin$ telnet localhost 8000
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Connection closed by foreign host.
a0868495@uda0868495:~/work/LoadLin$ telnet localhost 8000
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

```

Open the Scripting console by selecting View -> Scripting Console. In the scripting console, type following:-

```
js>loadJSFile <<release-folder/sc-mcsdk-<version>/linux/>/host-tools/loadlin/loadlin-sim.js >
```


Once this is done, run the target by selecting the ARM core and clicking on to Resume button. Linux Kernel will boot up as shown:-



```

a0868495@uda0868495: ~/work/LoadLin
File Edit View Terminal Help
[ 0.000000] memory.cnt = 0x1
[ 0.000000] memory[0x0] [0x0000000800000000-0x000000083ffffff], 0x4000000 bytes
[ 0.000000] reserved.cnt = 0x1
[ 0.000000] reserved[0x0] [0x0000000800040000-0x0000000801747c3], 0x1707c4 bytes
[ 0.000000] Memory policy: ECC disabled, Data cache writeback
[ 0.000000] DaVinci tci6614 rev 1.0 variant 0x0
[ 0.000000] On node 0 totalpages: 16384
[ 0.000000] free_area_init_node: node 0, pgdat c0161ec4, node_mem_map c0175000
[ 0.000000] DMA zone: 128 pages used for memmap
[ 0.000000] DMA zone: 0 pages reserved
[ 0.000000] DMA zone: 16256 pages, LIFO batch:3
[ 0.000000] pcpu-alloc: s0 r0 d32768 u32768 alloc=1*32768
[ 0.000000] pcpu-alloc: [0] 0
[ 0.000000] Built 1 zonelists in Zone order, mobility grouping on. Total pages: 16256
[ 0.000000] Kernel command line: console=ttyS0,115200n8 debug earlyprintk lpj=4000000 memblock=debug
[ 0.000000] PID hash table entries: 256 (order: -2, 1024 bytes)
[ 0.000000] Dentry cache hash table entries: 8192 (order: 3, 32768 bytes)
[ 0.000000] Inode-cache hash table entries: 4096 (order: 2, 16384 bytes)

```

Creating .cpio file from *.tar.gz file

The latest file system images will be available in tar.gz format. This section describes how to convert it to the .cpio format expected by the loadlin script.

```

$ mkdir ~/fileSYS
$ cd fileSYS
$ sudo tar -xvzf <path of *.tar.tgz file>
$ find . | cpio -H newc -o -O <path of output .cpio file>

```

At this point .cpio file is created.

Bootling Linux kernel using a modified file system

The loadlin folder has canned root file system images(arago-min-root-sim.cpio & arago-min-root-evm.cpio). This section describes how to add file to this file system and then boot up kernel using the modified filesystem. Use appropriate rootfs file based on the target platform.

To add file to this filesystem, follow the steps below:- assuming user has modified CFG_LOADER_BASE and CFG_SRC_BASE variables to point to loadlin folder

```

$ mkdir ~/fileSYS
$ cd fileSYS
$ sudo cpio -i -F <release-folder>/sc-mcsdk-<version>/linux/host-tools/loadlin/arago-min-root.cpio

```

At the point a flat file system will be available under ~/fileSYS

```

$ ls

```

bin boot dev etc home lib media mnt proc sbin sys tmp usr var

Copy required files to this filesystem. You will have to use sudo command to copy files to this folder.

To recreate the cpio file, invoke

```
$ find . | cpio -H newc -o -O release-folder>/sc-mcsdk-<version>/linux/host-tools/loadlin/initramfs.cpio
```

This will create initramfs.cpio under loadlin. currently initrd size is set to 8M. Edit the loadlin-sim.js/loadlin-evm.js as needed and modify the following:- Change CFG_INITRD_SIZE if filesystem is bigger than 9M as follows:- // To change this to 10M var CFG_INITRD_SIZE = 10 * 1024 * 1024 set pathInitrd to "/initramfs.cpio"

The loadlin-evm.js or loadlin-sim.js file is now ready to run.

Note: kernel boot up time increases with increase in size of the file system.

Running Linux Kernel on EVM

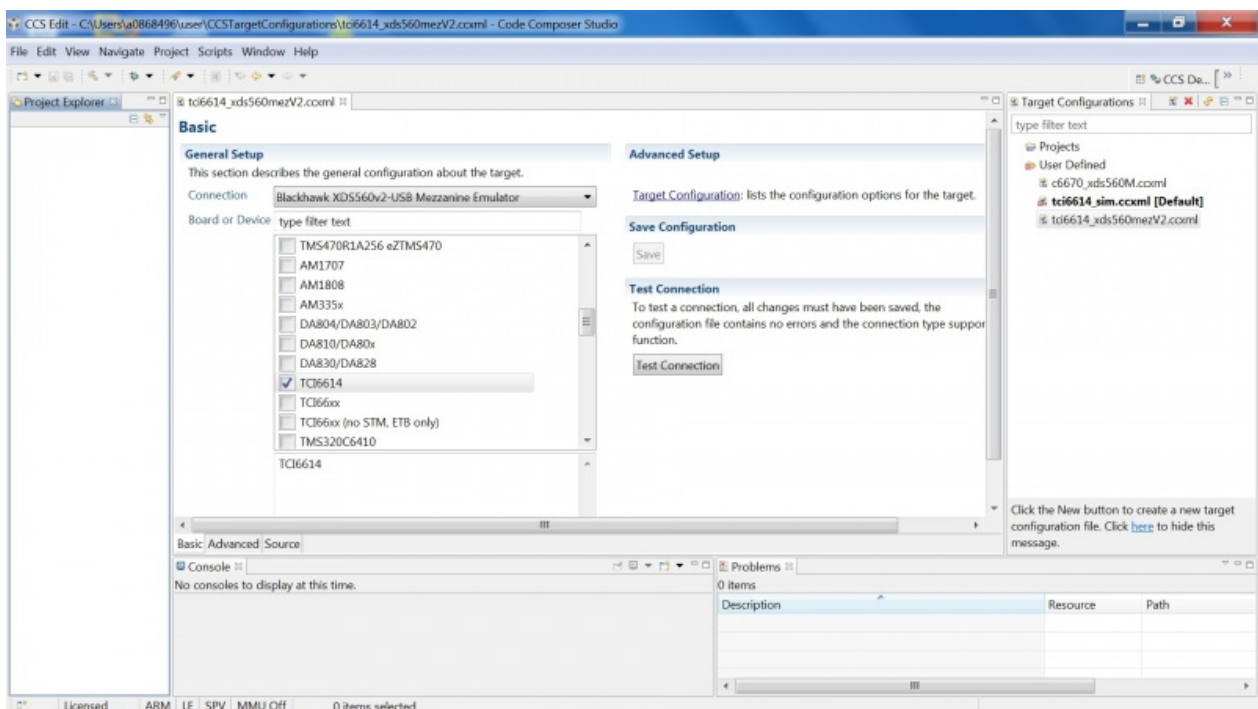
Booting Linux through CCS

- Download and install CCS v5.1 on Windows XP/7
- copy the contents of host-tools folder from release folder/sc_mcsdk_linux_<version> to the windows Host machine c:/host-tools. Copy the contents of images folder from release folder/sc_mcsdk_linux_<version> to c:/images. Edit the loadlin-evm.js file for the following:-

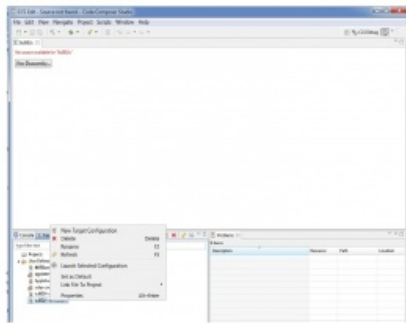
```
CFG_LOADER_BASE = "c:/host-tools/loadlin";
```

```
CFG_SRC_BASE = "c:/images";
```

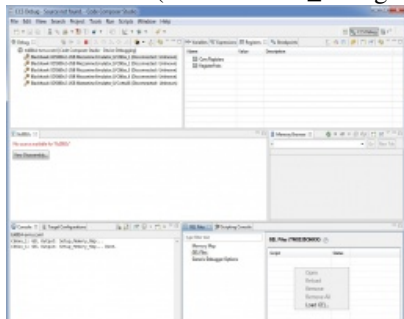
- Create a new target configuration using Blackhawk XDS560v2-USB Mezzanine Emulator for TCI6614 and save the configuration (example tci6614-evm.ccxml)



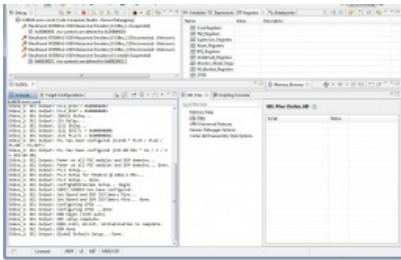
- Launch Target configuration for EVM. Click view-> Target Configuration from the top level menu. Inside the Target Configuration window, right click on to the tci6614-evm.ccxml and select "Launch Selected Configuration" as shown below:-



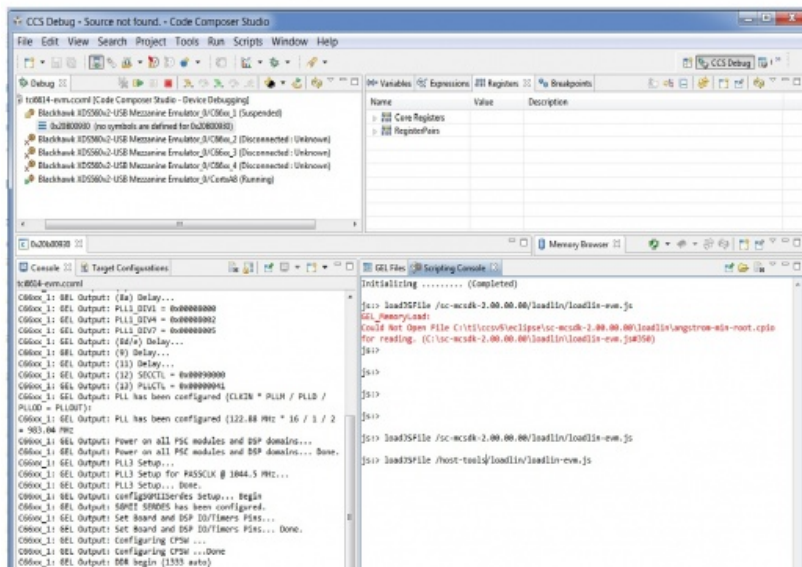
- Load Gel file (evmc66141xe_v0.5.gel from C:/host-tools as shown



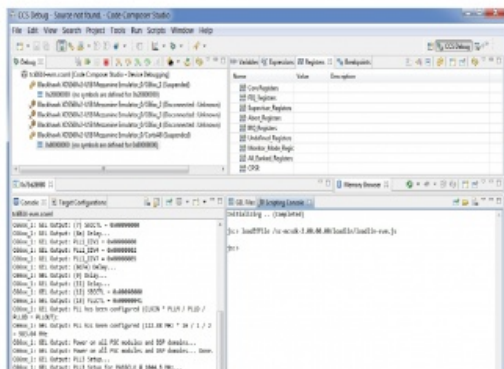
- Connect to the c66xx_1 target (DSP). The "Global_Default_Setup" will be run automatically as shown below. Connect also to CortexA8 (ARM)



- Open the scripting window. Click view->Scripting Console and run loadlin-evm.js dss script as shown



- At the end of the dss script, the ARM target will halt at address 0x80008000 as shown below



- Click on the CortexA8 emulator target and click on Resume(F8) button. At the serial port, Linux boot up log will be shown as

Booting Linux through U-Boot

Please refer the section "U-Boot TFTP Boot" for the procedure.

Booting Linux using NFS file system

To boot using nfs rootfs, add following options to the kernel command line parameter

```
root=/dev/nfs rw nfsroot=<nfs-server-ip>:/<file system path>,nolock ip=dhcp
```

Edit loadlin-evm.js and change the value of CFG_ROOTFS_NFS variable to update <nfs-server-ip> and <file system path>

```
var CFG_ROOTFS_NFS = "root=/dev/nfs rw nfsroot=<nfs-server-ip>:/<file system path>,nolock ip=dhcp"
```

If you are using the latest loadlin-evm.js, to use nfs rootfs, Comment the rootfs variable below:-

```
//var rootfs = "initrd";
```

And uncomment for nfs rootfs

```
var rootfs = "nfs";
```

Make sure the nfs server is running on the ubuntu Host machine. Refer the network driver section on how to start the nfs server on Linux Host.

Sample Linux boot up log

This is only a sample log with UBIFS rootfs. The actual log on your machine may vary from release to release.

```
TCI6614 EVM # boot
Creating 1 MTD partitions on "nand0":
0x0000000180000-0x0000008000000 : "mtd=2"
UBI: attaching mtd1 to ubi0
UBI: physical eraseblock size: 131072 bytes (128 KiB)
UBI: logical eraseblock size: 126976 bytes
UBI: smallest flash I/O unit: 2048
UBI: VID header offset: 2048 (aligned 2048)
UBI: data offset: 4096
UBI: attached mtd1 to ubi0
UBI: MTD device name: "mtd=2"
UBI: MTD device size: 126 MiB
UBI: number of good PEBs: 1008
UBI: number of bad PEBs: 4
UBI: max. allowed volumes: 128
UBI: wear-leveling threshold: 4096
UBI: number of internal volumes: 1
UBI: number of user volumes: 3
UBI: available PEBs: 0
UBI: total number of reserved PEBs: 1008
UBI: number of PEBs reserved for bad PEB handling: 10
UBI: max/mean erase counter: 2/1
```

```
UBIFS: static UBI volume - read-only mode
UBIFS: mounted UBI device 0, volume 0, name "boot"
UBIFS: mounted read-only
UBIFS: file system size:      2031616 bytes (1984 KiB, 1 MiB, 16 LEBs)
UBIFS: journal size:         1142785 bytes (1116 KiB, 1 MiB, 8 LEBs)
UBIFS: media format:         w4/r0 (latest is w4/r0)
UBIFS: default compressor: LZO
UBIFS: reserved for root: 0 bytes (0 KiB)
Loading file 'uImage' to addr 0x88000000 with size 1730200
(0x001a6698)...
Done
Loading file 'tci6614-evm.dtb' to addr 0x80000200 with size 3366
(0x00000d26
)...
Done
## Booting kernel from Legacy Image at 88000000 ...
   Image Name:   Arago/3.2/tci6614-evm
   Created:      2012-02-20   8:05:09 UTC
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    1730136 Bytes = 1.6 MiB
   Load Address: 80008000
   Entry Point:  80008000
   Verifying Checksum ... OK
## Flattened Device Tree blob at 80000200
   Booting using the fdt blob at 0x80000200
   Loading Kernel Image ... OK
OK
   Loading Device Tree to 8fec1000, end 8fec4d25 ... OK

Starting kernel ...

Uncompressing Linux... done, booting the kernel.
Linux version 3.2.0 (jenkins@ares-ubuntu.am.dhcp.ti.com) (gcc version
4.3.3
(Sourcery G++ Lite 2009q1-203) ) #1 PREEMPT Mon Feb 20 03:05:05 EST
2012
CPU: ARMv7 Processor [413fc082] revision 2 (ARMv7), cr=10c53c7d
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction
cache
Machine: TCI6614 EVM, model: Texas Instruments TCI6614 SoC
cma: CMA: reserved 16 MiB at 8e800000
Memory policy: ECC disabled, Data cache writeback
DaVinci tci6614 rev 1.0 variant 0x0
main_pll_clk rate is 983040000, postdiv = 2, pll_m = 15, plld = 0
Built 1 zonelists in Zone order, mobility grouping on. Total pages:
65024
Kernel command line: console=ttyS0,115200n8 ip=dhcp mem=256M rootwait=1
```

```

rw u
bi.mtd=2,2048 rootfstype=ubifs root=ubi0:rootfs rootflags=sync
PID hash table entries: 1024 (order: 0, 4096 bytes)
Dentry cache hash table entries: 32768 (order: 5, 131072 bytes)
Inode-cache hash table entries: 16384 (order: 4, 65536 bytes)
Memory: 256MB = 256MB total
Memory: 239800k/239800k available, 22344k reserved, 0K highmem
Virtual kernel memory layout:
    vector   : 0xfffff0000 - 0xfffff1000   (   4 kB)
    fixmap   : 0xffff00000 - 0xffffe0000   ( 896 kB)
    vmalloc   : 0xd08000000 - 0xfe6000000   ( 734 MB)
    lowmem    : 0xc00000000 - 0xd00000000   ( 256 MB)
    modules   : 0xbf0000000 - 0xc00000000   (  16 MB)
      .text    : 0xc00080000 - 0xc0324ce4    (3188 kB)
      .init    : 0xc03250000 - 0xc0342000    ( 116 kB)
      .data    : 0xc03420000 - 0xc036c3d8    ( 169 kB)
      .bss     : 0xc036c3fc0 - 0xc03884c8    ( 113 kB)
SLUB: Genslabs=11, HWalign=64, Order=0-3, MinObjects=0, CPUs=1, Nodes=1
NR_IRQS:512
IRQ: Found an omap-aintc at 0xd0800000 (revision 5.0) with 128
interrupts
IRQ: intd version 1.0 at fee10000
IRQ: cpintc version 1.0 at fee0c000
sched_clock: 32 bits at 163MHz, resolution 6ns, wraps every 26214ms
Calibrating delay loop... 978.94 BogoMIPS (lpj=4894720)
pid_max: default: 4096 minimum: 301
Mount-cache hash table entries: 512
CPU: Testing write buffer coherency: ok
hw perfevents: enabled with ARMv7 Cortex-A8 PMU driver, 5 counters
available
DaVinci: 32 gpio irqs
NET: Registered protocol family 16
DMA: preallocated 256 KiB pool for atomic coherent allocations
hw-breakpoint: debug architecture 0x4 unsupported.
bio: create slab <bio-0> at 0
keystone-hwqueue 2a00000.hwqueue: registered queues 0-4095
keystone-pktdma 2a6c000.pktdma: veth tx channel: pool pool-veth,
descs 128
, channel 0 (d0816400), prio 1, tag 10000, submit 800, complete -1
keystone-pktdma 2a6c000.pktdma: veth rx channel: pool pool-veth,
descs 128
, channel 0 (d081e800), flow 0 (d0822000), submit -1, complete -1
keystone-pktdma 2a6c000.pktdma: registered flows 32, tx chans: 32, rx
chans:
    32, loopback
keystone-pktdma 2004000.pktdma: net tx channel: pool pool-net, desc
128,

```

```
channel 0 ( (null)), prio 2, tag 0, submit 648, complete -1
keystone-pktdma 2004000.pktdma: netrx rx channel: pool pool-net, descs
128,
channel 0 ( (null)), flow 0 (fe805000), submit -1, complete 657
keystone-pktdma 2004000.pktdma: patx tx channel: pool pool-net, descs
8, cha
nnel 0 ( (null)), prio 2, tag 0, submit 640, complete -1
keystone-pktdma 2004000.pktdma: parx rx channel: pool pool-net, descs
4, cha
nnel 0 ( (null)), flow 1 (fe805020), submit -1, complete -1
keystone-pktdma 2004000.pktdma: registered flows 32, tx chans: 9, rx
chans:
24
Switching to clocksource timer0_1
NET: Registered protocol family 2
IP route cache hash table entries: 2048 (order: 1, 8192 bytes)
TCP established hash table entries: 8192 (order: 4, 65536 bytes)
TCP bind hash table entries: 8192 (order: 3, 32768 bytes)
TCP: Hash tables configured (established 8192 bind 8192)
TCP reno registered
NET: Registered protocol family 1
NetWinder Floating Point Emulator V0.97 (double precision)
JFFS2 version 2.2. (NAND) © 2001-2006 Red Hat, Inc.
Block layer SCSI generic (bsg) driver version 0.4 loaded (major 254)
io scheduler noop registered
io scheduler deadline registered
io scheduler cfq registered (default)
Serial: 8250/16550 driver, 3 ports, IRQ sharing disabled
serial8250.0: ttyS0 at MMIO 0x2540000 (irq = 448) is a 16550A
console [ttyS0] enabled
serial8250.0: ttyS1 at MMIO 0x2541000 (irq = 451) is a 16550A
loop: module loaded
at24 1-0050: 131072 byte 24c1024 EEPROM, writable, 128 bytes/write
Generic platform RAM MTD, (c) 2004 Simtec Electronics
ONFI flash detected
ONFI param page 0 valid
NAND device: Manufacturer ID: 0x2c, Chip ID: 0xa1 (Micron
MT29F1G08ABBD4HC)
Bad block table found at page 65472, version 0x01
Bad block table found at page 65408, version 0x01
Creating 3 MTD partitions on "davinci_nand.0":
0x00000000000000-0x00000001000000 : "u-boot"
0x00000001000000-0x00000001800000 : "params"
0x00000001800000-0x00000008000000 : "ubifs"
davinci_nand davinci_nand.0: controller rev. 2.5
UBI: attaching mtd2 to ubi0
UBI: physical eraseblock size: 131072 bytes (128 KiB)
```

```
UBI: logical eraseblock size:      126976 bytes
UBI: smallest flash I/O unit:      2048
UBI: sub-page size:                512
UBI: VID header offset:            2048 (aligned 2048)
UBI: data offset:                  4096
UBI: max. sequence number:         210
UBI: attached mtd2 to ubi0
UBI: MTD device name:              "ubifs"
UBI: MTD device size:              126 MiB
UBI: number of good PEBs:          1008
UBI: number of bad PEBs:           4
UBI: number of corrupted PEBs:     0
UBI: max. allowed volumes:         128
UBI: wear-leveling threshold:      4096
UBI: number of internal volumes:   1
UBI: number of user volumes:       3
UBI: available PEBs:               0
UBI: total number of reserved PEBs: 1008
UBI: number of PEBs reserved for bad PEB handling: 10
UBI: max/mean erase counter: 2/1
UBI: image sequence number: 16460284
UBI: background thread "ubi_bgt0d" started, PID 235
m25p80 spi0.0: n25q032 (4096 Kbytes)
Creating 2 MTD partitions on "m25p80":
0x00000000000000-0x00000000800000 : "u-boot-spl"
0x00000000800000-0x00000004000000 : "test"
spi_davinci spi_davinci.0: Controller at 0xd0876000
watchdog watchdog: heartbeat 60 sec
keystone-crypto 20c0000.crypto: crypto accelerator enabled
keystone-rproc 108ffffc.rproc: dsp-core0 is available
keystone-rproc 118ffffc.rproc: dsp-core1 is available
keystone-rproc 128ffffc.rproc: dsp-core2 is available
keystone-rproc 138ffffc.rproc: dsp-core3 is available
oprofile: using arm/armv7
TCP cubic registered
NET: Registered protocol family 17
VFP support v0.3: implementor 41 architecture 3 part 30 variant c rev 3
UBIFS: parse sync
UBIFS: recovery needed
UBIFS: recovery completed
UBIFS: mounted UBI device 0, volume 2, name "rootfs"
UBIFS: file system size: 111357952 bytes (108748 KiB, 106 MiB, 877
LEBs)
UBIFS: journal size: 9023488 bytes (8812 KiB, 8 MiB, 72 LEBs)
UBIFS: media format: w4/r0 (latest is w4/r0)
UBIFS: default compressor: lzo
UBIFS: reserved for root: 0 bytes (0 KiB)
```

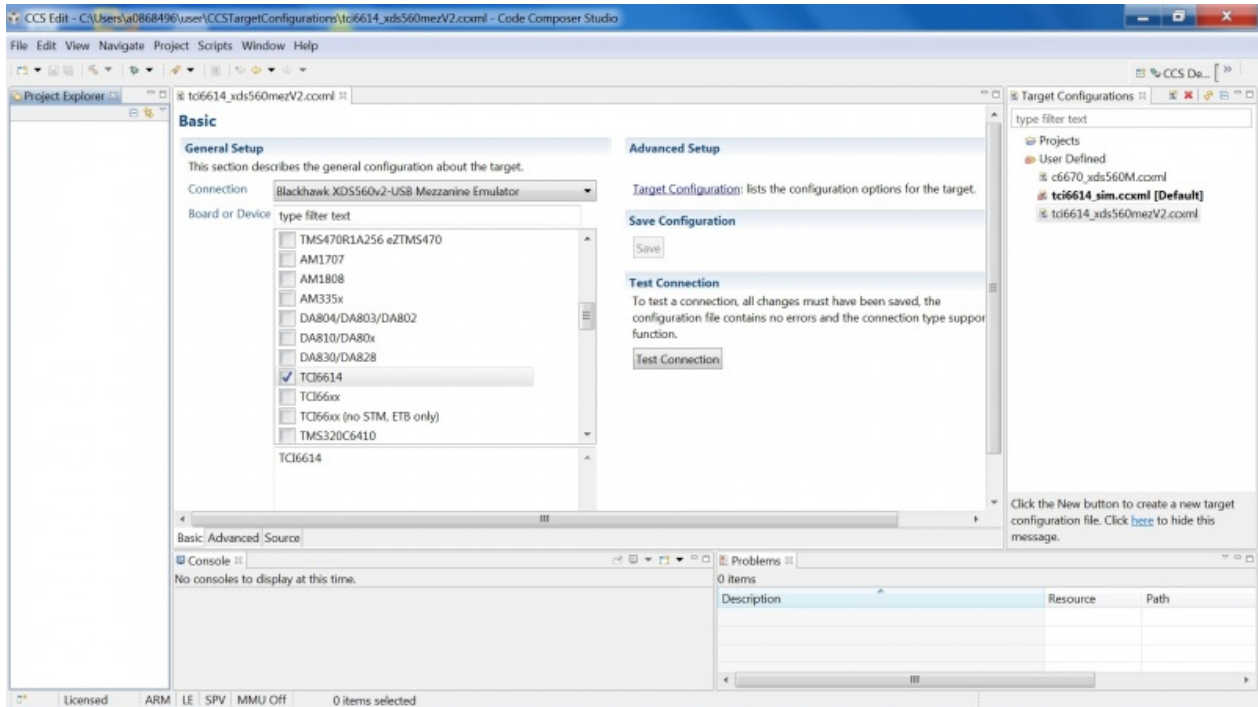


```
login[827]: root login on `ttyS0'
```

Running U-Boot on EVM

Installing Emulator

- Download and install CCS v5.1 on Ubuntu Linux 10.04 or Windows XP/7
- Create a new target configuration using Blackhawk XDS560v2-USB Mezzanine Emulator for TCI6614.



Loading and running U-Boot on EVM through CCS

Please refer to *Getting Started Guide*.

Loading and running U-Boot on EVM through Ethernet boot

A u-boot image for booting through Ethernet boot can be created by building u-boot using `tc16614_evm_min_config` and then using the host tool mentioned in the u-boot software section. This boot ROM Ethernet formatted image (u-boot-eth.bin) can be used for booting the EVM through Ethernet boot. The dip switch settings needed for Ethernet boot is provided in *Getting Started Guide*.

Connect the EVM ethernet port to the Host PC Ethernet port either directly or through a switch. Run a tftp server utility software such as the free tftp32 at <http://tftpd32.jounin.net/>. Configure the tftp server software to run DHCP server as well. Configure the bootfile as u-boot-eth.bin, IP address and Mask. start the server. Connect the UART port of the PC to the EVM's UART port. Power on the EVM. The u-boot boot up log will be shown on the UART console.

Loading and running Linux through NAND Boot on EVM

U-Boot supports NAND boot with Linux Kernel and filesystem. In the NAND boot mode, U-Boot will boot the kernel image from the NAND. The kernel will boot using the file system flashed on the NAND. Note that you may need to change the U-Boot bootargs parameter based on your boot settings. Below are the steps to test the NAND boot:

- Copy the RBL compliant u-boot image (u-boot_hdr_tlr.bin), kernel image (uImage) and root filesystem (tc16614-evm-ubifs.ubi) to the TFTP server root directory
- Connect the EVM to the lab network or any private network that has DHCP/TFTP server running
- Set the boot dip switches to ARM no boot mode
- Run the U-Boot using CCS, and get to the U-Boot command prompt
- Write the U-Boot, kernel and filesystem images to NAND

You may use the following commands as an example:

- Setup the TFTP server IP address in u-boot

```
>setenv serverip <Server IP>
```

Where <Server IP> is IP address of the TFTP server

- Setup mtdparts variable to define partition info in sync with the Linux kernel partition. This will allow setenv commands

to use the partition name instead of offset in the NAND.

```
>setenv mtdparts
```

```
mtdparts=davinci_nand.0:1024k (bootloader) , 512k (params) ro, 129536k (ubifs)
```

```
>saveenv
```

- TFTP u-boot images to load address 0x88000000 and burn to the flash:

```
>setenv bootfile u-boot_hdr_tlr.bin
```

```
>mw.b 0x88000000 0xFF 0x80000
```

```
>dhcp
```

```
>nand erase.part bootloader
```

```
>nand write 0x88000000 bootloader <size of u-boot_hdr_tlr.bin>
```

```
>oob fmt 0 <size of u-boot_hdr_tlr.bin>
```

Where size is u-boot_hdr_tlr.bin file size aligned to NAND page size.

The u-

boot-nand.bin file size in Hex is printed at the end of dhcp or tftp by the boot loader.



Note: U-Boot image needs to be converted to an RBL compliant bootable image with the header and trailer added and ECC layout re-formatted. Please read this section further in this document **U-Boot Utilities:Utility to add header and trailer**

- TFTP rootfs (tc16614-evm-ubifs.ubi) image to load address 0x88000000 and burn to the flash:

UBI maintains an erase counter to do wear levelling. So the procedure to flash the rootfs into NAND is different for very first time and there after. First time NAND is erased using nand erase command. Subsequently if any of the volume is to be updated, it can be just removed and recreated.

1) Very first time

```
>setenv bootfile tci6614-evm-ubifs.ubi
>setenv bootargs 'console=ttyS0,115200n8 ip=dhcp mem=512M
rootwait=1
    rootfstype=ubifs root=ubi0:rootfs rootflags=sync rw ubi.mtd=2,2048'
>setenv bootcmd 'ubi part ubifs; ubifsmount boot; ubifsload
0x88000000 uImage;
    ubifsload 0x80000200
tci6614-evm.dtb; bootm 0x88000000 - 0x80000200'
>saveenv
>nand erase.part ubifs
>dhcp
>nand write 0x88000000 ubifs <size of ubifs.ubi in Hex. ie. with
0x prefix. Example 0x2600000>
>boot
```

In order to allow power cycling the board without corrupting the UBIFS file system, following procedure to be followed very first time:-

- After flashing the UBI image on to NAND, Boot up Linux and login
- Issue a 'reboot' command from the Linux shell and wait for EVM to reboot to u-boot prompt.

Now user will be able to power cycle the board without corrupting the file system.

2) If there is existing ubi volumes, (boot, rootfs, rootfs-recovery) To update boot volume

```
>setenv bootfile tci6614-boot.ubifs.img
>dhcp
>ubi part ubifs
>ubi remove boot
>ubi create boot <size>
- where size is actual size of tci6614-boot.ubifs.img in hex i.e
with 0x prefix
>ubi write 0x88000000 boot <size>
- where size is actual size of tci6614-boot.ubifs.img in hex i.e with
0x prefix
```

For rootfs and rootfs-recovery, use the same steps as above.

- You also need to set the dip switch to ROM NAND boot mode on the EVM and do POR. please refer to Hardware Setup Steps section in TMDXEVM6614LXE EVM Hardware Setup Guide on how to set the dip switch to ARM Master NAND boot mode.



Note:

1. There is a problem for the ROM boot loader to boot from NAND with PLL configuration set to 122.88MHz input frequency because the wait time for the NAND device ready is too short. As a workaround the PLL configuration is set to 312.50MHz

For boot using NFS file system, add "root=/dev/nfs rw nfsroot=<nfs-server-ip>:<file system path>,nolock ip=dhcp" to the bootargs environment variable.

E.g.: setenv bootargs 'console=ttyS0,115200n8 root=/dev/nfs rw
 nfsroot=158.218.103.87:/nfs_path,nolock ip=dhcp'

Loading and running Linux through TFTP Boot on EVM

U-Boot supports TFTP boot with Linux Kernel and filesystem. In the TFTP boot, U-Boot will tftp the kernel image and device tree blob from the TFTP server and boot the kernel. The kernel will boot with the file system flashed on the NAND. Note that you may need to change the U-Boot bootargs parameter based on your boot settings. It is assumed that the u-boot is programmed in the NAND as per steps in previous section. Below are the steps to test the TFTP boot:

- Copy the kernel image (uImage), device tree blob, tci6614-evm.dtb, and root filesystem (tci6614-evm-ubifs.ubi) to the TFTP server root directory
- Connect the EVM to the lab network or any private network that has DHCP/TFTP server running
- Set the boot dip switches to NAND boot mode. Please refer to Hardware Setup Steps section in TMDXEVM6614LXE EVM Hardware Setup Guide on how to set the dip switch to NAND boot mode.
- Power up EVM and press Esc to get u-boot prompt
- tftp filesystem image to DDR and program the ubifs NAND partition using the image

You may use the following commands as an example:

- Setup the TFTP Server IP in U-Boot

```
>setenv serverip <TFTP Server IP>
```

- Setup mtdparts variable in U-Boot with partition info that match with that defined in Linux kernel.

```
>setenv mtdparts
mtdparts=davinci_nand.0:1024k (bootloader) , 512k (params) ro, 129536k (ubifs)
```

- TFTP rootfs(tci6614-evm-ubifs.ubi) image from the server to load address 0x88000000 and program the ubifs partition.

See previous section for the steps for doing this.

- TFTP uImage image from the server to load address 0x88000000 and tci6614-evm.dtb to load address 0x80000200.

```
>setenv bootfile uImage
>dhcp
>tftp 0x80000200 tci6614-evm.dtb
>setenv bootcmd 'dhcp; bootm 0x88000000 - 0x80000200'
>boot
```

Loading and running file system using initramfs file system option

Loading and running the linux kernel and file system from NAND boot (using UBI images) increases an application's latency as the file system is present in the NAND. Some scenarios may require you to load the file system onto the RAM instead of being accessed from the NAND to improve latency times for an application. This section discusses the steps to creating an init ramfs based file system, loading it as part of a UBI volume and modifying the boot arguments to load this initramfs file system to RAM during boot up of the EVM.

- Loading an initramfs based file system

The standard release software for SC-MCSDK-2.0 contains a init ramfs type file. The file ti-scmsdk-rootfs-tci6614-evm.cpio is located in sc_mcsdk_linux_2_00_00_xx/images/. It can be used for a filesystem for the tci6614 evm. To use this cpio.gz for your filesystem from RAM, do the following:

From u-boot prompt, do:

```
setenv bootargs 'root=/dev/ram0 rw console=ttyS0,115200n8
initrd=0x83000000,120M rdinit=/sbin/init'
tftp 0x83000000 ti-scmcsdk-rootfs-tci6614-evm.cpio
boot
```

To create a initramfs based filesystem from your existing filesystem (because the user has custom contents in the file system), use the following commands:

```
( cd <directory-to-file-system> ; find . | cpio -o -H newc |
gzip ) > fs.gz
```

Then follow the note above to load this file system to DDR and change bootargs appropriately.

To load the initramfs as a separate volume into the UBI based image, do the following:

```
mkdir <work-dir>/boot (work-dir is directory where you want to
generate these images)
cd <work-dir>/boot
Copy kernel, device tree and initramfs(fs.gz that was
generated/cpio.gz that is part of the release images) images to current
directory.
cd <work-dir>/boot
cd <work-dir>
mkfs.ubifs -r ./boot -F -o ./tci6614-boot.ubifs.img -m 2048 -e 126976
-c 898
ubinize -o ubifs.ubi -m 2048 -p 128KiB -s 2048 -O 2048 ubinize.cfg
```

- Note: mkfs.ubifs and ubinize binaries are provided with the standard release.

The ubifs.ubi image is a unified image that has the kernel, device tree and the init ramfs file system images inside the “boot” volume of the UBI image.

For the ubinize command to work, there needs to be a “ubinize.cfg” file created with the contents looking like below:

```
[boot]
mode=ubi
image=tci6614-boot.ubifs.img
vol_type=dynamic
vol_id=0
vol_name=boot
```

- U-boot environment setup for booting an initramfs file system

The following parameters are setup via u-boot to boot up a initramfs file system:

```
setenv bootcmd 'ubi part ubifs; ubifsmount boot; run load_kernel
load_dtb load_fs boot_kernel'
setenv bootargs 'root=/dev/ram0 rw console=ttyS0,115200n8
initrd=0x83000000,120M rdinit=/sbin/init'
setenv boot_kernel 'bootm 0x88000000 - 0x80000200'
setenv load_dtb 'ubifsload 0x80000200 tci6614-evm.dtb'
setenv load_fs 'ubifsload 0x83000000 fs.gz'
setenv load_kernel 'ubifsload 0x88000000 uImage'
```

- Note:* The 120M value in initrd (part of bootargs command above) needs to be re-adjusted as per your needs. Please remember that the NAND's limit is 128M and the file system cannot be more than this size (leaving room for kernel and dtb file)

After setting up the environment like above - please re-flash to NAND the ubifs.ubi that was generated.

Setting up a static ip address from U-Boot

To setup a static IP we need to appropriately edit the kernel command line A good write-up can be found at Static IP Setup ^[15]

Basically a user needs to fill up the fields depicted below in the command line

```
ip=${ipaddr}:${serverip}:${gatewayip}:${netmask}:${hostname}::off
```

If NFS is being used as the filesystem then an **example** bootargs can be setup as shown below

```
setenv bootargs 'earlyprintk debug console=ttyS0,115200n8 mem=512M rootwait=1 rootfstype=nfs root=/dev/nfs rw
nfsroot=158.218.103.29:/opt/arago-iperf,v3,tcp ip=158.218.103.199::158.218.102.2:255.255.254.0::eth0:'
```

where

static ip is **158.218.103.199**, gateway is **158.218.102.2**, mask is **255.255.254.0**

If UBIFS is being used as the filesystem then an **example** bootargs can be setup as shown below

```
setenv bootargs 'console=ttyS0,115200n8 ip=10.218.107.199::10.218.107.2:255.255.255.0::eth0: rootflags=sync
mem=512M rootwait=1 rw ubi.mtd=2,2048 rootfstype=ubifs root=ubi0:rootfs'
```

where

static ip address is **10.218.107.199**, gateway is **10.218.107.2**, mask is **255.255.255.0**

Linux-devkit usage

The linux-devkit can be used to build ARM/Linux applications outside Yocto framework. The devkit provides necessary header files and libraries for application to compile. This can be used during application's initial bringup and debugging phases.

The application must subsequently have its own bitbake recipe and build/install using Yocto framework in the Linux filesystem/kernel.

Location of the devkit

The devkit is in `sc_mcsdk_linux_2_00_00_##linux-devkit`

The shell script (`arago-2013.04-armv7a-linux-gnueabi-sc-mcsdk-sdk-i686.sh`) needs to be run for the devkit to install itself.

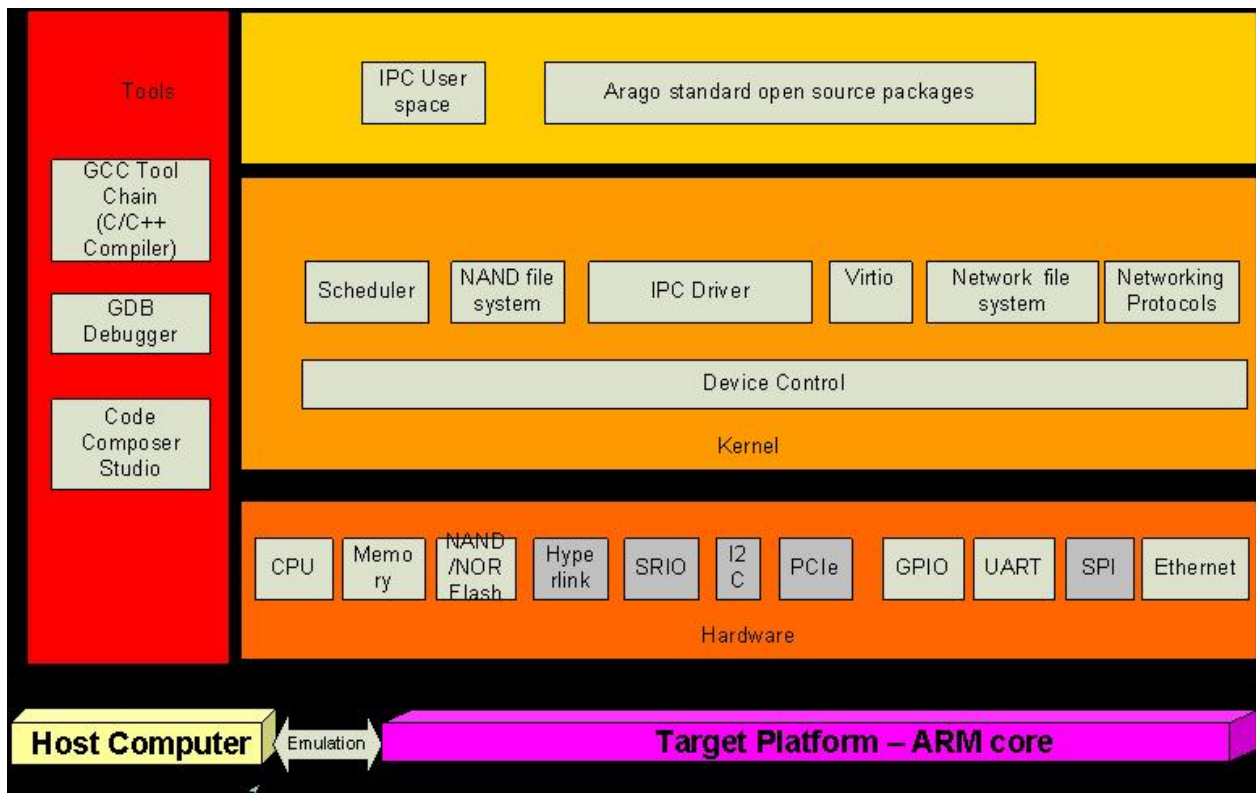
Linux Software Overview

Introduction

Linux software port described here is based on upstream Linux kernel releases. The base line for the release is Linux 3.0 and the patches developed to support the target platforms are up ported to a stable Linux release as and when it become available. Linux port re-uses the davinci architecture as many of the davinci specific IPs are re-used in this SoC. This section describes the various components and identify the components modified to support the target platforms. The figure below shows the architecure of Linux eco system.

The software components falls into two categories.

- Base port
- Device Drivers



Base Port

The base port is based on DaVinci architecture. The base port re-uses the following existing software components available in DaVinci architecture.

- DaVinci common infrastructure
- Clock infrastructure
- DaVinci Timer subsystem
- Serial Port initialization
- AEMIF
- NAND Flash
- GPIO
- SPI
- I2C
- Watchdog

Following drivers from upstream kernel are re-used

- EEPROM
- SPI NOR Flash

Specific changes for support TCI6614 target platforms:-

- Platform initialization
- Board and device initialization
- Interrupt controllers

Platform initialization

TCI6614 re-uses the DaVinci common initialization code for platform initialization. For the location of files mentioned in this section, please refer the porting guide (TBD).

DaVinci common initialization

DaVinci arch specific code common initialization code is located in common.c. TCI6614 platform initialization code customize the SoC specific parameters in tci6614_soc_info which is of type davinci_soc_info. The board specific initialization code calls tci6614_init() function to initialize the SoC. This in turn calls davinci_common_init() passing the SoC specific info to initialize the SoC (a.k.a platform).

DaVinci common code does following:-

- 1) `iotable_init()` - create architecture specific IO mapping using the `io_desc` provided in the `tci6614_soc_info`
- 2) `local_flush_tlb_all()` - flushes TLB entries
- 3) `flush_cache_all()` - flush the cache entries
- 4) `davinci_init_id()` - reads `jtag_id` info from the SoC and validate it againsts the ids provided the `soc_info` structure. Once validation succeeds,
 it fills the `cpu_id` info in the `soc_info`.
- 5) `davinci_clk_init()` - TCI6614 platform code initializes the SoC specific clock info in the `cpu_clks` field of `tci6614_soc_info` structure. This has
 a clock lookup table for all of the clocks managed by the DaVinci clock infrastructure. Device drivers uses `clock_get()` API call to acquire a clock
 and enable/disable the same as needed. The DaVinci clock infrastructure code is in `clock.c`. It implements a clock tree based on the clock look up
 table and configures the various clock deviders in the SoC. It has built in functions for calculating the clock rate for individual clocks based
 on the parent clock rate and the divider. The default clock rate calculate functions can be overridden by the platform. TCI6614 platform code
 provides two functions for calculating the reference clock rate input to the SoC.

 - `tci6614_ref_clk_recalc()` - Reads the DEVSTAT register to read the input clock rate to the Main PLL. There is a lookup table provided to read the
 value of input clock rate based on the DEVSTAT settings. The u-boot program the DEVSTAT based on the Main PLL clock used in the board.


```
- tci6614_main_pll_clk_recalc() - reads the PLLM, PLLD and Main PLL
clock rate (parent clock or reference clock rate) to calculate the
System clock
    rate.
```

The individual clock rates are calculated by the infrastructure code using the PLL divider value (read from the PLLDIVX register) applicable for the specific clock output.

clock_enable() and clock_disable() functions in clock.c calls davinci_psc_config() to enable or disable the clock gate for an IP. It also turns on the Power Domain as applicable.

Board and device initialization

TCI6614 EVM (board-tci6614-evm.c) and Simulator (board-tci6614-sim.c) specific code is responsible for the board or Simulator specific initialization. Note that Only basic drivers such as UART, timers etc are initialized in Simulator. ARM architecture provides the macro MACHINE_START() to define machine or board specific features.

Following are defined for TCI6614 EVM or Simulator

- Machine type and Name
- boot_params - Boot parameters - Array of TAGs.
- map_io() - initialize the platform and IO map by calling tci6614_init()
- init_irq() - initialize the interrupt controller (described later in this section)
- timer - initialized to davinci_timer. The board re-uses the DaVinci timer sub system (time.c) for initializing the the event and clock source timers (described later)
- init_machine() - Board specific initialization code

init_machine() calls following:-

```
tci6614_devices_init() - implemented in devices-tci6614.c
    Initializes following platform devices
        - Re-uses the davinci_serial_init() to do platform specific
initialization for the UART and register the platform device for serial
ports
        - Registers the watchdog device, nand, i2c master, spi master,
netcp devices etc.
    i2c_register_board_info()
        - Register the i2c slave devices in the board (EEPROM)
    spi_register_board_info()
        - Registers the SPI slave devices in the board (SPI NOR Flash)
```

DaVinci common platform code re-used

DaVinci AEMIF driver

Re-uses AEMIF driver from DaVinci (aemif.c). NAND is connected to the AEMIF. DaVinci NAND driver calls the `davinci_aemif_setup_timing()` to configure the timings for NAND. This function converts the various setup time values to AEMIF clock rate to program the AEMIF.

DaVinci Serial driver

`davinci_serial_init()` in `serial.c` initializes the DaVinci serial port specific hardware and registers the UART devices configured in the board specific code.

Davinci Timer subsystem

DaVinci Timer subsystem consists of initialization of two of the main timers in the Linux kernel

- Clockevent timer
- Clocksource timer

The DaVinci timer subsystem is implemented in `timer.c`. As explained earlier, the `MACHINE_START` macro defined in the board specific code provides callback functions to the Kernel to initialize the timer at boot up. The specific function implemented by DaVinci sub system is `davinci_timer_init()`. `TIMER7` is a 64 bit timer available in the SoC which is configured as two independent 32 bit timers :- one each for Clockevent and Clocksource. Clockevent timer generates interrupts at HZ rate and Clocksource runs a monotonous timeline clock for kernel. The DaVinci timer sub system calls `clocksource_register_hz()` to registers the clocksource timer and `clockevents_register_device()` register the clockevent timer. DaVinci timer also override the weak `sched_clock()` with a high resolution clock value which is used by the scheduler. Clockevent timer is configured as a single shot timer and Clocksource timer is a free running timer. Since the `sched_clock()` requires a monotonous timer and the clocksource timer wraps around quickly, the DaVinci timer sub system re-uses the `sched_clock` infrastructure in ARM kernel (`arch/arm/sched_clock.c`) to implement a 64 bit timer (non-wrapping) using the clocksource timer. See calls to `init_sched_clock()` from `davinci_timer_init()`.

Interrupt controllers

Fundamentally, there are three interrupt controllers in TCI6614:

Main ARM Interrupt Controller

The main interrupt controller for ARM is the OMAP style AINTC controller, which is initialized by `omap_aintc_init()`. The output interrupt lines from this controller are tied to the ARM CPU's IRQ and FIQ inputs. The interrupt dispatch code for this controller is in `arch/arm/mach-davinci/include/mach/entry-macro.S`. The dispatch code (written in assembly) determines the IRQ number and dispatches the IRQ via the kernel interrupt infrastructure. The call flow in this scenario is as follows:

1. CPU gets an interrupt and calls the vector
(see `__vectors_start` in `entry-armv.S`)
... skipped irrelevant stuff here...
2. Dispatch calls `get_irqnr_and_base`
(see `arch/arm/mach-davinci/include/mach/entry-macro.S`)
This assembly fragment reads OMAP AINTC registers to determine the irq number to dispatch
3. Dispatch calls:

```
--> asm_do_IRQ() --> handle_IRQ() -->
generic_handle_irq()
--> generic_handle_irq_desc()
```

4. `generic_handle_irq_desc()` calls `desc->handle_irq()`, which is a function pointer to `handle_level_irq()`. This handler is setup by the OMAP AINTC initialization code as follows:

```
> irq_set_chip_and_handler(i, &omap_irq_chip,
handle_level_irq);
```

5. `handle_level_irq()` masks and acks the IRQ before calling the action handler, and finally unmask the IRQ on return. The action handler here is the handler passed into `request_irq()` in the case of non-threaded IRQs. In the case of threaded IRQs, the flow is much more complicated, and involves the scheduling of an IRQ thread for dispatch.

CPINTD Interrupt Controller

Some of the interrupts coming in from on-chip peripherals are pulse interrupts. The OMAP AINTC can only handle level interrupt inputs. Therefore, we have a CPINTD module on chip for pulse to level conversion. In this scenario, the on-chip peripheral's interrupt is first processed by CPINTD, which does the necessary pulse to level conversion, and then passes on a level interrupt to the OMAP AINTC interrupt controller. The CPINTD does not do interrupt aggregation, i.e., there is a one-to-one relationship between CPINTD interrupt and a corresponding OMAP AINTC interrupt.

CPINTD interrupts in Linux are handled via IRQ chaining. The ARM AINTC interrupt is first handled as before, and dispatched through the kernel.

Subsequently:

1. The kernel dispatcher ends up calling the chained IRQ handler (`intd_irq_handler`) instead of `handle_level_irq()` in step 4 of the ARM AINTC dispatch sequence above. This is setup by the CPINTD initialization code as follows:

```
> irq_set_chained_handler(i, intd_irq_handler)
```

2. The chained IRQ handler handles masking and interrupt acknowledgment of the parent (OMAP AINTC) interrupt. In addition, it maps from OMAP AINTC interrupt number to the CPINTD interrupt number (see `intc_to_intd_map[]`)
3. The chained IRQ handler dispatches the mapped CPINTD interrupt using the kernel dispatcher mechanisms as before, by calling `generic_handle_irq()`
4. The rest of the flow beyond `generic_handle_irq()` is as before (see step 3 of the OMAP AINTC interrupt dispatch flow)

CPINTC Interrupt Controller

Appleton has a large number of on-chip peripherals, and therefore needs a large number of interrupts - much more than are supported by the OMAP AINTC controller. Consequently, a number of (mostly lower priority interrupts) have been connected via the CPINTC module, which combines multiple input interrupts into fewer output interrupts. CPINTC capabilities are very thoroughly documented, and I am not going into details here.

CPINTC interrupts in Linux are again handled via IRQ chaining. The ARM AINTC interrupt is first handled as before, and dispatched through the kernel. Some of the ARM AINTC input interrupts are actually outputs from CPINTC (IRQ_TCI6614_INTC3_OUT0 ... IRQ_TCI6614_INTC3_OUT32). The dispatch flow for these interrupts are similar to the CPINTD case described above:

1. The kernel dispatcher ends up calling the chained IRQ handler (cpintc_irq_handler) instead of handle_level_irq() in step 4 of the ARM AINTC dispatch sequence above. This is setup by the CPINTC initialization code as follows:


```
> irq_set_chained_handler(i, cpintc_irq_handler)
```
2. The chained IRQ handler handles masking and interrupt acknowledgment of the parent (OMAP AINTC) interrupt. In addition, it reads the CPINTC prioritized global index register to determine the CPINTC interrupt number to dispatch.
3. The chained IRQ handler dispatches the mapped CPINTC interrupt using the kernel dispatcher mechanisms as before, by calling generic_handle_irq()
4. The rest of the flow beyond generic_handle_irq() is as before (see step 3 of the OMAP AINTC interrupt dispatch flow)

Flattened Device Tree

Flattened Device Tree (FDT) is a framework in Linux kernel by which machine specific resource information such as bootargs, memory range, resources for devices such as register address space, IRQ, platform data etc can be defined outside the code in a configuration tree and pass a configuration blob to kernel during boot up. This is traditionally defined in a board-<platform>.c and devices-<platform>.c. FDT will allow users to define common code for all variants of a SoC in a single platform or SoC specific code and thus move customization of drivers to device tree. FDT can thus allow easy customization of machine variants without changing the code. FDT replace traditional ATAGs based boot up by passing address of the device tree blob in memory (In ARM, passed in register R2 instead of the ATAG address) to kernel during bootup. The Device Tree Specification (.dts) file is compiled in to a device tree blob (dtb) by the Device Tree compiler (dtc) available under scripts/dtc directory. This will get compiled if CONFIG_DTC option is selected.

For ARM, few platforms already implemented device tree support and the dts files for these are available under arch/arm/boot/dts. The device tree dts file is tci6614-evm.dts

To build tci6614-evm.dtb, user has to do following:-

```
make uImage
make tci6614-evm.dtb
```

The tci6614-evm.dtb file will be created under arch/arm/boot directory.

Following steps required to migrate a driver to device tree model

- 1) Defined following device tree bindings in tci6614-evm.dts. See example bindings under Documentation/devicetree/bindings
- 2) Remove platform_device_register() and associated code from devices-tci6614.c or board-tci6614-evm.c. Devices tree sub system

will parse the device tree bindings and populate the nodes and their properties. The specific device node pointer to access the device tree properties are passed to the device driver through the pdev->dev.of_node pointer. Device driver probe() can now access these nodes using API calls provided by the device tree. Following are examples of API calls

- * of_property_read_u32() - read a u32 property value
- * of_property_read_u32_array() - read an array of u32 values

Check include/linux/of.h for more API calls to process the device tree nodes.

- 3) Update the probe() function of your driver

- * There is no platform_data pointer available anymore. All of the data should appear as device tree binding properties inside the device node.
- * Use the above API calls to get the device property values. A reg entry value can be obtained and iomapped using the API, of_iomap(). It accepts an index. So any value in that index will be iomapped using this API. See keystone_hwqueue.c as an example.

Device Tree procs

The device tree bindings are exported through procs. To see the value of a property use the following command:-

```
>cat /proc/device-tree/chosen/bootargs
earlyprintk debug console=ttyS0,115200n8 ip=dhcp mem=512M rootwait=1
rootfstype=nfs root=/dev/nfs rw nfsroot=192.168.1.2:/opt/arago-min-
root,nolock
```

Some of the entries are not displayable using the above command. In that case the user has to pipe the output of cat command to hexdump utility to display the value in hex.

```
>cat /proc/device-tree/soc6614@2000000/hwqueue@2a00000/link-index |
hexdump -C
00000000 00 00 14 00 00 00 04 00
```

Compiling DTS file from Target

There is a dtc compiler available on the target file system that can be used to compile a DTS file to generate DTB file. Here are the steps:-

TFTP the tci6614-evm.dts and skeleton.dtsi to the target file system

```
# tftp -g -r tci6614-evm.dts <tftp server IP>
# tftp -g -r skeleton.dtsi <tftp server IP>
```

Compile the DTS file into DTB file as

```
# dtc -I dts -O dtb -o tci6614-evm.dtb tci6614-evm.dts
DTC: dts->dtb on file "tci6614-evm.dts"
```

The compiled DTB file (tci6614-evm.dtb) may be copied to the boot volume and EVM may be rebooted using the new DTB file. Check the UBIFS section on how to update the DTB file in boot volume

UBI/UBIFS

UBI

UBI (Latin: "where?") stands for "Unsorted Block Images". It is a volume management system for raw flash devices which manages multiple logical volumes on a single physical flash device and spreads the I/O load (i.e., wear-leveling) across whole flash chip.

In a sense, UBI may be compared to the Logical Volume Manager (LVM). Whereas LVM maps logical sectors to physical sectors, UBI maps logical eraseblocks to physical eraseblocks. But besides the mapping, UBI implements global wear-leveling and transparent I/O errors handling.

An UBI volume is a set of consecutive logical eraseblocks (LEBs). Each logical eraseblock may be mapped to any physical eraseblock (PEB). This mapping is managed by UBI, it is hidden from users and it is the base mechanism to provide global wear-leveling (along with per-physical eraseblock erase counters and the ability to transparently move data from more worn-out physical eraseblocks to less worn-out ones).

UBI volume size is specified when the volume is created and may later be changed (volumes are dynamically re-sizable). There are user-space tools which may be used to manipulate UBI volumes.

There are 2 types of UBI volumes - dynamic volumes and static volumes. Static volumes are read-only and their contents are protected by CRC-32 checksums, while dynamic volumes are read-write and the upper layers (e.g., a file-system) are responsible for ensuring data integrity.

UBI is aware of bad eraseblocks (e.g., NAND flash may have them) and frees the upper layers from any bad block handling. UBI has a pool of reserved physical eraseblocks, and when a physical eraseblock becomes bad, it transparently substitutes it with a good physical eraseblock. UBI moves good data from the newly appeared bad physical eraseblocks to good ones. The result is that users of UBI volumes do not notice I/O errors as UBI takes care of them.

NAND flashes may have bit-flips which occur on read and write operations. Bit-flips are corrected by ECC checksums, but they may accumulate over time and cause data loss. UBI handles this by moving data from physical eraseblocks which have bit-flips to other physical eraseblocks. This process is called scrubbing. Scrubbing is done transparently in background and is hidden from upper layers.

Here is a short list of the main UBI features:

- UBI provides volumes which may be dynamically created, removed, or re-sized;
- UBI implements wear-leveling across whole flash device (i.e., you may continuously write/erase only one logical eraseblock of an UBI volume, but UBI will spread this to all physical eraseblocks of the flash chip);
- UBI transparently handles bad physical eraseblocks;
- UBI minimizes chances to lose data by means of scrubbing.

For more information on UBI, refer [[16][UBI]]

UBIFS

UBIFS may be considered as the next generation of the JFFS2 file-system.

JFFS2 file system works on top of MTD devices, but UBIFS works on top of UBI volumes and cannot operate on top of MTD devices. In other words, there are 3 subsystems involved:

MTD subsystem, which provides uniform interface to access flash chips. MTD provides an notion of MTD devices (e.g., /dev/mtd0) which basically represents raw flash UBI subsystem, which is a wear-leveling and volume management system for flash devices. UBI works on top of MTD devices and provides a notion of UBI volumes; UBI volumes are higher level entities than MTD devices and they are devoid of many unpleasant issues MTD devices have (e.g., wearing and bad blocks); For more information on MTD, refer [[17][MTD]]

For more information on UBIFS, refer [[18][UBIFS]]

UBIFS User-space tools

UBI user-space tools, as well as other MTD user-space tools, are available from the the following git repository: [git://git.infradead.org/mtd-utils.git](https://git.infradead.org/mtd-utils.git)

The repository contains the following UBI tools:

- ubinfo - provides information about UBI devices and volumes found in the system
- ubiattach - attaches MTD devices (which describe raw flash) to UBI and creates corresponding UBI devices
- ubidetach - detaches MTD devices from UBI devices (the opposite to what ubiattach does)
- ubimkvol - creates UBI volumes on UBI devices
- ubirmvol - removes UBI volumes from UBI devices
- ubiupdatevol - updates UBI volumes. This tool uses the UBI volume update feature which leaves the volume in "corrupted" state if the update was interrupted; additionally, this tool may be used to wipe out UBI volumes.
- ubirc32 - calculates CRC-32 checksum of a file with the same initial seed as UBI would use
- ubinize - generates UBI images
- ubiformat - formats empty flash, erases flash and preserves erase counters, flashes UBI images to MTD devices
- mtdinfo - reports information about MTD devices found in the system.

All UBI tools support "-h" option and print sufficient usage information.

NAND Layout

The NAND flash in the EVM contains three partitions:-

- bootloader - Contains u-boot
- params - contains env variables
- ubifs - contains following UBI volumes:-
 - boot volume - contains Kernel image (uImage), device tree blob etc,
 - rootfs volume - contains the rootfs which is the primary filesystem
 - rootfs-recovery volume - contains back up rootfs and can be used for booting Linux if primary volume is corrupted and not usable.

Compiling UBIFS Tools

The MTD and UBI user-space tools are available from the the following git repository:
[git://git.infradead.org/mtd-utils.git](https://git.infradead.org/mtd-utils.git)

Suggest using 1.4.8 of the mtd-utils

For instructions on compiling MTD-utils, refer [[19][MTD-Utils Compilation]]. In the instruction for building mtd-utils, please replace PREFIX with INSTALL_DIR. The makefile doesn't like the use of PREFIX variable and result in build error. This is a work around to fix the build error.

Creating UBIFS file system

For information on how to create a UBIFS image. refer [[20]]

- mkfs.ubifs

```
mtd-utils# mkfs.ubifs -r filesystem/ -F -o ubifs.img -m 2048 -e 126976
-c 936
```

Where

- -m 2KiB (or 2048). The minimum I/O size of the underlying UBI and MTD devices. In our case, we are running the flash with no sub-page writes, so this is a 2KiB page.
- -e 124KiB (or 126976) Erase Block Size: UBI requires 2 minimum I/O units out of each Physical Erase Block (PEB) for overhead: 1 for maintaining erase count information, and 1 for maintaining the Volume ID information. The PEB size for the Micron flash is 128KiB, so this leads to each Logical Erase Block (LEB) having 124KiB available for data.
- -c 936 The maximum size, in LEBs, of this file system. See calculation below for how this number is determined.
- -r filesystem. Use the contents of the 'filesystem/' directory to generate the initial file system image.
- -o ubifs.img Output file.

-F option is required when creating ubifs image. If this option is not used, Kernel may crash while loading the Filesystem from UBI partition.

The output of the above command, ubifs.img is fed into the 'ubinize' program to wrap it into a UBI image.

Creating UBI image

The images produced by mkfs.ubifs must be further fed to the ubinize tool to create a UBI image which must be put to the raw flash to be used a UBI partition.

- Create ubinize.cfg file and write the contents into it

```
mtd-utils# vi ubinize.cfg
[ubifs_rootfs_volume] <== Section header
mode=ubi <== Volume mode (other option is static)
image=ubifs.img <== Source image
vol_id=0 <== Volume ID in UBI image
vol_size=113MiB <== Volume size
vol_type=dynamic <== Allow for dynamic resize
vol_name=rootfs <== Volume name
```



```
vol_flags=autoresize <== Autoresize volume at first mount [See
calculations below to determine the value
associated with 'vol_size']
```

- ubinize

```
mtd-utils# ubi-utils/ubinize -o ubifs.ubi -m 2048 -p 128KiB -s 2048 -O
2048 ubinize.cfg
```

Where:

- -o ubifs.ubi Output file
- -m 2KiB (or 2048) Minimum flash I/O size of 2KiB page
- -p 128KiB Size of the physical eraseblock of the flash this UBI image is created for
- -s 2028 Use a sub page size of 2048 -O 2048 offset if the VID header from start of the physical eraseblock

The output of the above command, 'ubifs.ubi' is the required image.

Calculations

Usable Size Calculation As documented here, UBI reserves a certain amount of space for management and bad PEB handling operations. Specifically:

- 2 PEBs are used to store the UBI volume table
- 1 PEB is reserved for wear-leveling purposes;
- 1 PEB is reserved for the atomic LEB change operation;
- a % of PEBs is reserved for handling bad EBs. The default for NAND is 1%
- UBI stores the erase counter (EC) and volume ID (VID) headers at the beginning of each PEB.
- 1 min I/O unit is required for each of these.

To calculate the full overhead, we need the following values:

Symbol	Meaning	value
-----	-----	

SP	PEB Size	128KiB
SL	LEB Size	128KiB - 2
* 2KiB = 124 KiB		
P	Total number of PEBs on the MTD device	126.5MiB / 128KiB = 1012
B	Number of PEBs reserved for bad PEB handling	1% of P = 10.12(round to 10)
O	The overhead related to storing EC and VID headers in bytes,	4KiB
	i.e. O = SP - SL	

Assume a partition size of 126.5M (We use 1.5M for bootloader and params leaving 126.5M for ubifs partition)

UBI Overhead = (B + 4) * SP + O * (P - B - 4) = (10 + 4) * 128Kib + 4 KiB * (1012 - 9- 4) = 5784 KiB = 45.1875 PEBs (round to 45)

This leaves us with 967 PEBs or 123776 KiB available for user data.

Note that we used "-c 998" in the above mkfs.ubifs command line to specify the maximum filesystem size, not "-c 967". The reason for this is that mkfs.ubifs operates in terms of LEB size (124 KiB), not PEB size (128KiB). 123776 KiB / 124 KiB

```
= 998.19 (round to 998) .
```

Volume size = 123776 KiB (~120 MiB)

Use this calculation method to calculate the size required for each ubifs image going into each of the ubifs volumes on NAND.

A sample ubinize.cfg file is included in the images folder of the release. This was used to create the tci6614-evm-ubifs.ubi image. There is also a tci6614-boot.ubifs.img that is part of the release images folder. This was created as follows: Use the mkfs.ubifs and ubinize utilities provided with the release (under bin folder)

```
cd <release_folder>/sc_mcsdk_linux_<version>/images
mkdir boot
cp images/uImage-tci6614-evm.bin boot/uImage
cp tci6614-evm.dtb boot/uImage
export
PATH=<release_folder>/sc_mcsdk_linux_<version>/bin:$PATH
mkfs.ubifs -r boot -F -o tci6614-boot.ubifs.img -m 2048 -e 126976 -c 41
cp tci6614-boot.ubifs.img images/
cd images/
ubinize -o tci6614-evm-ubifs.ubi -m 2048 -p 128KiB -s 2048 -O 2048
ubinize.cfg
```

Using UBIFS file system

Preparing NAND partition Kindly erase the NAND partition before using it for UBI file system. The partition can be erased from either u-boot or from Linux.

Follow below steps to erase.

From U-boot.

Assuming NAND partition to be erased is called ubifs and mtdparts env variable is set.

```
u-boot# nand erase.part ubifs
```

From Linux. Assuming MTD partition 2 needs to be erased and used for UBI file system.

```
root@arago-armv7:~# flash_eraseall /dev/mtd2
```

Flashing UBIFS image to a NAND partition

We can Flash UBIFS image from either Linux Kernel or U-Boot.

Follow steps mentioned here to create an UBIFS image.

From Linux,

Flash the UBI file system image (ubifs.ubi) to MTD partition "X"

```
ubiformat /dev/mtd<X> -f ubifs.ubi -s 2028 -O 2048
```

Assuming 2nd mtd partition, we can use the following command to flash the ubifs ubi image to partition 2.

```
#flash_erase /dev/mtd2 0 0
#ubiformat /dev/mtd2 -f tci6614-evm-ubifs.ubi -s 2048 -O 2048
```

Using UBIFS image as root file system

Set up the bootargs environment variable as below to use the UBIFS file system image present in a MTD partition:

```
setenv bootargs 'console=ttyS0,115200n8 mem=512M earlyprintk debug
rootwait=1 rw ubi.mtd=X,YYYY rootfstype=ubifs
root=ubi0:rootfs rootflags=sync'
```

Where X is the MTD partition number being used for file system and YYYY is the NAND page size. make sure that an UBI file system is flashed into this partition before passing it as a boot partition for Linux.

Assuming 2nd mtd partition and page size of 2048,

```
setenv bootargs 'console=ttyS0,115200n8 mem=512M earlyprintk debug
rootwait=1 rw ubi.mtd=2,2048 rootfstype=ubifs
root=ubi0:rootfs rootflags=sync'
```

Updating Boot volume images from Linux kernel

The files in the boot volume may be removed and replaced by new file and EVM may be rebooted using the new images. See below the steps to replace the file in boot volume.

```
root@tc6614-evm:~# mkdir /mnt/boot
root@tc6614-evm:~# mount -t ubifs /dev/ubi0_0 /mnt/boot

UBIFS: mounted UBI device 0, volume 0, name "boot"
UBIFS: file system size: 3936256 bytes (3844 KiB, 3 MiB, 31 LEBs)
UBIFS: journal size: 1142785 bytes (1116 KiB, 1 MiB, 8 LEBs)
UBIFS: media format: w4/r0 (latest is w4/r0)
UBIFS: default compressor: lzo
UBIFS: reserved for root: 0 bytes (0 KiB) root@tc6614-evm:~#
root@tc6614-evm:~# cd /mnt/boot
root@tc6614-evm:/mnt/test# ls
tc6614-evm.dtb uImage
```

The above files can be deleted and overwritten with new file. For example to replace the dtb file, do

```
root@tc6614-evm:/mnt/test# rm tc6614-evm.dtb
```

TFTP the tc6614-evm.dtb to this folder or using the DTC compiler on target DTS file may compiled and saved in this folder. Please note that the file name should match with default files in the boot volume.

Once done unmount the folder as

```
root@tc6614-evm:/# umount /mnt/boot
UBIFS: un-mount UBI device 0, volume 0 root@tc6614-evm:/# reboot
```

Common Clock Framework

The old DaVinci clock driver (arch/arm/mach-davinci/clock.c) is now replaced by the common clock framework introduced in upstream Linux kernel recently. Common clock framework that has factored out all of the platform independent layers into a set of driver modules that are used by platform code to initialize the clock tree on a SoC. The clock driver code is located under drivers/clk folder in the Linux source tree. The platform needs to implement drivers for a custom clock hardware that are not modelled by the framework common drivers. Example of the existing drivers are clk-fixed, clk-divider, clk-mux etc. When implementing a clock driver, the driver needs to implement a subset of the clk ops applicable to the specific clock hardware. The driver also implements a

register_*() function to register the driver with the framework. Two additional clock drivers are added for TCI6614 namely clk-pll to model main PLL and clk-psc to model the psc clock. For more details on the Clock framework, please refer Documentation/clk.txt

The code is organized into following files:-

- arch/arm/mach-davinci/include/mach/davinci-clock.h
 - Defines the structures and types for defining parameters for various clock drivers used by a specific SoC
- arch/arm/mach-davinci/include/mach/pll.h
 - Defines the structures and types for PLL based clocks (clk-pll and clk-dividers)
- arch/arm/mach-davinci/include/mach/psc.h
 - Defines the structures and types for PSC based clocks (clk-psc)
- arch/arm/mach-davinci/tci6614.c
 - updates to use the new common clock framework
- drivers/clk/davinci/clock.c
 - DaVinci clock initialization code. The davinci_clk_init() is implemented here and is called from arch/arm/mach-davinci/timer.c
- drivers/clk/davinci/clk-pll.c
 - Main PLL clock driver
- drivers/clk/davinci/clk-psc.c
 - PSC clock driver

The common clock code is enabled through the CONFIG_COMMON_CLK KConfig option.

There are debugfs entries for each clock node to display the rates, usecounts etc. This replaces the old debugfs entry to display the clock information. For example to show the main PLL clock rate, use the following command

```
root@tci6614-evm:/# cat /debug/clk/ref_clk/main_pll/clk_rate
983040000
```

Device drivers

UART

The serial port driver used is implemented in drivers/tty/serial/8250.c which is the driver for 8250/16550-type serial ports. There are two serial ports available in the tci6614 evm. The driver implements the Low Level Serial API as described in Documentation/serial/driver documentation in Linux kernel tree. Please refer this for more details.

GPIO

The DaVinci GPIO driver available at arch/arm/mach-davinci/gpio.c is re-used for TCI6614. The GPIO resource information is configured in the tci6614_soc_info structure which is used by the DaVinci GPIO driver. The TCI6614 supports up to 32 GPIO pins that can be configured as input or output. GPIO also supports interrupts and can be configured to generate interrupt on a per pin basis. The users of GPIO driver call the GPIO library functions to access the GPIO function both from the kernel space and user space. Please refer Documentation/gpio.txt for more details.

SPI & SPI NOR Flash

DaVinci SPI master driver is re-used for TCI6614. The driver code is located in `drivers/spi/spi-davinci.c`. For understanding the driver code, there are some documentation available under `Documentation/spi` folder of Linux kernel tree. `spi-summary` provides an overview of the Linux kernel SPI support. SPI NOR Flash is a slave device connected to the SPI bus. The driver for SPI NOR flash part used in the EVM is available under `drivers/mtd/devices/m25p80.c`. `flash_erase` and `flashcp` commands are available in the rootfs supplied with the release for erase and copy files to NOR flash device.

Following mtd partitions are available on SPI NOR Flash mtd device:-

- `u-boot-spl` - For storing u-boot spl
- `test` - test partition

Please refer the `board-tci6614-evm.c` for the size of these partitions.

NAND Flash

DaVinci NAND device driver (`drivers/mtd/nand/davinci-nand.c`). The NAND device is accessed through the mtd device driver. `flash_erase` and `nandwrite` commands are available in the rootfs provided with the release to erase and copy files to the NAND device.

Following partitions are available on NAND mtd device:-

- `u-boot`
- `params` (for saving the env variable in u-boot)
- `kernel`
- `filesystem`

Please refer the `board-tci6614-evm.c` for the size of these partitions.

I2C & EEPROM

DaVinci i2c adapter driver is re-used for TCI6614 and the code is in `drivers/i2c/busses/i2c-davinci.c`. There are some useful documentation under `Documentation/i2c` folder to understand the i2c protocol and driver. The EEPROM is a slave i2c device and the driver for the same is available under `drivers/misc/eeprom/at24.c`. A sysfs interface is provided to access the eeprom from the user space and is available through `/sys/devices/platform/i2c_davinci.1/i2c-1/1-0050/eeprom`

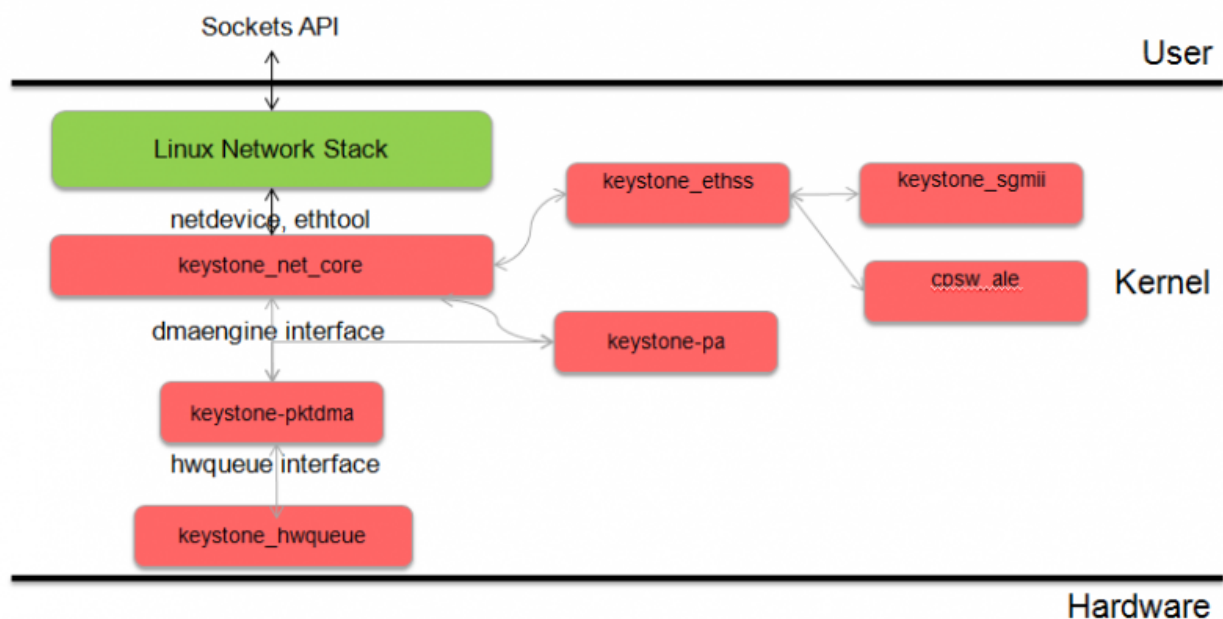
Watchdog

DaVinci watchdog device driver (drivers/watchdog/davinci_wdt.c) is re-used for TCI6614. The watchdog timer is initialized in time.c under arch/arm/mach-davinci folder. APIs provided in user space to configure watchdog timer. Please refer the documents under Documentation/watchdog for more details on APIs available and the device driver code for APIs supported.

Network Driver

Overview

NetCP Linux Driver Stack



- The NETCP driver code is at **/drivers/net/ethernet/ti/keystone_net_core.c**.

The network driver sets up the streaming switch interface to route all packets to PDSP0. At present this is the only mode that is supported by the driver. In the near future we intend to add a feature in the driver which will enable a user to bypass the PA.

The SGMII ports need to be initialized and appropriately configured. SERDES initialization also needs to be done. These APIs are found in **keystone_sgml.c**.

The switch and the mac sliver ports need to be configured. In the switch we enable all statistics to be collected. All ports of the ALE are put in the forward mode.

When the network device is opened, a function is called to configure the PA. This function will download the appropriate firmware to all the 6 PDSPs Packet Classifier 1 image **pa_pdsp02_1_3_0_1.fw.ihex** is downloaded to PDSP0, PDSP1 and PDSP2 Packet Classifier 2 image **pa_pdsp3_1_3_0_1.fw.ihex** is downloaded to PDSP3 Packet Modifier image **pa_pdsp45_1_3_0_1.fw.ihex** is downloaded to PDSP4 and PDSP5

We also add FIVE MAC rules to PDSP 0.

A private structure is maintained which stores the dma channels and dma channel names and rx state. The platform device and network device are also stored in this private structure each packet has a structure (called **netcp_packet**) associated with it. The comprises of a scatterlist array, the number of scatterlist entries, the status of the dma and the dma cookie. A pointer to the skb is also stored in this structure.

Model:

1) Open:

On device open, we request DMA channels using standard dma engine APIs provided by the dmaengine layer. The capabilities of the DMA are appropriately set and the appropriate channels are requested by name. We invoke the `dma_request_channel_by_name` API to acquire dma engine channels. The netdev open also sets up the streaming switch, SGMII, Serdes, switch and mac sliver. Finally we configure the PA

2) Stop:

A standard netdev stop operation stops the netif queue. It sets the receive state to teardown and calls `dmaengine_pause` APIs from the dmaengine layer to pause both the RX and TX channel. After NAPI is disabled, the operation goes onto release the dma channels by calling the `dma_release_channel` API. The RX state is set to invalid and the ethernet subsystem is stopped

3) Polling

A NAPI poll function is implemented, which sets the RX state to poll and resume the dma engine. It will then go onto initialize the queues. We allocate memory for each packet and initialize the scatterlist. We iterate in a loop till we run out memory for the descriptors. These dma descriptors are acquired using a dmaengine API called `device_prep_slave_sg`. We pass appropriate arguments to the API. At this instant we are not concerned about the Software info and PS info associated with each descriptor. We use a single scatterlist associated with each packet. `netcp_rx_complete` is the callback associated with the dma engine on the RX side. The callback function checks for the correct DMA and RX state and warns the user on seeing abnormal behavior. It then goes onto stahs the received packet to the tail of a linked list.

4) Xmit

`netcp_ndo_start_xmit` is the stanndard netdev operation that is used to push an outgoing packet. Again we have a structure for each packet. This will contain an array of scatterlists and the number of scatterlist entries which at this point should be 1. We then call the `device_prep_slave_sg` API with appropriate parameters to acquire a descriptor for the TX operation. '`netcp_tx_complete`' is the callback fuction attached to each TX dma callback. All the network statistics are appropriately updated on a successful transfer. The callback then proceeds to free the skb with the `dev_kfree_skb_any` API.

5) Receive

In the `napi_poll` netdev operation, we call the `netcp_refill_rx` function which will allocate skbs and atch these skbs to a descriptor until we run out of descriptor memory. the `deliver_stash` routine fills the appropriate RX

- The QMSS driver code is at **drivers/hwqueue/keystone_hwqueue.c**. The keystone hwqueue drivers sits below the linux hwqueue layer. This layer has been implemented by TI. There are API's to open and close queues and to push and pop descriptors.
- The Packet DMA driver code is at **drivers/dma/keystone-pktdma.c**.

The dmaengine interface exposes logical DMA channels to user drivers. These logical channels are explicitly specified via device tree, and are fixed at probe time. Further, dmaengine logical channels are unidirectional, with the direction specified in the device tree. This deviates slightly from the traditional dmaengine model, where channels may be used for both to-device and from-device directions.

The specifics on the device tree bindings for Packet DMA can be found in: [Documentation/devicetree/bindings/dma/keystone-pktdma.txt](#)

Mapping to Packet DMA Hardware :

Both transmit and receive channels map to a pair of underlying hardware queues provided by the hardware queue subsystem. This driver supports only a strict ring model of descriptor handoff to/from hardware DMAs. More exotic modes of Packet DMA usage are not supported at present. This includes support for packet buffer bins of varying

sizes, as well as support for packet scatter/gather.

Most of the mappings to hardware resources (tx channels, rx channels, rx flows) are optional in this implementation. This allows the Packet DMA driver to be used flexibly in a variety of different use cases ranging from direct I/O rings with DSPs/accelerators, DMA copied rings with the infrastructure DMA, and traditional packet I/O rings with Ethernet and SRIO hardware.

Usage Model:

Higher level drivers have a consistent usage model independent of the transfer direction. Typically, this involves the following steps:

1. Channel open:

DMA channels are opened using standard dmaengine API calls(`dma_request_channel` or `dma_request_channel_by_name`). The `dma_request_channel()` interface allows for a flexible "filter" model by which the requestor may scan and select a channel based on various criteria. In the Packet DMA usage model, the "by name" variant is expected to be used in most scenarios.

2. Transfer request submission:

Once a channel is opened, the user may submit one or more transfer requests on the opened channel. This is a three step process: a. Allocate a transfer descriptor:

The DMA device includes a `device_prep_slave_sg()` interface which allocates and fills up a transfer descriptor based on the contents of a scatterlist. Standard Linux for scatterlist manipulation APIs (`sg_table_init()`, `dma_map_sg()`, etc.) apply here. The Keystone Packet DMA recognizes a couple of extra flags in the `device_prep_slave_sg()` call. These flags indicate the presence of "software info" and/or "protocol specific" info as required by the user driver. When so indicated, the software/protocol info buffers are passed as separate scatterlist entries in the scatterlist array passed into the `device_prep_slave_sg()` call. The Keystone Packet DMA driver then ensures that these pieces of information are copied to/from the descriptor as needed. Note: The DMA engine writes to the scatterlist entries on completion. For example, on RX channels the actual packet/buffer size will be filled into the scatterlist entry. Therefore, the scatterlist array MUST NOT be placed on the stack.

b. Prepare descriptor callbacks:

The `device_prep_slave_sg()` call returns a DMA transfer descriptor. Note that this descriptor has nothing to do with the hardware descriptor specified in the Keystone Navigator architecture. The user driver is expected to fill up a callback and an optional callback parameter into the returned descriptor.

c. Submit descriptor:

Once the descriptor callbacks have been set, the user driver is expected to call `dmaengine_submit()` in order to hand the descriptor back to the DMA engine. The `dmaengine_submit()` call returns a transfer cookie that may be used at subsequent points to check the transfer status (via `dma_async_is_tx_complete()`). This cookie must also be checked for error values using the `dma_submit_error()` macro.

2. Transfer request completion:

When a pending DMA transfer has completed, the DMA engine driver will call the user provided callback with the user provided callback parameter. This callback may be in an atomic context, and therefore the callback must not block. While within the callback, the user driver may use the `dma_async_is_tx_complete()` function to determine if the transfer completed successfully (see `dma_status` codes). The callback may also optionally pause the channel (see `dmaengine_pause()`) to prevent further callbacks. Once the callback returns, the DMA driver frees up associated transfer resources.

3. Channel pause and resume:

User drivers may pause or resume the DMA channel at any point of time. Note that pause and resume have no impact on the actual hardware operation of the DMA, and in reality DMA transfers continue as normal. However,

when a channel is paused, the DMA driver will not issue any further callbacks, and may choose to disable the underlying hardware interrupt or other notification mechanism as necessary.

4. Channel close:

As a part of the user driver's shutdown sequence, it must close all opened DMA channels in order to free resources. In the process of closing down the channel, the DMA driver will issue callbacks regardless of the pause/resume state. Further, some of these transfers may indicate a failure status (DMA_ERROR) indicating that the transfer did not complete.

Other Features that can be modified through the device tree:

Loopback mode: This mode supports Packet DMA loopback mode, as is the case with the infrastructure DMA built into the hardware queue subsystem.

Big-Endian mode: When running in mixed-endian mode (i.e. ARM-LE, DSP-BE), the Keystone SoC peripherals (including DMAs) are big-endian. Consequently, descriptors need to be endian converted at read/write.

Enable-All mode: The mapping between hardware channels and hardware receive flows are not consistent across DMA instances on the Keystone SoC. For example, infrastructure DMAs have a near one-to-one mapping between flows and RX channels, but the network coprocessor DMA does not. In order to support such variations, this driver has an enable-all mode that simply enables all TX and RX channels on the DMA at probe time, instead of incrementally enabling/disabling DMA resources at logical channel initialization.

Flow Tag support: For infrastructure mode DMA usage, the transmit descriptors need to specify the receive flow number in the descriptor "flow tag". This is useful in the virtual ethernet use case, where ARM/Linux and DSP software open their respective receive flows, and transmit packets to each others' flows. In this scenario, the DSP flow number would be configured into the Linux dmaengine channel's flowtag.

Debug support: When built with debug enabled (CONFIG_DMADEVICES_DEBUG), debug messages of individual channels and individual dmaengine instances are controllable via device tree. By default all debugs are turned off.

- The Packet Accelerator driver is at **drivers/net/ethernet/ti/keystone_pa.c**.
 - The Packet Accelerator has 6 PDSPs.
 - The driver will download firmware version 1.2.1.0 to these 6 PDSPs.
 - The packet classifier image 1 is downloaded to PDSP0, PDSP1 and PDSP2.
 - The packet classifier image 2 is downloaded to PDSP3.
 - The packet modifier image is downloaded to PDSP4 and PDSP5.
- The SGMII driver code is at **drivers/net/ethernet/ti/keystone_sgmi.c**.
 - This driver provides APIs to configure the SGMII ports and initialize the Serdes.
- MAC sliver ports and the switch need to be initialized and the relevant code can be found at **drivers/net/ethernet/ti/keystone_ethss.c**.

Setting up an NFS filesystem

An NFS filesystem can be setup during development.

To setup an NFS filesystem, first obtain a tarball of the filesystem. In SC-MCSDK the tarball of the filesystem is provided in `images/ti-scmsdk-rootfs-tci6614-evm.tar.gz`. This should be untarred on the linux host machine. This section will explain how to setup an NFS server on a Ubuntu 10.04 linux host.

- Untar the filesystem that is made as part of the release into location `/opt/filesys`
- On the linux host open file `/etc/exports` and add the following line

```
/opt/filesys *(rw, subtree_check, no_root_squash, no_all_squash, sync)
```

- Please note that the location of the filesystem on the host should be the same as that in `/etc/exports`
- Next we have to start the nfs server and this is achieved by giving the following command

```
/etc/init.d/nfs-kernel-server restart
```

Modifying the command line for NFS filesystem boot

The kernel command line should be appropriately modified to enable kernel boot up with an NFS filesystem

Add the following to the kernel command line **`root=/dev/nfs nfsroot=192.168.1.140:/opt/filesys,v3,tcp rw ip=dhcp`**

During kernel boot up, in the boot up log we should be able to see the following

Kernel command line: `earlyprintk debug console=ttyS0,115200n8 ip=dhcp mem=512M rootwait=1 rootfstype=nfs root=/dev/nfs rw nfsroot=192.168.1.140:/opt/filesys,v3,tcp`

QM and PA Firmware

- The NETCP driver will download the QMSS firmware. The firmware is acquired by generic linux kernel "request_firmware" API and then downloaded to the appropriate PDSP.
- The PA driver will download **PA firmware version 1.3.0.1** to all the 6 PDSPs.
- **Packet Classifier 1** Image is downloaded to PDSP0, PDSP1 and PDSP2.
- **Packet Classifier 2** Image is downloaded to PDSP3.
- **Packet Modifier** Image is downloaded to PDSP4 and PDSP5.
- During kernel boot up, user should see the following in the kernel boot up log

```
keystone-netcp 2090000.netcp: firmware: using built-in firmware keystone/pa_pdsp02_1_3_0_1.fw
```

```
keystone-netcp 2090000.netcp: firmware: using built-in firmware keystone/pa_pdsp02_1_3_0_1.fw
```

```
keystone-netcp 2090000.netcp: firmware: using built-in firmware keystone/pa_pdsp02_1_3_0_1.fw
```

```
keystone-netcp 2090000.netcp: firmware: using built-in firmware keystone/pa_pdsp3_1_3_0_1.fw
```

```
keystone-netcp 2090000.netcp: firmware: using built-in firmware keystone/pa_pdsp45_1_3_0_1.fw
```

```
keystone-netcp 2090000.netcp: firmware: using built-in firmware keystone/pa_pdsp45_1_3_0_1.fw
```

Modifying the device tree

- Resources needed by components like QMSS, Packet DMA, packet Accelerator can be modified using the device tree. The device tree itself can be found at **arch/arm/boot/dts/tci6614-evm.dts**. Resource allocation will differ from use case to use case and should be modified based on a particular need. Appropriately modify the device tree based on the following documents below.
- Device tree bindings for packet DMA can be found at Packet DMA Device Tree Bindings ^[21]
- Keystone Network driver device tree bindings can be found at Network Driver Device Tree Bindings ^[22]
- Device tree bindings for the QMSS layer can be found at Hardware Queue Device Tree bindings ^[23]

PA Timestamping

- PA timestamping has been implemented in the network driver.
- All Receive packets will be timestamped and this timestamped by PDSP0 and this timestamp will be available in the timestamp field of the descriptor itself.
- To obtain the TX timestamp, we call a PA API to format the TX packet. Essentially what we do is to add a set of params to the "PSDATA" section of the descriptor. This packet is then sent to PDSP5. Internally this will route the packet to the switch. The TX timestamp is obtained in a return packet and we have designed the network driver in such a way to obtain both ethernet RX packet and the TX timestamp packet on the same queue and flow.
- The way we differentiate between an Ethernet RX packet and a TX timestamp packet is based on the "Software info 0" field of the descriptor.
- To obtain the timestamps itself, we use generic kernel APIs and features.
- Appropriate documentation for this can be found at Timestamping Documentation ^[24]
- The timestamping was tested with open source timestamping test code found at Timestamping Test Code ^[25]

```
root@tci6614:/# ./timestamping eth0 SOF_TIMESTAMPING_RX_HARDWARE
```

VLAN

Keystone devices have an on chip Common Platform Ethernet Switch (CPSW) which can operate in the VLAN aware mode. The CPSW also has an Address Lookup Engine which has VLAN support and has provision for 1024 entries including VLAN.

The linux core network driver has 2 network device operations to add and remove vlan ids.

- `ndo_vlan_rx_add_vid` to add vlan ids
- `ndo_vlan_rx_kill_vid` to remove vlan ids.

Adding and removing vlans can be achieved using the `vconfig` utility which is already part of the filesystem released by TI.

The following updates have been made

- To enable generic vlan support in the linux kernel, we have to enable the config option , `CONFIG_VLAN_8021Q` in the kernel config. After enabling vlan support, on using the `vconfig` utility to add a vlan, the network device will invoke `ndo_vlan_rx_add_vid`. The keystone specific ndo will be called `netcp_rx_add_vid`.
- Removing a vlan can be achieved using the `vconfig` utility. On doing this, the network device will invoke the `ndo_vlan_rx_kill_vid`. The keystone specific ndo will be called `netcp_rx_kill_vid`.
- Each of these ndo's will implement call outs to cpsw functions to add/remove vlan ids. The `netcp_rx_add_vid` will implement a call out to the Ethernet subsystem driver to add a vlan id. The `netcp_rx_kill_vid` will implement a call out to the Ethernet subsystem driver to remove a vlan id.

- The add_vid call out from keystone net core engine will be implemented as cpsw_add_vid in the Ethernet subsystem driver. The del_vid call out from keystone net core engine will be implemented as cpsw_del_vid in the Ethernet subsystem driver.
- The cpsw_add_vid function in the Ethernet subsystem driver will call a CPSW ALE api to add a vlan entry. The parameters to this api will be the vlan id, the vlan member list, the unregistered multicast flood mask, the registered multicast flood mask and force untagged packet egress.
- The cpsw_del_vid function in the Ethernet subsystem driver will call a CPSW Ale api to remove a vlan entry.
- When the cpsw device is open it will initialize the host port and then each slave port of the the switch. While each port is being initialized, the appropriate sgmiid will also be initialized in a mode that will be decided by the user. The user will have the option to select the mode through the device tree.
- While initializing each slave depending the slave needs to be aware of the link, whether it is mac-phy or mac-mac. If the link is mac-phy then phylib support will be used with the phy state machine detecting changes in the link status. If the link is mac-mac then the sgmiid link status register needs to be monitored

Testing can be done using the “vconfig” utility. Vconfig is already part of the filesystem. This utility can be used to add/remove vlans.

The linux host may/may not have the vconfig utility. On Ubuntu hosts, issue the command

apt-get install vlan

The above will install vconfig on the Ubuntu host

To add a vlan on the target EVM, issue the command

vconfig add eth0 50

Then issue the command

ifconfig eth0.50 192.168.2.40 netmask 255.255.255.0 up

A VLAN with the same tag will also have to be setup on the linux host as well

Issue the command

vconfig add eth0 50

Then issue the command

ifconfig eth0.50 192.168.2.50 netmask 255.255.255.0 up

At the end of this the linux host will have an IP of 192.168.2.50 and the target EVM will have an IP of 192.168.2.40. Ping between the two vlans

We can have multiple tci6614 EVMs connected to the switch. Each of these EVMs should be able to communicate over vlan.

Vlan specific statistics can be obtained from

cat /proc/net/vlan/eth0.50

and

ifconfig eth0.50

In the above example we have used vlan id of 50. Other vlan ids between 1 to 4095 can also be used. The commands will need to appropriately change

SGMII

The SGMII driver can initialize both SGMII 0 and SGMII 1 in the appropriate mode. It is conceivable that we may have to configure each port in a different mode. This becomes a necessity in a case where we cascade multiple tci 6614 EVMs. Only slave port 1(port 2) is connected on the EVM. We can use a break out card to cascade multiple Appleton EVMs. In this case, atleast 1 slave port has to be configured to operate in the cpgmac-cpgmac mode. A certain level of configurability will be added to the sgmi driver so that when we call api's to configure the appropriate SGMII, the api will take a parameter so that the SGMII can be appropriately configured. There are 4 modes that can be supported

- Mac to Phy
- Mac connected to Mac forced link
- Mac connected Mac slave
- Mac connected to fiber

The mode to operate each SGMII will be obtained from the device tree.

We can set the link-interface for each slave port in the device tree bindings.

To set the sgmi port in mac-mac autonegotiate mode, set link-interface to 0

To set the sgmi port in mac-phy autonegotiate mode, set link-interface to 1

To set the sgmi port in mac-mac forced link mode, set link-interface to 2

To set the sgmi port in mac-fiber mode, set link-interface to 3

Quality of Service

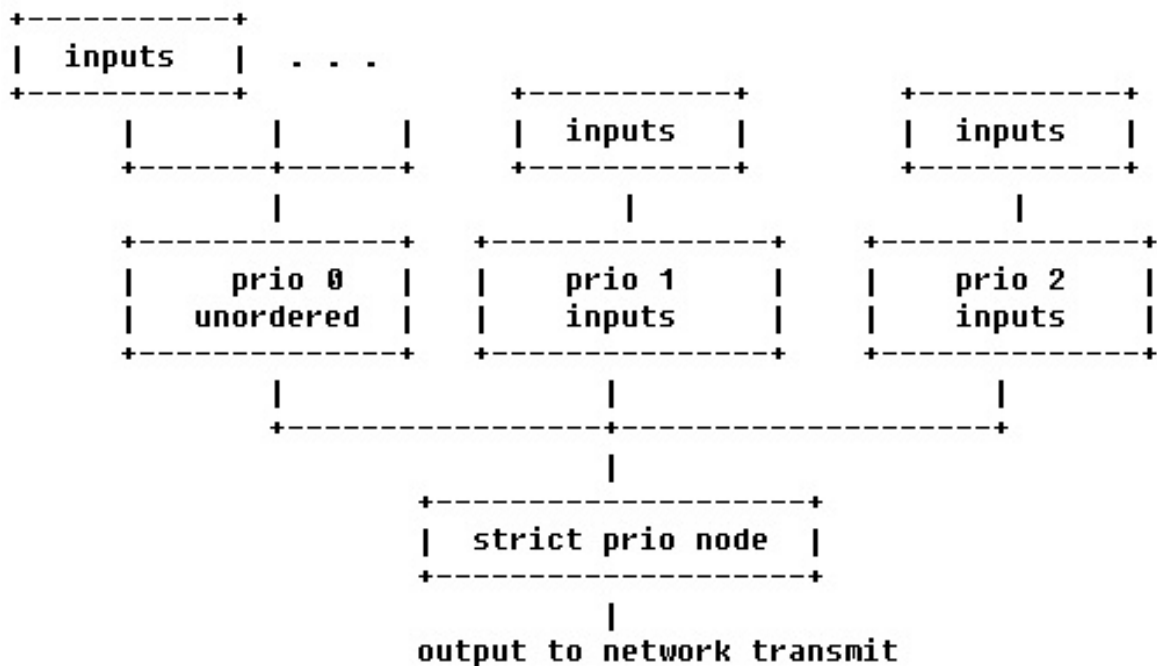
The linux hardware queue driver will download the Quality of Service Firmware to PDSP 1 of QMSS. PDSP 0 has accumulator firmware.

The firmware will be programmed by the linux keystone hardware queue QoS driver.

The configuration of the firmware is done with the help of device tree bindings. These bindings are documented in the kernel itself at **Documentation/devicetree/bindings/hwqueue/keystone-qos.txt**

QoS Tree Configuration

The QoS implementation allows for an abstracted tree of scheduler nodes represented in device tree form. An example is depicted below



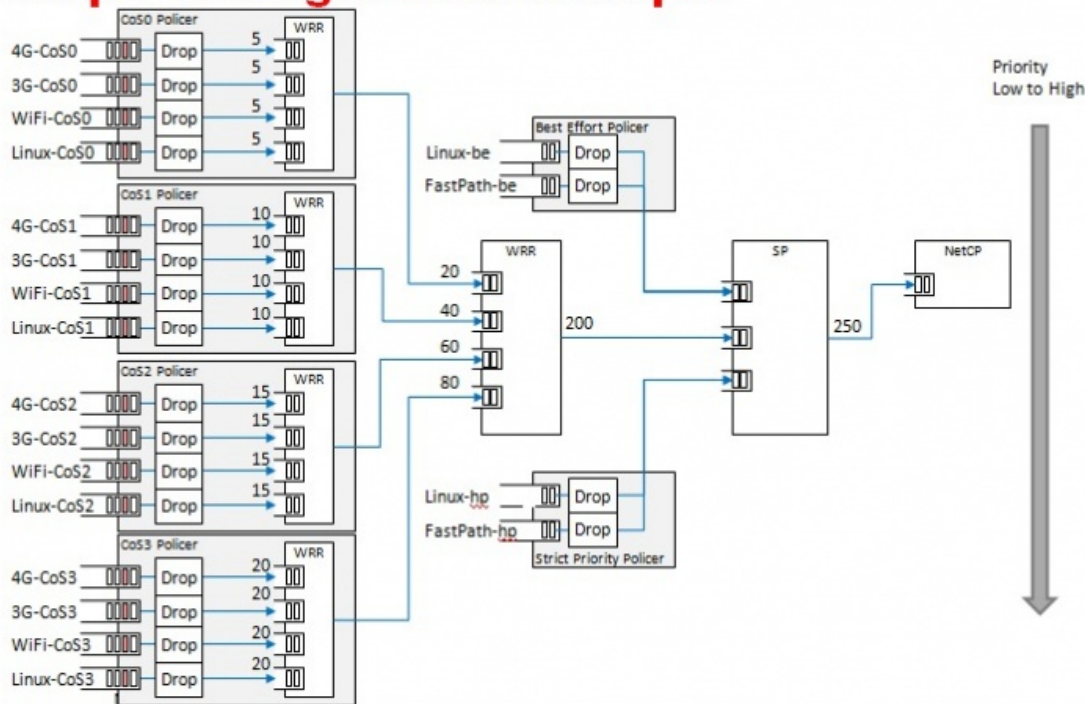
At each node, shaping and dropping parameters may be specified, within limits of the constraints outlined in this document. The following sections detail the device tree attributes applicable for this implementation.

The actual qos tree configuration can be found at [linux-tci6614/arch/arm/boot/dts/tci6614-evm.dts](#).

The device tree has attributes for configuring the QoS shaper. In the sections below we explain the various qos specific attributes which can be used to setup and configure a QoS shaper.

In the device tree we are setting up a shaper that is depicted below

Shaper configuration example



When egress shaper is enabled, all packets will be sent to the QoS firmware for shaping via a set of the queues starting from the QoS base queue which is 8000 by default. DSCP value in the IP header(outer IP incase of IPSec tunnels) or VLAN pbits (if VLAN interface) are used to determine the QoS queue to which the packet is sent. E.g., if the base queue is 8000, if the DSCP value is 46, the packet will be sent to queue number 8046. i.e., base queue number + DSCP value Incase of VLAN interfaces, if the pbit is 7, the packet will be sent to queue number 8071. i.e., base queue number + skip 64 queues used for DSCP + pbit value.

Shaper Configuration example, details

FlowID	DSCP	VLAN p-bit	CoS	FastPath DSCP Queues 8000-8063	FastPath VLAN Queues 8064-8071	Linux CoS Queues 8072-8079	Flow Description
FastPath-hp	46	7		8046	8071	N/A	3gpp, high priority
4G-CoS3	34	4		8034	8068	N/A	4G Class3, platinum
3G-CoS3	36	4		8036			3G Class3, platinum
WiFi-CoS3	38	4		8038			WiFi Class3, platinum
4G-CoS2	26	3		8026			4G Class2, gold
3G-CoS2	28	3		8028	8067	N/A	3G Class2, gold
WiFi-CoS2	30	3		8030			WiFi Class2, gold
4G-CoS1	18	2		8018			4G Class1, silver
3G-CoS1	20	2		8020			3G Class, silver
WiFi-CoS1	22	2		8022	8066	N/A	WiFi Class1, silver
4G-CoS0	10	1		8010			4G Class0, bronze
3G-CoS0	12	1		8012			3G Class0, bronze
WiFi-CoS0	14	1		8014			WiFi Class0, bronze
FastPath-be	0+ all the rest	0+all the rest		8000+all the rest	8064+all the rest	N/A	3GPP, best effort
Linux-hp	Per DSCP to CoS mapping	Per p-bit to CoS mapping	5	N/A	N/A	8077	Linux high priority
Linux-CoS3			4	N/A	N/A	8076	Linux Class3, platinum
Linux-CoS2			3	N/A	N/A	8075	Linux Class2, gold
Linux-CoS1			2	N/A	N/A	8074	Linux Class1, silver
Linux-CoS0			1	N/A	N/A	8073	Linux Class0, bronze
Linux-be			0	N/A	N/A	8072+all the rest	Linux best effort

QoS Node Attributes

The following attributes are recognized within QoS configuration nodes:

- **"strict-priority" and "weighted-round-robin"**

e.g. strict-priority;

This attribute specifies the type of scheduling performed at a node. It is an error to specify both of these attributes in a particular node. The absence of both of these attributes defaults the node type to unordered(first come first serve).

- **"weight"**

e.g. weight = <80>;

This attribute specifies the weight attached to the child node of a weighted-round-robin node. It is an error to specify this attribute on a node whose parent is not a weighted-round-robin node.

- **"priority"**

e.g. priority = <1>;

This attribute specifies the priority attached to the child node of a strict-priority node. It is an error to specify this attribute on a node whose parent is not a strict-priority node. It is also an error for child nodes of a strict-priority node to have the same priority specified.

- **"byte-units" or "packet-units"**

e.g. byte-units;

The presence of this attribute indicates that the scheduler accounts for traffic in byte or packet units. If this attribute is not specified for a given node, the accounting mode is inherited from its parent node. If this attribute is not specified for the root node, the accounting mode defaults to byte units.

- **"output-rate"**

e.g. output-rate = <31250000 25000>;

The first element of this attribute specifies the output shaped rate in bytes/second or packets/second (depending on the accounting mode for the node). If this attribute is absent, it defaults to infinity (i.e., no shaping). The second element of this attribute specifies the maximum accumulated credits in bytes or packets (depending on the accounting mode for the node). If this attribute is absent, it defaults to infinity (i.e., accumulate as many credits as possible).

- **"overhead-bytes"**

e.g. overhead-bytes = <24>;

This attribute specifies a per-packet overhead (in bytes) applied in the byte accounting mode. This can be used to account for framing overhead on the wire. This attribute is inherited from parent nodes if absent. If not defined for the root node, a default value of 24 will be used. This attribute is passed through by inheritance (but ignored) on packet accounted nodes.

- **"output-queue"**

e.g. output-queue = <645>;

This specifies the QMSS queue on which output packets are pushed. This attribute must be defined only for the root node in the qos tree. Child nodes in the tree will ignore this attribute if specified.

- **"input-queues"**

e.g. input-queues = <8010 8065>;

This specifies a set of ingress queues that feed into a QoS node. This attribute must be defined only for leaf nodes in the QoS tree. Specifying input queues on non-leaf nodes is treated as an error. The absence of input queues on a leaf node is also treated as an error.

- **"stats-class"**

e.g. stats-class = "linux-best-effort";

The stats-class attribute ties one or more input stage nodes to a set of traffic statistics (forwarded/discarded bytes, etc.). The system has a limited set of statistics blocks (up to 48), and an attempt to exceed this count is an error. This attribute is legal only for leaf nodes, and a stats-class attribute on an intermediate node will be treated as an error.

- **"drop-policy"**

e.g. drop-policy = "no-drop"

The drop-policy attribute specifies a drop policy to apply to a QoS node (tail drop, random early drop, no drop, etc.) when the traffic pattern exceeds specified parameters. The drop-policy parameters are configured separately within device tree (see "Traffic Police Policy Attributes section below). This attribute defaults to "no drop" for applicable input stage nodes. If a node in the QoS tree specifies a drop-policy, it is an error if any of its descendent nodes (children, children of children, ...) are of weighted-round-robin or strict-priority types.

Traffic Police Policy Attributes

The following attributes are recognized within traffic drop policy nodes:

- **"byte-units" or "packet-units"**

e.g. byte-units;

The presence of this attribute indicates that the drop accounts for traffic in byte or packet units. If this attribute is not specified, it defaults to byte units. Policies that use random early drop must be of byte unit type.

- **"limit"**

e.g. limit = <10000>;

Instantaneous queue depth limit (in bytes or packets) at which tail drop takes effect. This may be specified in combination with random early drop, which operates on average queue depth (instead of instantaneous). The absence of this attribute, or a zero value for this attribute disables tail drop behavior.

- **"random-early-drop"**

e.g. random-early-drop = <32768 65536 2 2000>;

The random-early-drop attribute specifies the following four parameters in order:

low threshold: No packets are dropped when the average queue depth is below this threshold (in bytes). This parameter must be specified. high threshold: All packets are dropped when the average queue depth above this threshold (in bytes). This parameter is optional, and defaults to twice the low threshold.

max drop probability: the maximum drop probability

half-life: Specified in milli seconds. This is used to calculate the average queue depth. This parameter is optional and defaults to 2000.

Sysfs support

The keystone hardware queue driver has sysfs support for statistics, drop policies and the tree configuration.

```
root@tc-i6614-evm:/sys/devices/soc6614.0/2a000000.hwqueue#
root@tc-i6614-evm:/sys/devices/soc6614.0/2a000000.hwqueue#
root@tc-i6614-evm:/sys/devices/soc6614.0/2a000000.hwqueue#
root@tc-i6614-evm:/sys/devices/soc6614.0/2a000000.hwqueue# pwd
/sys/devices/soc6614.0/2a000000.hwqueue
root@tc-i6614-evm:/sys/devices/soc6614.0/2a000000.hwqueue# ls
driver      modalias   qos         subsystem  uevent
root@tc-i6614-evm:/sys/devices/soc6614.0/2a000000.hwqueue# cd qos/
root@tc-i6614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos# ls
drop-policies  qos-tree      statistics
root@tc-i6614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos#
root@tc-i6614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos# █
```

The above shows the location in the kernel where sysfs entries for the keystone hardware queue can be found. There are sysfs entries for qos. Within the qos directory there are separate directories for statistics, drop-policies and the qos-tree.

Statistics are displayed for each statistics class in the device tree. Four statistics are represented for each stats class.

- bytes forwarded
- bytes discarded
- packets forwarded
- packets discarded

An example is depicted below

```

drop-policies qos-tree statistics
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos#
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos#
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos# ls
drop-policies qos-tree statistics
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos# cd statistics/
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos/statistics# ls
cos0 cos2 fastpath-be linux-be linux-cos1 linux-cos3
cos1 cos3 fastpath-hp linux-cos0 linux-cos2 linux-hp
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos/statistics# cd linux
-hp/
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos/statistics/linux-hp# cat bytes_forwarded
1648
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos/statistics/linux-hp# █

```

Drop policy configuration is also displayed for each drop policy. In the case of a drop policy, the parameters can also be changed. This is depicted below. Please note the the parameters that can be modified for tail drop are a subset of the parameters that can be modified for random early drop.

```

root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos# ls
drop-policies qos-tree statistics
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos# cd drop-policies/
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos/drop-policies# ls
all-drop red-32kb-128kb tail-drop-100pkts tail-drop-64kb
no-drop red-32kb-64kb tail-drop-32kb tail-drop-64pkts
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos/drop-policies# cd re
d-32kb-64kb/
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos/drop-policies/red-32kb-64kb# ls
32768
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos/drop-policies/red-32kb-64kb#
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos/drop-policies/red-32kb-64kb#
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos/drop-policies/red-32kb-64kb#
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos/drop-policies/red-32kb-64kb#
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos/drop-policies/red-32kb-64kb# cat limit
0
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos/drop-policies/red-32kb-64kb#
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos/drop-policies/red-32kb-64kb# cat red_high
65536
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos/drop-policies/red-32kb-64kb#
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos/drop-policies/red-32kb-64kb#
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos/drop-policies/red-32kb-64kb# echo "64000" > red_high
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos/drop-policies/red-32kb-64kb#
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos/drop-policies/red-32kb-64kb# cat red_high
64000
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos/drop-policies/red-32kb-64kb#
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos/drop-policies/red-32kb-64kb#
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos/drop-policies/red-32kb-64kb#
root@tc16614-evm:/sys/devices/soc6614.0/2a000000.hwqueue/qos/drop-policies/red-32kb-64kb# █

```

Debug Filesystem support

Debug Filesystem(debugfs) support is also being provided for QoS support. To make use of debugfs support a user might have to mount a debugfs filesystem. This can be done by issuing the command.

mount -t debugfs debugfs /debug

The appropriate path and contents are shown below

```

root@tc16614-evm:/debug/qos#
root@tc16614-evm:/debug/qos# pwd
/debug/qos
root@tc16614-evm:/debug/qos# ls
config_profiles out_profiles queue_configs sched_ports
root@tc16614-evm:/debug/qos# █

```

With the debugfs support we will be able to see the actual configuration of

- QoS scheduler ports
- Drop scheduler queue configs
- Drop scheduler output profiles
- Drop scheduler config profiles

The QoS scheduler port configuration can be seen by issuing the command **cat /debug/qos/sched_ports**. This is shown below

```

root@tc16614-evm:/debug/qos# cat sched_ports
port 14
unit flags 15 group # 1 out q 8171 overhead bytes 24 throttle thresh 0 cir credit 6400000 cir max 51200000
total q's 4 sp q's 0 wrr q's 4
queue 0 cong thresh 0 wrr credit 6144000
queue 1 cong thresh 0 wrr credit 6144000
queue 2 cong thresh 0 wrr credit 6144000
queue 3 cong thresh 0 wrr credit 6144000

port 15
unit flags 15 group # 1 out q 8170 overhead bytes 24 throttle thresh 0 cir credit 6400000 cir max 51200000
total q's 4 sp q's 0 wrr q's 4
queue 0 cong thresh 0 wrr credit 6144000
queue 1 cong thresh 0 wrr credit 6144000
queue 2 cong thresh 0 wrr credit 6144000
queue 3 cong thresh 0 wrr credit 6144000

port 16
unit flags 15 group # 1 out q 8169 overhead bytes 24 throttle thresh 0 cir credit 6400000 cir max 51200000
total q's 4 sp q's 0 wrr q's 4
queue 0 cong thresh 0 wrr credit 6144000
queue 1 cong thresh 0 wrr credit 6144000
queue 2 cong thresh 0 wrr credit 6144000
queue 3 cong thresh 0 wrr credit 6144000

port 17
unit flags 15 group # 1 out q 8168 overhead bytes 24 throttle thresh 0 cir credit 6400000 cir max 51200000
total q's 4 sp q's 0 wrr q's 4
queue 0 cong thresh 0 wrr credit 6144000
queue 1 cong thresh 0 wrr credit 6144000
queue 2 cong thresh 0 wrr credit 6144000
queue 3 cong thresh 0 wrr credit 6144000

port 18
unit flags 15 group # 1 out q 8173 overhead bytes 24 throttle thresh 0 cir credit 6400000 cir max 51200000
total q's 4 sp q's 0 wrr q's 4
queue 0 cong thresh 0 wrr credit 2457600
queue 1 cong thresh 0 wrr credit 4915200
queue 2 cong thresh 0 wrr credit 7372800
queue 3 cong thresh 0 wrr credit 9830400

port 19
unit flags 15 group # 1 out q 645 overhead bytes 24 throttle thresh 0 cir credit 6400000 cir max 51200000
total q's 3 sp q's 3 wrr q's 0
queue 0 cong thresh 0 wrr credit 0
queue 1 cong thresh 0 wrr credit 0
queue 2 cong thresh 0 wrr credit 0

root@tc16614-evm:/debug/qos#

```

The Drop scheduler queue configs can be seen by issuing the command `cat /debug/qos/queue_configs`. This is shown below

```

root@tc16614-evm:/debug/qos# ls
config_profiles  out_profiles  queue_configs  sched_ports
root@tc16614-evm:/debug/qos# cat queue_configs
q cfg 0 stats q pair # 1 block 37 out prof 17
q cfg 1 stats q pair # 1 block 37 out prof 17
q cfg 2 stats q pair # 1 block 37 out prof 17
q cfg 3 stats q pair # 1 block 37 out prof 17
q cfg 4 stats q pair # 1 block 37 out prof 17
q cfg 5 stats q pair # 1 block 37 out prof 17
q cfg 6 stats q pair # 1 block 37 out prof 17
q cfg 7 stats q pair # 1 block 37 out prof 17
q cfg 8 stats q pair # 1 block 37 out prof 17
q cfg 9 stats q pair # 1 block 37 out prof 17
q cfg 10 stats q pair # 1 block 45 out prof 33
q cfg 11 stats q pair # 1 block 37 out prof 17
q cfg 12 stats q pair # 1 block 45 out prof 32
q cfg 13 stats q pair # 1 block 37 out prof 17
q cfg 14 stats q pair # 1 block 45 out prof 31
q cfg 15 stats q pair # 1 block 37 out prof 17
q cfg 16 stats q pair # 1 block 37 out prof 17
q cfg 17 stats q pair # 1 block 37 out prof 17
q cfg 18 stats q pair # 1 block 43 out prof 29
q cfg 19 stats q pair # 1 block 37 out prof 17
q cfg 20 stats q pair # 1 block 43 out prof 28
q cfg 21 stats q pair # 1 block 37 out prof 17
q cfg 22 stats q pair # 1 block 43 out prof 27
q cfg 23 stats q pair # 1 block 37 out prof 17
q cfg 24 stats q pair # 1 block 37 out prof 17
q cfg 25 stats q pair # 1 block 37 out prof 17
q cfg 26 stats q pair # 1 block 41 out prof 25
q cfg 27 stats q pair # 1 block 37 out prof 17
q cfg 28 stats q pair # 1 block 41 out prof 24
q cfg 29 stats q pair # 1 block 37 out prof 17
q cfg 30 stats q pair # 1 block 41 out prof 23
q cfg 31 stats q pair # 1 block 37 out prof 17
q cfg 32 stats q pair # 1 block 37 out prof 17
q cfg 33 stats q pair # 1 block 37 out prof 17
q cfg 34 stats q pair # 1 block 39 out prof 21
q cfg 35 stats q pair # 1 block 37 out prof 17
q cfg 36 stats q pair # 1 block 39 out prof 20
q cfg 37 stats q pair # 1 block 37 out prof 17
q cfg 38 stats q pair # 1 block 39 out prof 19
q cfg 39 stats q pair # 1 block 37 out prof 17
q cfg 40 stats q pair # 1 block 37 out prof 17
q cfg 41 stats q pair # 1 block 37 out prof 17
q cfg 42 stats q pair # 1 block 37 out prof 17

```

The Drop scheduler output profiles can be seen by issuing the command `cat /debug/qos/out_profiles`. This is shown below


```

config_profiles out_profiles queue_configs sched_ports
root@tc16614-evm:/debug/qos# cat out_profiles
output profile 17 output q # 8174 red prob 0 enable 1 config profile 5 average q depth 0
output profile 18 output q # 8155 red prob 0 enable 1 config profile 5 average q depth 0
output profile 19 output q # 8154 red prob 0 enable 1 config profile 5 average q depth 0
output profile 20 output q # 8153 red prob 0 enable 1 config profile 5 average q depth 0
output profile 21 output q # 8152 red prob 0 enable 1 config profile 5 average q depth 0
output profile 22 output q # 8159 red prob 1310 enable 1 config profile 3 average q depth 0
output profile 23 output q # 8158 red prob 1310 enable 1 config profile 3 average q depth 0
output profile 24 output q # 8157 red prob 1310 enable 1 config profile 3 average q depth 0
output profile 25 output q # 8156 red prob 1310 enable 1 config profile 3 average q depth 0
output profile 26 output q # 8163 red prob 1310 enable 1 config profile 4 average q depth 0
output profile 27 output q # 8162 red prob 1310 enable 1 config profile 4 average q depth 0
output profile 28 output q # 8161 red prob 1310 enable 1 config profile 4 average q depth 0
output profile 29 output q # 8160 red prob 1310 enable 1 config profile 4 average q depth 0
output profile 30 output q # 8167 red prob 0 enable 1 config profile 6 average q depth 0
output profile 31 output q # 8166 red prob 0 enable 1 config profile 6 average q depth 0
output profile 32 output q # 8165 red prob 0 enable 1 config profile 6 average q depth 0
output profile 33 output q # 8164 red prob 0 enable 1 config profile 6 average q depth 0
output profile 34 output q # 8172 red prob 0 enable 1 config profile 7 average q depth 0
output profile 35 output q # 8172 red prob 0 enable 1 config profile 7 average q depth 0
root@tc16614-evm:/debug/qos#

```

The Drop scheduler config profiles can be seen by issuing the command `cat /debug/qos/config_profiles`. This is shown below

```

root@tc16614-evm:/debug/qos#
root@tc16614-evm:/debug/qos# cat config_profiles
drop cfg prof 3 unit flags 1 node 1 time const 5 tail thresh 131072 red low 32768 red high 65536 thresh recip 4194304
drop cfg prof 4 unit flags 1 node 1 time const 5 tail thresh 196608 red low 32768 red high 131072 thresh recip 1398101
drop cfg prof 5 unit flags 1 node 0 time const 0 tail thresh 65536 red low 0 red high 0 thresh recip 0
drop cfg prof 6 unit flags 1 node 0 time const 0 tail thresh 32768 red low 0 red high 0 thresh recip 0
drop cfg prof 7 unit flags 0 node 0 time const 0 tail thresh 0 red low 0 red high 0 thresh recip 0
root@tc16614-evm:/debug/qos#

```

Configuring Appleton Egress QoS in Linux

Appleton supports QoS in the QMSS firmware. As configured by default, the Linux kernel will route all traffic through QoS queue 0 (QMSS queue 8072). If this is not desirable, the kernel must be configured to classify egress traffic and direct it to the proper QoS queue. This document describes one possible configuration.

Baseband Configuration and Testing

To start, remove any previous traffic class configuration by removing the root qdisc (queuing discipline) from the interface:

```
# tc qdisc del dev eth0 root
```

Next, add DSMARK as the root qdisc for the interface. DSMARK supports the creation of multiple traffic classes, and traffic assigned to a class may be marked with a specified TOS value. In this case, we specify the maximum number of class indices as 8 (this value must be a power of 2), and assigned otherwise unidentified traffic to class 0. The qdisc will be identified with the numeric handle 1 (if a handle is not specified, a random one is assigned):

```
# tc qdisc add dev eth0 root handle 1 dsmark indices 8 default_index 0
```

Now we can configure the traffic classes within the DSMARK qdisc. In this example we configure five classes, one for each of the non-default QMSS QoS queues. For each class we can specify two values, a mask and a value. As traffic passes through the class, the existing TOS value from the packet is bitwise-ANDed with the mask, then bitwise-ORed with the value. This allows portions of the original TOS value to be passed through unchanged. Here we clear the DSCP value (the upper six bits) and replace them with a new value corresponding to the CS1, CS2, CS3, CS4, and CS7 class of service:

```

# tc class change dev eth0 classid 1:1 dsmark mask 0x03 value 0x20
# tc class change dev eth0 classid 1:2 dsmark mask 0x03 value 0x40
# tc class change dev eth0 classid 1:3 dsmark mask 0x03 value 0x60
# tc class change dev eth0 classid 1:4 dsmark mask 0x03 value 0x80
# tc class change dev eth0 classid 1:7 dsmark mask 0x03 value 0xe0

```

Since we've done nothing yet to direct traffic to one of these classes, all traffic still passes unchanged through the qdisc and ends up in QoS queue 0. To change that we can add filters. These filters can be quite complex, but for

simplicity we'll use the U32 classifier to match against the destination IP port. The first filter identifies traffic destined for IP port 5002, directs it to classid 1:1, which will cause it to be marked with DSCP class CS1. We also mark the packet to be mapped to driver queue 1, which the driver will pass on to QMSS queue 8073:

```
tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
```

```
u32 match ip dport 5002 0xffff \classid 1:1 \action skbedit queue_mapping 1
```

Similar filters can be added using other criteria. For example, these filters map destination IP port 5003 to class 1:2 (class CS2) and driver queue 2 (QMSS queue 8074) and so on.

```
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
```

```
u32 match ip dport 5003 0xffff \classid 1:2 \action skbedit queue_mapping 2
```

```
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
```

```
u32 match ip dport 5004 0xffff \classid 1:3 \action skbedit queue_mapping 3
```

```
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
```

```
u32 match ip dport 5005 0xffff \classid 1:4 \action skbedit queue_mapping 4
```

```
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 \
```

```
u32 match ip dport 5006 0xffff \classid 1:7 \action skbedit queue_mapping 5
```

Before running any tests, however, we have one more critical step. The DSMARK qdisc is not multi-queue aware, meaning that if driver queue 0 becomes full queuing will stop, even if driver queues 1 through 6 are not full. This is referred to as head of line blocking. To address this we can add the MULTIQ qdisc to the path; this qdisc is specifically designed to address this problem. Here's the command to do so:

```
# tc qdisc add dev eth0 parent 1: handle 2 multiq
```

With this configuration in place, it's simple to test the various QoS paths by setting up six iperf servers on a host, one server for each of the destination IP ports.

VLAN Configuration and Testing

Our example VLAN configuration builds on the above baseband configuration. VLAN traffic is also seen by the baseband queuing disciplines, so our example need only address the setting of the "P-bits" in the VLAN tag. First we need to set up a VLAN. This is done using the vconfig utility. The following command configures VLAN 3, and sets its IP address to 10.0.3.92/24:

```
# vconfig add eth0 3
# ifconfig eth0.3 inet 10.0.3.92 netmask 255.255.255.0
```

Since the P-bit field in the VLAN tag is only 3 bits wide, it can only have values in the range of 0 to 7. In contrast, the priority field in the Linux packet descriptor is an unsigned 32-bit value. To map packet priorities to VLAN P-bit values we again use the vconfig utility. Here we map priority 5432 to P-bit value 5:

```
# vconfig set_egress_map eth0.3 5432 5
```

RapidIO

The kernel RapidIO driver supports peer enumeration, peer discovery, message passing (required by RIONET, networking over RapidIO). The driver is implemented in file `drivers/rapidio/devices/keystone_rio.c`. The RapidIO support in u-boot is implemented in files `drivers/rapidio/keystone_rio.c` and `common/cmd_rio.c`.

Setting up to test RapidIO

Hardware Requirements

- * Two TCI6614 EVMs (Rev.3 or above)
- * Dual EVM Break Out Card (Ref. [[26][CI Dual EVM Break Out Card]])

Hardware Setup

1. Attach the two TCI6614 EVMs to the Dual EVM Break Out Card.
2. Connect an ethernet cable to **one of the two** EVMs. Do **NOT** connect ethernet cable to the second EVM.

IMPORTANT: CONNECT ETHERNET CABLE TO **ONLY ONE** EVM. **DO NOT CONNECT ETHERNET CABLE TO BOTH EVMS.** OTHERWISE IT WILL CREATE A LOOP AND MAY CAUSE THE OFFICE NETWORK PORT TO SHUTDOWN!

3. In the following, we'll call the EVM that has an ethernet cable attached EVM-A, the other EVM-B.

Sample Setup

```

      Wall eth port
      //////////////////////////////////////////////////
      -----
              |
              |
      |-----|
      | eth sw |
      |-----|
              |
              |
              |
      |---|      |---|
      |evm|      |evm|
      | A |      | B |
      |---|      |---|
      |=====|
      | Dual EVM Breakout Card |
      |=====|
  
```

4. Set both EVMs dip switch setting to

SW3	SW4	SW5	SW6	SW7	SW2
off	on	on	on	on	off
off	on	off	off	off	on
off	on	on	on	on	on

Refer to [[27][TMDXEVM6614LXE_QSG-rev3.0]] and [[28][TMDXEVM6614LXE_EVM_TRM_DVD_v12]] for more details.

5. Save u-boot to NAND on both EVMs *NAND Boot* section in *User Guide* for details).

6. Prepare env variables on both EVMs to boot uImage and

```
tc16614-evm.dtb
```

- * Add "riohdid=0" to EVM-A's bootargs env variable.
- * Add "riohdid=-1" to EVM-B's bootargs env variable.

7. Power off both EVMs.

Testing RIONET

1. Follow the steps in section **Hardware Setup** to set up the EVMs.
2. Power up and boot EVM-A into kernel **FIRST**.
3. This is required especially if NFS and/or tftp kernel boot method is used when EVM-B is boot up. Ethernet connection on EVM-B will go through the breakout card and the ethernet cable attached to EVM-A.

4. Power up and boot EVM-B into kernel.

5. After EVM-A RIO driver finishes the enumeration process, similar debug messages regarding rionet should be shown.

```
RIO: enumerate master port 0, RIO0 mport
device: '00:e:0001': device_add
bus: 'rapidio': add device 00:e:0001
device: 'rio0.1': device_add
keystone-rapidio 2900000.rapidio: RIO: port RIO0 host_deviceid 0
registered
bus: 'rapidio': add driver rionet
bus: 'rapidio': driver_probe_device: matched device 00:e:0001
with driver rionet
bus: 'rapidio': really_probe: probing driver rionet with device
00:e:0001
device: 'eth1': device_add
eth1: rionet Ethernet over RapidIO Version 0.2, MAC
00:01:00:01:00:00
Using 00:e:0001 (vid 0030 did b962)
driver: '00:e:0001': driver_bound: bound to device 'rionet'
bus: 'rapidio': really_probe: bound device 00:e:0001 to driver
rionet
```

6. After EVM-B RIO driver finishes the discovery process, similar debug messages regarding rionet should be shown.

7. On EVM-A, under Linux prompt, issue command "ifconfig eth1 192.168.1.1". Substitute eth1 for the interface that corresponds to MAC address 00:01:00:01:00:00 (see results from command "ifconfig -a")

8. On EVM-B, under Linux prompt, issue command "ifconfig eth1 192.168.1.2". Substitute eth1 for the interface that corresponds to MAC address 00:01:00:01:00:01 (see results from command "ifconfig -a")

9. On either EVM, issue command "ifconfig eth1" to see the configuration of the eth1 interface.

10. On EVM-A, issue command "ping 192.168.1.2". Make sure the ping receives response successfully.

11. On EVM-B, issue command "telnet 192.168.1.1". Make sure a telnet session can be opened successfully.

12. Ping and telnet can be performed on either EVM as long as the

appropriate remote IP address is used in the command.

Testing RIOBOOT

1. Apply the rioboot test utilities patch to the kernel source code and recompile the kernel for testing rioboot. Call this kernel **uImage_rioboot**.
2. Compile the rioboot test application riodio.c to generate the rioboot test app **riodio**.
3. EVM-A will boot into the **uImage_rioboot** kernel in the test and will need to have access to **uImage**, **tci6614-evm.dtb** and **riodio** from command line (see below).
4. EVM-B will be the EVM under test, i.e., it will receive the kernel image uImage over RapidIO direct IO and boot into that kernel.
5. Save the following files into the same directory in the file system (e.g NFS file system) that EVM-A will boot into.
 - * **uImage** - release kernel image
 - * **tci6614-evm.dtb** - the binary device tree file
 - * **riodio** - the test app
6. Follow the steps in section **Hardware Setup** to set up the EVMs.
7. Power on EVM-A and allow it to boot into the uImage_rioboot kernel.
8. Power on EVM-B and allow it to boot into u-boot (may need to hit any key to stop auto-boot).
9. On EVM-B, under the u-boot prompt, issue command "rioboot"
10. On EVM-B, similar messages should be shown


```
TCI6614 EVM # rioboot
rioboot: waiting for link up ...
KeyStone RapidIO driver v0.0, hdid=-1
SRIO link up: read remote reg offset 0 val 0xb9620030
RIO: port RIO0 ready
rioboot: waiting for image download ...
```
11. On EVM-A, similar messages should be shown


```
RIO: enumerate master port 0, RIO0 mport
device: '00:e:0001': device_add
bus: 'rapidio': add device 00:e:0001
device: 'rio0.1': device_add
keystone-rapidio 2900000.rapidio: RIO: port RIO0 host_deviceid
0 registered
bus: 'rapidio': add driver rionet
bus: 'rapidio': driver_probe_device: matched device 00:e:0001
with driver rionet
bus: 'rapidio': really_probe: probing driver rionet with device
00:e:0001
device: 'eth1': device_add
eth1: rionet Ethernet over RapidIO Version 0.2, MAC
00:01:00:01:00:00
Using 00:e:0001 (vid 0030 did b962)
```



```
driver: '00:e:0001': driver_bound: bound to device 'rionet'
bus: 'rapidio': really_probe: bound device 00:e:0001 to driver
rionet
```

12. Make sure the dev file /dev/rio0.1 exists on EVM-A by executing the command "ls /dev/rio0.1" without error.

13. On EVM-A, under Linux prompt, issue the following commands in order to send the image files to EVM-B over RapidIO DIO

```
cd /path/to/where/riodio, tci6614-evm.dtb and uImage/are/saved
./riodio -f uImage /dev/rio0.1 0x88000000          /* sends
uImage to location 0x88000000 on EVM-B */
./riodio -f tci6614-evm.dtb /dev/rio0.1 0x80000200 /* sends
dtb file to location 0x80000200 on EVM-B */
./riodio -v 0x88000000 /dev/rio0.1 0x80000000      /* tells
EVM-B u-boot where the uImage is loaded */
./riodio -v 0x80000200 /dev/rio0.1 0x80000004      /* tells
EVM-B u-boot where the dtb file is loaded */
```

14. EVM-B rioboot is monitoring the two locations 0x80000000 and 0x80000004. Once it picks up the values 0x88000000 and 0x80000200 at these two locations respectively. It will start auto-boot of the kernel received over !RapidIO DIO by running the command "bootm 0x88000000 - 0x80000200".

15. Hit any key on EVM-B to stop the auto boot.

16. Cycle power EVM-A and allow it to boot into the kernel (either release kernel or uImage_rioboot).

17. On EVM-B, under u-boot prompt, issue command "bootm 0x88000000 - 0x80000200" to boot the kernel loaded in memory over !RapidIO DIO.

18. Allow EVM-B to boot into the kernel.

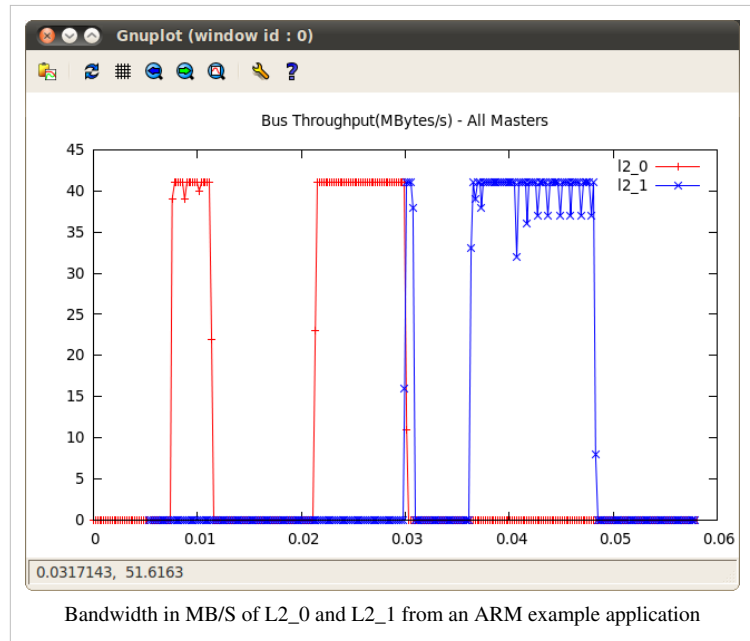
Kernel resource usage of the System

The ARM side resource usage is defined in the device tree file under `arm/arch/boot/dts/tci6614-evm.dts` in the linux kernel tree. System integrator needs to use this file for resource definition and the resources used shouldn't conflict that defined on the DSP side.

User Space Tools

User Space Profile Tool - `ctprof_srv`

For profiling internal device bus activity, `ctprof_srv` is provided in `/usr/bin` and works in conjunction with the `ctprof` utility on a host system for setup of standard profiling use cases, collection of profile data and processing of profile data. See `ctprof`^[29] for `ctprof` utility installation instructions, examples, use cases supported, and additional resources. The `gnuplot` screenshot(right) shows bus activity of the DSP's L2_0 and L2_1 memory spaces generated by `ctprof/ctprof_srv` using `ctprof_ex`. `ctprof_ex` is a simple user mode example application (provided in `/usr/bin`) that writes to the DSP's L2 memory space. The `gnuplot` screen was automatically launched from `ctprof` by simply selecting `gnuplot` formatted data.



User Space Application Libraries

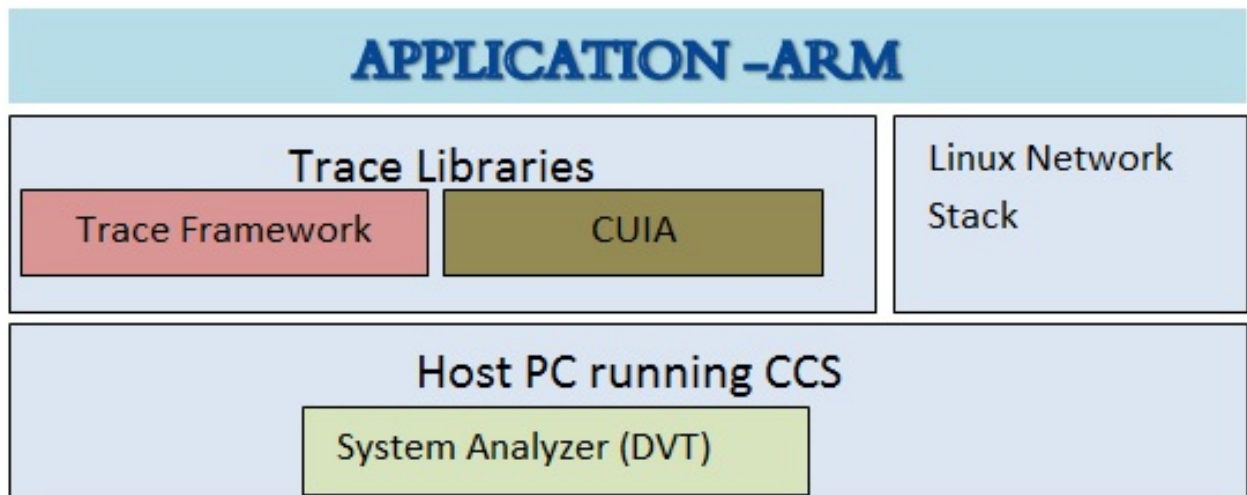
Instrumentation

Trace Framework

The trace framework enables the communication between DSP and ARM for the log messages. This component goes in conjunction with DSP's trace framework.

ARM Traceframework Library

The library provides necessary hooks to have single producer and multiple consumers. The static library `libtraceframework.a` is located under `/usr/bin/`. Any user space application has to link this library to use the trace framework. The source files for the traceframework library is located under `<pdsk_ver>/ti/instrumentation/traceframework` tree. Below picture shows the system level representation of trace framework.



CUIA Library

The CUIA package provides the APIs for the *LoggerStreamer* same as the UIA *LoggerStreamer* APIs. Please get the CUIA package from [http:// software-dl. ti. com/ dsp/ dsp_public_sw/ sdo_ccstudio/ UIA/ SCMCSDK/ cuia_1_00_00_09. tar. gz](http://software-dl.ti.com/dsp/dsp_public_sw/sdo_ccstudio/UIA/SCMCSDK/cuia_1_00_00_09.tar.gz) cUIA is an ARM component that is downloaded and included under devkit. The documentation for the cUIA are available under below locations.

1. cuia_1_00_00_##\docs (contains System analyzer User Guide and cUIA example guide)
2. cuia_1_00_00_##\docs\doc\index.html (API guide for CUIA)

User Space DMA

The user space DMA framework provides the zero copy access from user-space to packet dma channels. A new keystone specific udma kernel driver is added to the baseline which interfaces with the packet dma driver for this copy from user space to kernel (and vice versa). Apart from this, there is a user space library provided (libudma.a) which can be used by an application to create a udma channel, submit buffers to the channel, and receive callbacks for any buffers that the channel gets on the receive side. The user space udma code is checked into mcsdk-apps repository.

UDMA Files in the filesystem

The UDMA user space library is present in /usr/bin/libudma.a. This is the library that an application should include if they want to use the user space udma application. There is a small udma_test application which opens a single udma tx and a single udma rx channel and performs a loopback of 2M packets.

This udma_test application can be found in /usr/bin/udma_test.

Default udma channels setup in the kernel

As part of the integration of the UDMA into the baseline - the device tree was modified to setup 12 tx channels (udmatx0 - udmatx11) and 12 rx channels (udmarx0 - udmarx11) by default. Each of the TX channel is configured with a hardware queue number to use. Please refer to the device tree in the linux kernel tree (arch/arm/boot/dts/tci6614-evm.dts)

Using lldp daemon

The Link Layer Discovery Protocol (LLDP) is a vendor-neutral Link Layer protocol in the Internet Protocol Suite used by network devices for advertising their identity, capabilities, and neighbors on a IEEE 802 local area network, principally wired Ethernet.

lldp daemon is part of the root filesystem and is under /usr/bin/

The daemon is not enabled by default. To run the daemon use the following steps:

```
mkdir /var/run/lldpd
```

Run the daemon in the background:

Check different options available with lldpd, with "lldpd --help"

```
lldpd -c -s -e -f -k -dd & (There are different options to enable,
choose as per the user needs)
```

Now the daemon is running and if there are other neighbors, the log will show accordingly.

```
lldpctl will show any LLDP neighbors that have been added.
```

Capture kernel crash dump using kexec

When there is a kernel panic, and the system goes down there is a need to capture the coredump before resetting the system. This coredump could be later analyzed to debug the kernel panic and for other purposes. To achieve this with TCI-6614, a few user space utilities were added as part of the file system. The following sections provides details on what these utilities are and how they were used to enable this feature.

Kexec is a patch to the Linux kernel that allows you to boot directly to a new kernel from the currently running one. In the boot sequence, kexec skips the entire bootloader stage (the first part) and directly jumps into the kernel that we want to boot to. There is no hardware reset, no firmware operation, and no bootloader involved.

Kexec/Kdump Terminology

Some terms that will be used later on in this section are explained here:

Main-kernel: The kernel that is being used for regular use by the system.

Crash-kernel: The kernel that will be used for bringing up the system at the time of a kernel crash.

Crash-kernel-device tree: The device tree file that will be used by the crash kernel at the time of boot up after a panic from main kernel.

Kexec/Kdump User Space tools

Kexec-tools provides the /sbin/kexec binary that facilitates a new kernel to boot using the kernel's kexec feature either on a normal or a panic reboot. This package contains the /sbin/kexec binary and ancillary utilities that together form the userspace component of the kernel's kexec feature.

We use this tool only at the time of a kernel panic, but like the description says this can be used even to load a different kernel at a particular point in time.

User space kexec tools are available in gitweb at: [Kexec Git web](#) ^[30]

Kexec Usage

To use kexec to load a different kernel at the time of panic, do this: `kexec -p <path-to-zImage> --dtb=<path-to-device-tree-file>`

The `-p` option indicates that this rule will be applicable at the time of a panic only. The path to `zImage` is the `zImage` that will be loaded at the time of a panic, this is called `crashkernel`. The path to device tree is the device tree file that this `crashkernel` will use for boot up. This device tree will provide the command line argument for the new `crashkernel` to come up.

More options for kexec can be seen by doing: "kexec --help" from the shell prompt.

Reserving size of crashkernel

For `kexec-tools` to work, a separate area of memory will need to be used to load the `crashkernel`. This is done by adding `"crashkernel=32M@0x90000000"` to the bootargs of the main kernel's bootargs. If this is not done, the `"kexec -p"` command described previously will not work and will return an error saying `crashkernel` size is not allocated.

/proc/vmcore

When the main kernel panics, the `crashkernel` is triggered and when the `crashkernel` comes up, the `/proc/vmcore` entry will show the exact state of the previous kernel's view of the system. Capturing this file is needed to analyze the previous kernel panic. But this file will be really huge (because this is in effect, the previous kernel's view of the system). So there are some techniques done to get the exact coredump file that we want which are discussed in the next sections.

makedumpfile

With `kexec/kdump`, the memory image of the first kernel (called "main kernel") can be taken as `/proc/vmcore` while the second kernel (called "crash kernel") is running. `makedumpfile` makes a small `DUMPFIL` from the `/proc/vmcore` by compressing dump data or by excluding unnecessary pages for analysis, or both. `makedumpfile` needs the first kernel's debug information, so that it can distinguish unnecessary pages by analyzing how the first kernel uses the memory.

More information on `makedumpfile` can be found at: `makedump file` ^[31]

There is one particular usage of `makedumpfile` that we will be employing to capture relevant detail:

```
makedumpfile -E -d 31 /proc/vmcore/ coredump
the -E option means the output file coredump will be in ELF format.
(The default is in "crash" format, which needs "crash" utility)
-d 31 means all zero pages, cache pages, cache private, user and
free pages are removed when generating the coredump file. This is
needed otherwise the coredump file will become really huge.
```

makedumpfile option Description for "-d"

- 1 Zero pages
- 2 Cache pages
- 4 Cache private
- 8 User pages
- 16 Free pages

Refer for more information: `makedumpfile options` ^[32]

Crashkernel and crash kernel device-tree

Our kernel uImage file is relocatable and automatically relocates to 0x80008000, so to be used for crashkernel the zImage is preferred and tested.

At the time of kernel panic, the idea is that none of the hardware modules are reliable - because there may be some data traffic happening, and file system may be also be corrupted. So the idea is to bring up a crashkernel that does not have any of the hw modules enabled, and crash kernel will use a new filesystem. The "tc16614_evm_recovery_defconfig" has all TI_KEYSTONE disabled, networking disabled etc. Correspondingly we need a device tree file also to be used by the crashkernel. This device tree file need to provide the bootargs for the crashkernel, and list the "elfcorehdr=xxx" for the crashkernel to generate the /proc/vmcore entry.

This device tree source file is under <linux-repo>/arch/arm/boot/dts/tc16614-evm-recovery.dts.

The crashkernel can be built from our Linux kernel repo, by doing:

```
make tc16614_evm_recovery_defconfig
make -j8
make tc16614-evm-recovery.dtb
```

The output zImage and the dtb file is in <linux-repo>/arch/arm/boot

Init-script to run in the crash kernel's filesystem

The crash kernel will use a filesystem and the main expectation from this filesystem is to extract the vmcore information to a elf coredump. For this purpose, an init-script is written which gets executed automatically from the recovery-file-system. It looks like below:

```
#!/bin/sh
if [ -f /proc/vmcore ] ; then
    mkdir -p /mnt/boot2
    mount -t ubifs /dev/ubi0_2 /mnt/boot2
    if [ $? -eq 0 ] ; then
        if [ -f /usr/bin/makedumpfile ] ; then
            makedumpfile -E -d 31 /proc/vmcore
            /mnt/boot2/home/root/coredump.elf
            gzip -c /mnt/boot2/home/root/coredump.elf >
            /mnt/boot2/home/root/coredump.elf.gz
            rm -rf /mnt/boot2/home/root/coredump.elf
            sync
            reboot
        else
            echo "makedumpfile not found"
            exit 1
        fi
    else
        echo "mount unsuccessful"
        exit 1
    fi
fi
exit 0
```

Note: This script is part of the rootfs-recovery filesystem by default under: /etc/rc3.d/S09recoveryfs

How to enable crashdump kernel with SC-MCSDK on TCI6614-EVM

The default root-filesystem of SC-MCSDK contains the kexec and kdump user space utilities under /usr/sbin. The crashkernel and crash kernel's device tree binary file is pre-built and located under /usr/bin/crashdump. This kernel is built based on the "tci6614_evm_recovery_defconfig" located under arch/arm/configs of the linux repo. This device tree file is built based on the "tci6614-evm-recovery.dts" located under <linux-repo>/arch/arm/boot/dts/.

To verify the kexec way of loading a crashkernel and capturing main kernel panic information, use steps below:

From a UBI based NAND filesystem, do the following:

```
Boot the system with the following text added to the bootargs line in
u-boot:
```

```
"crashkernel=32M@0x90000000"
```

```
Boot the system.
```

```
cd /usr/bin/crashdump
```

```
kexec -p zImage --dtb=tci6614-evm-recovery.dtb
```

At this time, the main kernel is configured to use the zImage as the crash kernel.

Now invoke a panic, by manually building a simple panic kernel module.

A sample panic.c which can be built as kernel module is shown below:

```
/*
 * File name: panic.c
 */
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/sort.h>
#include <linux/bsearch.h>
#include <linux/xfrm.h>
#include <net/net_namespace.h>
#include <linux/list.h>
#include <linux/hash.h>
static int __init panic_init(void)
{
    printk("calling panic()\n");
    panic("panic has been called");
    return 0;
}
module_init(panic_init);
```

Now do:

```
insmod panic.ko
```

This will reboot the board and:

1. Come up to the new dump-kernel/recovery-fs (because the crash kernel device tree file uses the recovery

filesystem)

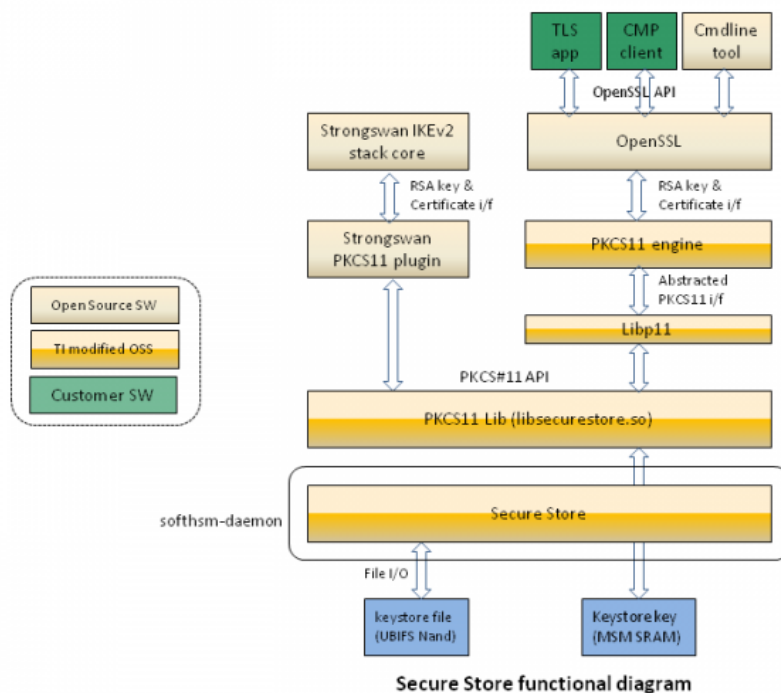
2. Automatically copy the elf-core-dump (the initscripts of recovery filesystem checks for /proc/vmcore entry and generates the coredump elf)

3. Reboots back to the original kernel/fs. (Again, the initscript of recovery fs does a reboot after copying the coredump)

Now: from /home/root directory, check if there is a coredump.elf.gz. This can be transferred to the linux pc, and analyzed with gdb against a vmlinux file.

```
arm-linux-gnueabi-gdb <vmlinux-file> --core=coredump.elf
```

Secure storage for storing long-term secrets



Overview

Secure store is a software representation of a hardware token, which provides facility to securely store and retrieve longterm device secrets in NAND.

This have following features

- The secure store filesystem is encrypted and stored in NAND with a AES128 bit key called *securestore encryption key*
- The *securestore encryption key* is stored in NAND with a separate UBI volume
 - The *securestore encryption key* is stored as a key blob in NAND with a header and checksum to check for integrity
 - In non-secure device the key blob is stored in plain text
 - In secure device the key is encrypted and signed by boot keys and stored in NAND
- The encrypted and signed secure store file is stored in NAND on two separate UBI volumes to increase reliability
- Secure store services are provided by a daemon process called *softhsm-daemon*
- Application access to secure store is through PKCS#11 APIs provided by the secure store library(*libsecstore.so*)

- This library serializes PKCS11 API requests and sends to *softhsm-daemon*.
- The communication between application & daemon is using Unix domain socket IPC.
- *softhsm-daemon* processes the PKCS11 requests and provides response to the application
- Applications in this context will be *strongswan*, *libp11* etc.
- OpenSSL applications like *TLS client/server* & *CMPv2 client* will be accessing secure store services via the OpenSSL engine interface.

Following is the startup sequence wrt. secure store

1. U-Boot reads the *securestore encryption key* blob from NAND UBI volume
 1. In secure device: Authenticated/Decrypted key blob is placed in internal memory
 2. In non-secure device: The key blob is in plain text, and placed in internal memory
2. During Linux kernel boot, the UBIFS secure store volumes are mounted in filesystem using fstab
3. The Linux init script starts softhsm-daemon
4. The softhsm-daemon reads/parses softhsm config file from Linux filesystem for the following
 1. location and size of internal memory
 2. location of *securestore encryption key*
 3. securestore file paths
 4. Unix domain socket names to be used for IPC
5. The softhsm-daemon waits on a Unix domain socket, the socket name is provided in the softhsm config file

Internal memory for secure store

Secure store uses internal memory (MSMC SRAM) to keep the working copy of secure file system. 64KBytes of MSMC SRAM starting from address *0x0c000000* is used by secure store and this memory range must not be used by any other DSP or ARM application. If configured, Secure store can also use internal memory to maintain heap for dynamic memory allocations of crypto library(OpenSSL), in which case it will consume 38KB of memory from the reserved 64KB.

Secure store ubifs volumes

- Following UBIFS volumes used for secure store, the volumes names are as follows
 - *securedbv0*, *securedbv1*: Two volumes to store encrypted and integrity protected filesystem blob
 - Securestore encryption key file blob is present in boot volume
- Sample UBIFS config file is provided in *<release_folder>/sc_mcsdk_linux_<version>/images/ubinize.cfg*

Creating securestore encryption key blob

A raw text or binary key file can be converted to the format required by secure store using *securestore-formatkey.py* script provided in *<release_folder>/sc_mcsdk_linux_<version>/bin* file. Note that the key has to be a 128bit AES key.

```
<release_folder>/sc_mcsdk_linux_<version>/bin/securestore-formatkey.py --input
./aeskey.txt --output securedb.key.bin
```

- Note: The u-boot will use "securedb.key.bin" file name to access load the key from volume "boot"
- Please see the script help *<release_folder>/sc_mcsdk_linux_<version>/bin/securestore-formatkey.py --help* for other usage details
- In case of secure board, this file has to be processed further, please see MCSDK-SECDEV user guide for further details

Creating securestore filesystem ubifs volume image

The default UBI volumes (securedbv[0-1].ubifs.img) are provided in the release package. The following sequence of commands will create securestore filesystem image ubifs partition export

```
PATH=<release_folder>/sc_mcsdk_linux_<version>/bin:$PATH
rm -rf ./securedbv0 ./securedbv1 mkdir ./securedbv0 mv <path to filesystem blob> ./securedbv0/securedb mkfs.ubifs
-r ./securedbv0 -F -o securedbv0.ubifs.img -m 2048 -e 126976 -c 22
mv ./securedbv0 ./securedbv1 mkfs.ubifs -r ./securedbv1 -F -o securedbv1.ubifs.img -m 2048 -e 126976 -c 22
```

Secure store token initialization

An empty, encrypted & initialized token is present in the default filesystem. In case it is needed to re-initialize the token, it can be done on the target using *softhsm-util* command. The default PINs are *1234*. NOTE: Secure store has a restriction that the SO PIN & User PIN be set the same. *softhsm-util --init-token --slot 0 --label token-0 --module /usr/lib/softhsm/libsecstore.so.1*

softhsm-util --show-slots --module /usr/lib/softhsm/libsecstore.so.1 For more information see help for the tool *softhsm-util --help* NOTE: A reboot of the system is needed after initializing the token.

Using Strongswan with secure storage

Storing credentials

- RSA Key pair
 - RSA key pair to be used by Strongswan during IKE negotiation has to be generated and stored on the secure store.
 - Key generation/storage can be done through OpenSSL command line or via engine API programmatically.
- X.509 Certificate
 - End entity certificate is issued by a certification authority, and needs to be brought into the secure store
 - In a production system this would be done by CMPv2 client.
 - For lab setup
 - Certificate issuance can be done using command line tools provided by Strongswan or OpenSSL.
 - This process will require access to the RSA public key of the end entity. Secure store solution provides an OpenSSL engine command to retrieve the Public key from store.
 - The issued certificate can then be written to secure store using OpenSSL engine commands.
- CA certificate also needs to be in the secure store and can be written using OpenSSL Engine commands.

PKCS#11 plugin configurations

To use the strongswan pkcs#11 plugin, PKCS#11 module has to be configured in */etc/strongswan.conf*.

Configuration Keys specific to pkcs#11 plugin are explained in the PKCS11Plugin wiki ^[33].

Default configuration provided will be: `libstrongswan { plugins { pkcs11 { modules { secstore { path = /usr/lib/softhsm/libsecstore.so.1 } } } } }`

PIN configurations

Soft token will have a static PIN. Following wiki explains the format for configuring a static PIN in the file */etc/ipsec.secrets* PIN Secret ^[34].

Private Key configuration

The RSA private key to be used from the smartcard is specified as part of the PIN configuration with the format: *%smartcard[<slotnr>[@<module>]]:<keyid>* e.g.

```
: PIN %smartcard0@secstore:04 1234
```

Certificate configuration

Certificates stored on smart cards will get loaded automatically when the IKEv2 daemon is started. You don't have to specify *leftcert=%smartcard* in *ipsec.conf* (it actually will fail if you do so). Instead the first certificate matching the "leftid" parameter is used.

CA certificates are also automatically available as trust anchors without the need to copy them into the */etc/ipsec.d/cacerts/* directory.

```
conn alice left=158.218.103.42 leftprotoport=udp/5000 #leftcert=bobCert.der
leftid="C=US, O=TestInc, CN=bob" right=158.218.103.108 rightid="C=US,
O=TestInc, CN=alice" rightprotoport=udp/5000 keyexchange=ikev2 type=tunnel
lifetime=24h auto=start
```

PKCS11 engine commands for OpenSSL

Following commands are provided by the OpenSSL engine (engine_pkcs11) to interface with secure storage

- **SO_PATH:** Specifies the path to the 'pkcs11-engine' shared library
- **MODULE_PATH:** Specifies the path to the pkcs11 module shared library
- **PIN:** Specifies the pin code
- **VERBOSE:** Print additional details
- **QUIET:** Remove additional details
- **INIT_ARGS:** Specifies initialization arguments to the pkcs11 module
- **LIST_OBJS:** List the objects from token
- **STORE_CERT:** Store X.509 certificate to token (DER format)
- **GEN_KEY:** Generate & store RSA key pair to token
- **DEL_OBJ:** Delete objects from token
- **GET_PUBKEY:** Get Public Key from token (PEM format)

Following is the listing of these commands

- Command to load the engine dynamically and set the PIN

```
engine -vvvv dynamic -pre SO_PATH:/usr/lib/engines/engine_pkcs11.so -pre ID:pkcs11 -pre
LIST_ADD:1 -pre LOAD -pre MODULE_PATH:/usr/lib/softhsm/libsecstore.so.1 -pre "VERBOSE" -pre
"PIN:1234"
```

- Command to enable verbose

```
engine pkcs11 -pre "VERBOSE"
```

- Command to set the PIN

```
engine pkcs11 -pre "PIN:1234"
```

- Command to list the objects on slot 0

```
engine pkcs11 -pre "LIST_OBJS:0"
```

- Command to store certificate to token

```
engine pkcs11 -pre "STORE_CERT:slot_<slot>:id_<id>:label_<label>:cert_<filename>"
```

e.g.

```
engine pkcs11 -pre "STORE_CERT:slot_0:id_03:label_tcert:cert_caCert.der"
```

- Command to generate a RSA key pair

```
engine pkcs11 -pre "GEN_KEY:slot_<slot>:size_<size>:id_<id>:label_<label>"
```

where <size> is the RSA key size in bits

e.g.

```
engine pkcs11 -pre "GEN_KEY:slot_0:size_2048:id_04:label_tkey"
```

- Command to get the Public key from token

```
engine pkcs11 -pre "GET_PUBKEY:slot_<slot>:id_<id>:label_<label>:key_<filename>"
```

e.g.

```
engine pkcs11 -pre "GET_PUBKEY:slot_0:id_04:label_tkey:key_tkey.pem"
```

- Command to delete an object from the token

```
engine pkcs11 -pre "DEL_OBJ:slot_<slot>:type_<type>:id_<id>:label_<label>"
```

where <type> is the object type (privkey/pubkey/cert)

e.g. to delete certificate

```
engine pkcs11 -pre "DEL_OBJ:slot_0:type_cert:id_03:label_tcert"
```

e.g. to delete a public key

```
engine pkcs11 -pre "DEL_OBJ:slot_0:type_pubkey:id_04:label_tkey"
```

e.g. to delete a private key

```
engine pkcs11 -pre "DEL_OBJ:slot_0:type_privkey:id_04:label_tkey"
```

Multi interface support in Linux kernel

To support both the ethernet ports available in the TCI6614 chip from the Linux kernel side, a few changes were added. On the regular TCI6614-EVM only one ethernet port is connected to a phy and visible externally via the RJ-45 connection. Hence the default set of images that is released as part of the SC-MCSDK release will support the single interface mode by default. But, if a user wants to try the multi-interface mode setup (can be done by connecting two TCI6614's via a back-plane), there is a change that needs to be done.

Kernel uses a device tree file to obtain all hardware related parameters and the default provided as part of the tci6614-evm-ubifs.ubi supports only the single interface mode. But as part of the release software, a new device tree file: tci6614-evm-multi-if.dtb is provided.

A user can replace the tci6614-evm.dtb with the tci6614-evm-multi-if.dtb (name should be changed to tci6614-evm.dtb) on the NAND device. Please follow instructions provided in the "Updating Boot volume images from Linux kernel" section to replace the single interface device tree file to a multi-interface device tree file.

Once the user replaces the tci6614-evm.dtb with the tci6614-evm-multi-if.dtb, there will be two interfaces from the kernel's perspective - eth0 and eth1. On the EVM, the port that is hooked upto the RJ 45 port will become the eth1 interface.

Bridging setup

For certain use cases pertaining to the small cell development, there is a need to setup bridging in the linux kernel with the multiple interface environment.

The following sequence of steps need to be done from the EVM to setup bridging:

```
ifconfig eth0 0.0.0.0 up
ifconfig eth1 0.0.0.0 up
brctl addbr br0
brctl addif br0 eth0
brctl addif br0 eth1
ifconfig br0 up
```

- Note: For br0 to obtain an ip-address, the dhcpclient can be used. Use "/sbin/dhclient br0".

Reload images from DDR during reboot

When a user types "reboot" from the shell prompt to reset the system, the DDR contents remain intact. What this allows us to do is to store a copy of the kernel/dtb contents in DDR, and use this location to reload the system instead of reloading back from NAND to DDR. This is an optional feature and can be configured with a few command changes in u-boot.

So for the first time when user flashes the image – it will fetch the kernel and dtb from NAND, but subsequent reboots should use the contents from the DDR (as long as they are not corrupted).

It is left to user's discretion to allocate an area in DDR that will not be corrupted. But the fail safe mode - is the images will be reloaded from NAND if loading from DDR fails.

Use the following two commands in u-boot, to set this up:

```
setenv bootcmd 'run reload_from_ddr; ubi part ubifs; ubifsmount
boot; ubifsload 0x98000000 uImage; ubifsload 0x97000200
tci6614-evm.dtb; bootm 0x98000000 - 0x97000200'
setenv reload_from_ddr 'bootm 0x98000000 - 0x97000200'
```

So, the boot command will try to reload the kernel and dtb file from the 0x98000000 and 0x97000200 addresses. If that fails (checksum failure), the images will be reloaded from NAND. User may choose different addresses and add ability to protect that area of DDR.

DSP

Introduction

The Small Cell Multicore Software Development Kit (SC-MCSDK) provides the core foundational building blocks that facilitate application software development on TI's high performance and multicore SoC. The foundational components include:

- SYS/BIOS which is a light-weight real-time embedded operating system for TI devices
- Chip support libraries, drivers, and basic platform utilities
- Interprocessor communication for communication across cores and devices
- Basic networking stack and protocols
- Optimized application-specific and application non-specific algorithm libraries
- Debug and instrumentation
- Bootloaders and boot utilities

- Demonstrations and examples

The purpose of this *User's Guide* is to provide more detailed information regarding the software elements and infrastructure provided with SC-MCSDK. SC-MCSDK pulls together all the elements into demonstrable multicore applications and examples for supported EVMs. The objective being to demonstrate device, platform, and software capabilities and functionality as well as provide the user with instructive examples. The software provided is intended to be used as a reference when starting their development.

Note: It is expected the user has gone through the *EVM Quick Start Guide (TBD)* provided with their EVM and have booted the out-of-box demonstration application flashed on the device. It is also assumed the user has gone through the → SC-MCSDK Getting Started Guide and have installed both CCS and the SC-MCSDK.

API and LLD User Guides

API Reference Manuals and LLD User Guides are provided with the software. You can reference them from the Eclipse Help system in CCS or you can navigate to the components *doc* directory and view them there.

Tools Overview

The following documents provide information on the various development tools available to you.

Document	Description
CCS v5 Getting Started Guide ^[35]	How to get up and running with CCS v5
XDS560 Emulator Information ^[36]	Information on XDS560 emulator
XDS100 Emulator Information ^[37]	Information on XDS100 emulator
TMS320C6000 Optimizing Compiler v 7.3 ^[38]	Everything you wanted to know about the compiler, assembler, library-build process and C++ name demangler.
TMS320C6000 Assembly Language Tools v 7.3 ^[39]	More in-depth information on the assembler, linker command files and other utilities.
Multi-core System Analyzer ^[40]	How to use and integrate the system analyzer into your code base.
Eclipse Platform Wizard ^[41]	How to create a platform for RTSC. The demo uses CCSv4 but the platform screens are the same in CCSv5.
Runtime Object Viewer ^[42]	How to use the Object Viewer for Eclipse Based Debugging.

Hardware - EVM Overview

The following documents provide information about the EVM.

Document	Description
Introducing the C66x Lite EVM Video ^[43]	Short video on the C66x Lite Evaluation Module, the cost-efficient development tool from Texas Instruments that enables developers to quickly get started working on designs for the C6670, C6672, C6674, and C6678 multicore DSPs based on the KeyStone architecture.
TMDXEVM6614 documentation and support (TBD)	Discusses the technical aspects of your EVM including board block diagram, DIP Switch Settings, memory addresses and range, power supply and basic operation.

Hardware - Processor Overview

The following documents provide information about the processor used on the EVM.

Document	Description
TMS320TCI6614 Data Manual ^[44]	Data manual for specific TI DSP

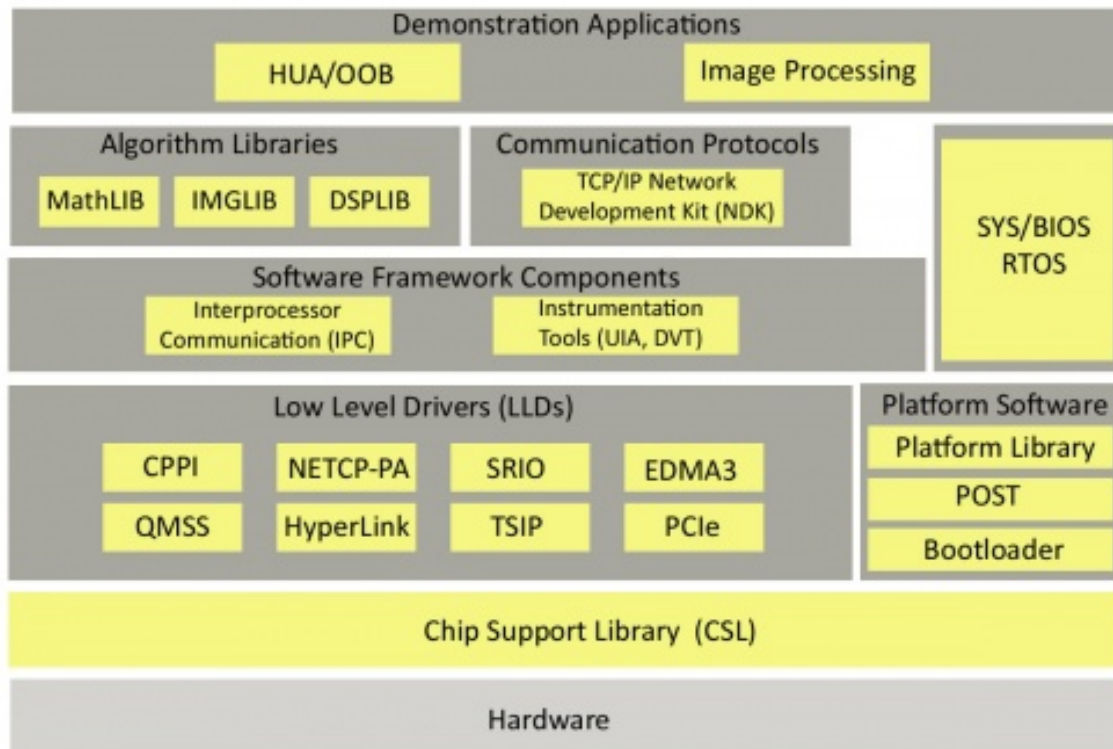
Related Software

This section provides a collection links to additional software elements that may be of interest.

Link	Description
C6x DSP Linux Project ^[45]	Community site for C6x DSP Linux project
Security Accelerator LLD ^[46]	Download page for Security Accelerator (SA) low level driver
Telecom Libraries ^[47]	TI software folder for information and download of Telecom Libraries (Voice, Fax, etc) for TI processors.
Medical Imaging Software Tool Kits ^[48]	TI software folder for information and download of medical imaging software tool kits for TI processors.

Software Overview

The SC-MCSDK is comprised of the foundational software infrastructure elements intended to enable development of application software on TI high-performance and multicore DSPs.



As can be seen in the architecture diagram, the SC-MCSDK consists of those components in the yellow boxes. Specifically they are:

- **Chip Support Library**
- **Low Level Drivers**
- **Platform Software consisting Platform Library**
- **SYS/BIOS**
- **Software Framework Components including Interprocessor Communication Package and Instrumentation Tools**
- **Network Developer's Kit (NDK) Package**

Platform Development Kit (PDK)

The Platform Development Kit (PDK) is a package that provides the foundational drivers and software to enable the device. It contains device specific software consisting of a Chip Support Library (CSL) and Low Level Drivers (LLD) for various peripherals; both the CSLs and LLDs include example projects and examples within the relevant directories which can be used with CCS. It also contains the transport (NIMU), platform library, platform/EVM specific software, applications, CCS configuration files and other board specific collaterals.

Operating System Adaptation Layer (OSAL)

Various components in the PDK support OSAL callbacks that allow applications to tailor common operations to their specific needs. The implementation of these callbacks is the applications responsibility. Typical callbacks include:

- Memory Management
- Critical Sections
- Cache Coherency

See the file `platform_osal.c` in the demos and examples. This file can be used as a basic starting point.

Resource Management

This section covers the resource management implementations delivered as part of the SC-MCSDK PDK package.

Platform Resource Manager

The Resource Manager defines a set of APIs and definitions for managing platform resources (e.g. Interrupts, Hardware semaphores, etc) and provides example code for initializing and using the PA, QMSS and CPPI subsystems.

The Resource Manager definitions are present in `pdktci####_#_#_#/packages/ti/platform/resource_mgr.h` header file. This header file is included by the demos/example, NIMU and platform library.

The example implementation is included in the SC-MCSDK demo and example applications in the `resourcemgr.c/osal.c` source files.

The following Linker Sections are used by the `reourcemgr.c` file and would need to be included in the application linker map or `.cfg` file.

- `.resmgr_memregion` = Contains QMSS descriptors region
- `.resmgr_handles` = Contains CPPI/QMSS/PA Handles
- `.resmgr_pa` = Contains PA Memory

Resource Manager (RM) LLD

The Resource Manager (RM) LLD allows a system integrator to specify DSP initialization and usage permissions for device resources. The RM lets the system integrator mark a clear separation between resources available for use by the DSPs and those available for use by Linux running on the ARM. When included in a system the RM LLD allows supported LLDs to callout to the RM LLD for resource permission verification.

Currently, RM LLD support is in the following LLDs:

- QMSS
- CPPI
- PA



Note: The API additions to the QMSS, CPPI, and PA LLDs to support the RM LLD are fully backwards compatible. No modifications are required to existing applications integrating the new QMSS, CPPI, and PA LLD versions in order to maintain existing behavior. The QMSS, CPPI, and PA LLDs consider RM callouts disabled by default.

Managed Resources

The RM allows initialization and usage permissions to be specified for the following resources:

QMSS

- PDSP Firmware Download
- Queues
- Memory Regions
- Linking RAM Control (RAM0/1 Base address programming)
- Linking RAM Indices
- Accumulator Channels
- QOS Clusters
- QOS Queues

CPPI

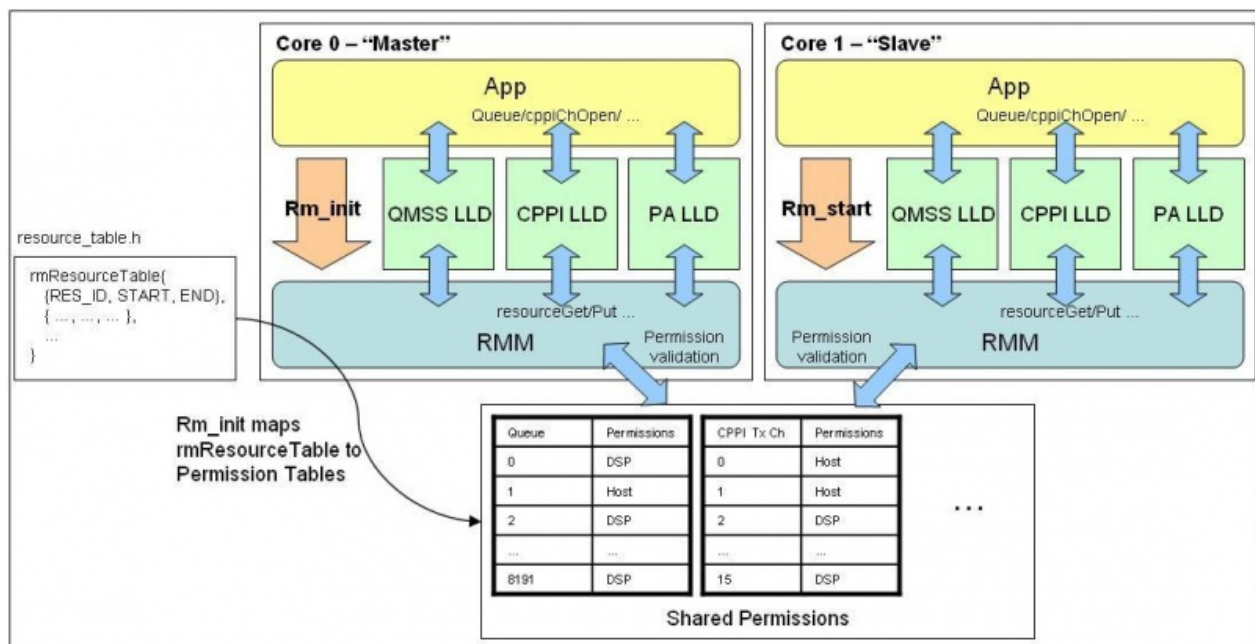
- Transmit Channels
- Receive Channels
- Receive Flows

PA

- Firmware Download
- Look-up Tables (The entire table, not individual entries)

RM Architecture Overview

The following figure provides a graphical representation of how the RM LLD fits into an application.



The Resource Manager LLD sits under the hood of the QMSS, CPPI, and PA LLDs to perform permission checks on the initialization and usage of resources. The RM LLD contains a permission field for each tracked QMSS, CPPI, and PA LLD resource. The permission fields contains an initialization and a usage bit for each DSP in the system. The permission fields are global and are required to be placed in the global address space for the device. Whenever a tracked LLD resource is specified for use by the application through the QMSS, CPPI, or PA LLD APIs the LLD internally sends a resource permission check request to the RM LLD. The RM LLD uses the resource data, a resource identifier and the resource value, to index the internal permission tables. When the resource entry is found the DSP number is used to extract the initialization and usage information for the resource. This information is

returned to the requesting LLD. Based on the RM LLD response, resource approved or denied, the LLD either continues normal operation or returns a resource denied failure for the application to act upon.

The APIs used by the RM LLD and the QMSS, CPPI, and PA LLDs are internal APIs that are not meant to be used by an application. The application gets a RM handle for each DSP from the RM LLD after it has initialized and started the RM. The RM handle contains RM LLD resource permission internal API information that is shared between the RM and the other LLDs. The application must provide the RM handle to each LLD for each DSP operating in the system. Providing the RM handle to the LLDs effectively registers the RM with the LLD and informs the LLD that it should check initialization and usage permissions for all covered resources.

It is the job of the system integrator, or application developer, to set the LLD resource permissions prior to compile time. A resource table must be defined and passed as an argument to the "master" DSP core via the RM initialization function. The RM initialization function will parse the resource table and transfer all defined resource permissions to the internal resource permission tables in global memory. Upon completion of the transfer the "master" core will write to a global synchronization object, signalling to the "slave" DSP cores that the internal permission tables have been populated. Each "slave" core will then invalidate the entire permission table so that no further cache invalidate operations need to be performed when checking resource permissions in the data path. The upfront cache invalidate operation is possible because the RM LLD does not allow dynamic resource permission modifications. The permissions defined by the system integrator and loaded during RM initialization are static throughout the system up-time.

Using the RM LLD

Defining the Resource Table

The first step in integrating the RM LLD is defining the resource table that specifies the resource permissions for the DSPs. The resource table is an array of resource structures. Each structure specifies a resource type, the start and end range for the resource and the initialization and usage permissions for the resource for each DSP. A default resource table is delivered with the RM LLD under the `resource_table/` directory. The default resource table is based on the target PDK device and gives all DSPs full permissions to all supported LLD resources.

If some resources are going to be used by another processor on the device, say Linux running on an ARM, there are two ways the system integrator can use to define the resource table. The first method, the system integrator should specify all resources that will be used by the DSPs in the resource table. Any resources that are not specified in the resource table are initialized to deny access to all DSPs by the RM LLD. The second method, the system integrator can specify all resources in the system but must make sure the resources that are used by a non-DSP processor give the DSP no permissions. The first method is preferred, and highlighted in this guide, because it provides a clear picture of the resources given to DSPs. The first method is also easier to modify if the used resources change.

A simple example for a resource table is provided below. The resources assigned in the example are not from a larger, validated example. If used to create an example the resources assigned permissions are not enough for a system to function properly. The below code is meant as a teaching example only.

```
/* The Rm_Resource structure and the resource identifiers used are defined in
resource_table_defs.h */

/** @brief RM LLD resource table permissions */ const Rm_Resource simpleResourceTable[] = {

/* Magic Number structure to verify RM can read the resource table */

{
/** DSP QMSS Firmware access */
RM_RESOURCE_MAGIC_NUMBER,
/** No start range */
```

```
0u,  
/** No end range */  
0u,  
/** No init permissions */  
0u,  
/** No use permissions */  
0u,  
,  
  
/* QMSS Resource Definitions */  
  
{  
/** DSP QMSS PDSP Firmware access */  
RM_RESOURCE_QMSS_FIRMWARE_PDSP,  
/** PDSP start range */  
0,  
/** PDSP end range */  
1,  
/** Full permissions for all DSPs */  
RM_RESOURCE_ALL_DSPS_FULL_PERMS,  
/** Full use permissions for all DSPs */  
RM_RESOURCE_ALL_DSPS_FULL_PERMS,  
,  
{  
/** DSP QMSS queue access */  
RM_RESOURCE_QMSS_QUEUE,  
/** Queue start range*/  
2000,  
/** Queue end range */  
3000,  
/** Full permissions for all DSPs */  
RM_RESOURCE_ALL_DSPS_FULL_PERMS,  
/** Full use permissions for all DSPs */  
RM_RESOURCE_ALL_DSPS_FULL_PERMS,  
,  
{  
/** DSP QMSS accumulator channels */  
RM_RESOURCE_QMSS_ACCUMULATOR_CH,  
/** Accumulator channel start range*/  
0,  
/** Accumulator channel end range */  
7,  
/** Full permissions for all DSPs */  
RM_RESOURCE_ALL_DSPS_FULL_PERMS,  
/** Full use permissions for all DSPs */  
RM_RESOURCE_ALL_DSPS_FULL_PERMS,  
,  
}
```

```

{
/** DSP CPPI QMSS tx channels */
RM_RESOURCE_CPPI_QMSS_TX_CH,
/** CPPI QMSS tx channel start range*/
0,
/** CPPI QMSS tx channel end range */
2,
/** Full permissions for all DSPs */
RM_RESOURCE_ALL_DSPS_FULL_PERMS,
/** Full use permissions for all DSPs */
RM_RESOURCE_ALL_DSPS_FULL_PERMS,
},
{
/** DSP CPPI QMSS rx channels */
RM_RESOURCE_CPPI_QMSS_RX_CH,
/** CPPI QMSS rx channel start range*/
0,
/** CPPI QMSS rx channel end range */
2,
/** Full permissions for all DSPs */
RM_RESOURCE_ALL_DSPS_FULL_PERMS,
/** Full use permissions for all DSPs */
RM_RESOURCE_ALL_DSPS_FULL_PERMS,
},
{
/** DSP CPPI QMSS rx flows */
RM_RESOURCE_CPPI_QMSS_FLOW,
/** CPPI QMSS rx flow start range*/
0,
/** CPPI QMSS rx flow end range */
2,
/** Full permissions for all DSPs */
RM_RESOURCE_ALL_DSPS_FULL_PERMS,
/** Full use permissions for all DSPs */
RM_RESOURCE_ALL_DSPS_FULL_PERMS,
},

/* Final entry structure for RM to find the last entry of resource
table */

{
/** Final entry */
RM_RESOURCE_FINAL_ENTRY,
/** No start range*/
0u,
/** No end range */
0u,
/** No init permissions */

```

```

Ou,
/** No use permissions */
Ou,
}

};

```

- **RM_RESOURCE_MAGIC_NUMBER** - The magic number entry should **ALWAYS** be the first entry in the resource table. This value is used by the RM to validate the resource table prior to using it to populate the internal permission tables.
- **RM_RESOURCE_QMSS_FIRMWARE_PDSP** - This entry gives all DSPs permission download the firmware for QMSS PDSP0 and PDSP1.
- **RM_RESOURCE_QMSS_QUEUE** - This entry gives all DSPs permission to initialize and use QMSS queues 2000 through 3000.
- **RM_RESOURCE_QMSS_ACCUMULATOR_CH** - This entry gives all DSPs permission to initialize and use QM Accumulator channels 0 through 7.
- **RM_RESOURCE_CPPI_QMSS_TX_CH** - This entry gives all DSPs permission to initialize and use CPPI QM transmit channels 0 through 2.
- **RM_RESOURCE_CPPI_QMSS_RX_CH** - This entry gives all DSPs permission to initialize and use CPPI QM receive channels 0 through 2.
- **RM_RESOURCE_CPPI_QMSS_FLOW** - This entry gives all DSPs permission to initialize and use CPPI QM flows 0 through 2.
- **RM_RESOURCE_FINAL_ENTRY** - The final entry should **ALWAYS** be the last entry in the resource table. This value is used by the RM to stop parsing the resource table.

The RM LLD will read this resource table and transfer the permissions specified to the internal permission tables. All resources that have been left unspecified will be assigned deny permissions for all DSPs.

Placing the RM LLD Permission Tables

The RM LLD internal permission tables contain the permissions for all DSP cores. Therefore, the tables are global and placed into the ".rm" memory section. Similar to the QMSS ".qmss", and CPPI ".cpqi" sections, this memory section **MUST** be manually placed in shared memory (MSMC or DDR) via the linker command file.


Initializing the RM LLD

The RM LLD has two initialization APIs that are used based on the context in which the application runs. The `Rm_init` API is the primary initialization routine and should be called on the "master" DSP core. The `Rm_start` routine should be called on all other "slave" DSP cores. Both APIs should be called prior to any other LLD init/start routines. The `Rm_init` function should be passed a pointer to the resource table. The `Rm_init` function will validate and parse the resource table, using it to populate the internal permission tables. When the RM completes populating the internal permissions table the `Rm_init` will write to a global synchronization object to sync with all slave DSP cores who have invoked the `Rm_start` API. The slave cores that have invoked `Rm_start` will stop spinning once the global synchronization has been written. At this time `Rm_start` will invalidate all internal permission tables so that no further cache invalidate operations need to be performed when checking resource permissions in the data path. The upfront cache invalidate operation is possible because the RM LLD does not allow dynamic resource permission modifications. The permissions defined by the system integrator and loaded during RM initialization are static throughout the system up-time.

Registering RM with LLDs

The RM must be registered with a LLD in order for the LLD to perform resource permission checks. If the RM is not registered with a LLD the LLD will operate as if the RM LLD is not there. This maintains full backwards compatibility with existing applications not using the RM LLD. In order to register the RM LLD with LLDs the

following steps should be taken

- Get a `Rm_Handle` via the `Rm_getHandle` API on each DSP that uses the RM LLD.
- Register the RM LLD with other LLDs by passing the `Rm_Handle` to the LLD's `_startCfg` API. Again, this should be performed on all DSP cores using the RM LLD.  **Note:** The master core for the QMSS LLD should have the `Rm_Handle` registered via the `Qmss_init` API. This is done by passing the `Rm_Handle` inside the `Qmss_GlobalConfigParams` structure.

When a LLD has registered with the RM the LLD will invoke permission check callouts to the RM whenever supported resources are initialized or requested. A permission denied or approved response will be given back to the invoking LLD based on the permissions stored in the RM LLD for the resource.

RM LLD Initialization Example

The following code snippet shows how to initialize the RM LLD and register it with other LLDs on "master" and "slave" DSP cores.

```
/* DSP Master is Core 0 */ define DSP_MASTER_CORE 0

/* Global PA instance */ Pa_Handle paInst;

/* Externally defined resource table */ extern Rm_Resource simpleResourceTable[];

Void main (Void) {

    Rm_Handle rmHandle;
    Qmss_StartCfg qmssStartCfg;
    Cppi_StartCfg cppliStartCfg;
    Pa_StartCfg paStartCfg;

    paSizeInfo_t paSize;
    paConfig_t paCfg;
    int sizes[pa_N_BUFS];
    int aligns[pa_N_BUFS];
    void* bases[pa_N_BUFS];

    if (DNUM == DSP_MASTER_CORE)
    {
        /* Master DSP Core */

        /* Initialize RM and provide the resource table */
        Rm_init(rmTestResourceTable);

        /* Get the Rm_Handle to register with LLDs */
        rmHandle = Rm_getHandle();

        /* Configure Qmss_InitCfg and Qmss_GlobalConfigParams values */

        /* Add the Rm_Handle to the Qmss_GlobalConfigParams structure */
        qmssGblCfgParams.qmRmHandle = rmHandle;

        /* Initialize QMSS and register RM */
        Qmss_init(&qmssInitConfig, &qmssGblCfgParams);

        /* Initialize CPPI */
        Cppi_init (&cppliGblCfgParams);
```

```

/* Register RM with CPPI */
cppiStartCfg.rmHandle = rmHandle;
Cppi_startCfg (&cppiStartCfg);
}
else
{
/* Slave DSP Core */

/* Wait for master core to complete RM initialization */
Rm_start();

/* Get the Rm_Handle to register with LLDs */
rmHandle = Rm_getHandle();

/* Start QMSS and register RM */
qmssStartCfg.rmHandle = rmHandle;
Qmss_startCfg (&qmssStartCfg);

/* Register RM with CPPI */
cppiStartCfg.rmHandle = rmHandle;
Cppi_startCfg (&cppiStartCfg);

}

/* Initialize PA, done from each core */

/* Get a PA buffer */
Pa_getBufferReq(&paSize, sizes, aligns);

/* Create a PA instance */
Pa_create (&paCfg, bases, &paInst);

/* Register RM with PA */
paStartCfg.rmHandle = rmHandle;
Pa_startCfg (paInst, &paStartCfg);

}

```

For a working example please see the `rm_testproject` under the `test/` directory of the RM LLD.

Chip Support Library (CSL)

The Chip Support Library constitutes a set of well-defined APIs that abstract low-level details of the underlying SoC device so that a user can configure, control (start/stop, etc.) and have read/write access to peripherals without having to worry about register bit-field details. The CSL services are implemented as distinct modules that correspond with the underlying SoC device modules themselves. By design, CSL APIs follow a consistent style, uniformly across Processor Instruction Set Architecture and are independent of the OS. This helps in improving portability of code written using the CSL.

CSL is realized as twin-layer – a basic register-layer and a more abstracted functional-layer. The lower register layer comprises of a very basic set of macros and type definitions. The upper functional layer comprises of “C” functions that provide an increased degree of abstraction, but intended to provide “directed” control of underlying hardware.

It is important to note that CSL does not manage data-movement over underlying h/w devices. Such functionality is considered a prerogative of a device-driver and serious effort is made to not blur the boundary between device-driver and CSL services in this regard.

CSL does not model the device state machine. However, should there exist a mandatory (hardware dictated) sequence (possibly atomically executed) of register reads/writes to setup the device in chosen “operating modes” as per the device data sheet, then CSL does indeed support services for such operations.

The CSL services are decomposed into modules, each following the twin-layer of abstraction described above. The APIs of each such module are completely orthogonal (one module’s API does not internally call API of another module) and do not allocate memory dynamically from within. This is key to keeping CSL scalable to fit the specific usage scenarios and ease the effort to ROM a CSL based application.

The source code of the CSL is located under `$(TI_PDK_INSTALL_DIR)\packages\ti\csl` directory.

Note: The CSL is build with LLD using same script, please refer the LLD build section for details

Chip Support Library Summary	
Component Type	Library
Install Package	PDK
Install Directory	pdk_tci6614_<version>\packages\ti\csl
Project Type	Eclipse RTSC ^[49]
Endian Support	Little & Big
Linker Path	\$(TI_PDK_INSTALL_DIR)\packages\ti\csl
Linker Sections	.vecs , .switch, .args, .cio
Section Preference	L2 Cache
Include Paths	\$(TI_PDK_INSTALL_DIR)\packages\ti\csl
Reference Guides	See docs under Install Directory
Support	Technical Support
Additional Resources	Chip support library ^[50]
Downloads	Product Updates
License	BSD ^[51]

Low Level Drivers

The Low Level Drivers (LLDs) provide interfaces to the various peripherals on your SoC Device.

The source code for the LLDs is located under `$(TI_PDK_INSTALL_DIR)\packages\ti\drv` directory.

To build all the example projects for the LLDs, run the batch file located under `$(TI_PDK_INSTALL_DIR)\packages\pdkProjectCreate.bat`.

The following table shows PDK LLD vs. SoC Availability

Driver	TCI6614
CSL	X
QMSS	X
PKTDMA (CPPI)	X
PA	X
SA	X
SRIO	X
PCIe	X
Hyperlink	X
EDMA3	X
FFTC	X
TCP3d	X
TCP3e	X
BCP	X
RM	X

Driver Library Summary	
Component Type	Library
Install Package	PDK
Install Directory	pdk_tci6614_<version>\packages\ti\drv
Project Type	Eclipse RTSC ^[49]
Endian Support	Little & Big
Linker Path	\$(TI_PDK_INSTALL_DIR)\packages\ti\drv
Linker Sections	N/A
Section Preference	N/A
Include Paths	\$(TI_PDK_INSTALL_DIR)\packages\ti\drv
Reference Guides	See docs under Install Directory
Support	Technical Support
Additional Resources	Chip support library ^[50]
Downloads	Product Updates
License	BSD ^[51]

Multicore Navigator

Multicore Navigator provides multicore-safe communication while reducing load on DSPs in order to improve overall system performance.

Packet DMA (CPPI)

The CPPI low level driver can be used to configure the CPPI block in CPDMA for the Packet Accelerator (PA). The LLD provides resource management for descriptors, receive/transmit channels and receive flows.

Additional documentation can be found in:

Document	Location
Hardware Peripheral Users Guide	User Guide ^[52]
LLD Users Guide	\$(TI_PDK_INSTALL_DIR)\packages\ti\drv\cppl\docs\ CPPI_QMSS_LLD_SDS.pdf
API Reference Manual	\$(TI_PDK_INSTALL_DIR)\packages\ti\drv\cppl\docs\cpplldDocs.chm
Release Notes	\$(TI_PDK_INSTALL_DIR)\docs\ReleaseNotes_CPPI_LLD.pdf

Queue Manager (QMSS)

The QMSS low level driver provides the interface to Queue Manager Subsystem hardware which is part of the Multicore Navigator functional unit for a keystone device. QMSS provides hardware assisted queue system and implements fundamental operations such as en-queue and de-queue, descriptor management, accumulator functionality and configuration of infrastructure DMA mode. The LLD provides APIs to get full entitlement of supported hardware functionality.

Additional documentation can be found in:

Document	Location
Hardware Peripheral Users Guide	User Guide ^[52]
LLD Users Guide	\$(TI_PDK_INSTALL_DIR)\packages\ti\drv\qmss\docs\ CPPI_QMSS_LLD_SDS.pdf
API Reference Manual	\$(TI_PDK_INSTALL_DIR)\packages\ti\drv\qmss\docs\qmsslldDocs.chm
Release Notes	\$(TI_PDK_INSTALL_DIR)\docs\ReleaseNotes_QMSS_LLD.pdf

Network Co-processor (NETCP)

NETCP provides hardware accelerator functionality for processing Ethernet packets.

Security Accelerator (SA)

The SA also known as cp_ace (Adaptive Cryptographic Engine) is designed to provide packet security for IPsec, SRTP and 3GPP industry standards. The SA LLD provides APIs to abstract configuration and control between application and the SA. Similar to the PA LLD, it does not provide a transport layer. The Multicore Navigator is used to exchange control packets between the application and the SA firmware.

Note: Due to export control restrictions the SA driver is a separate download from the rest of the SC-MCSDK. See download link in the *Related Software* section in *User Guide* above.

Additional documentation can be found in:

Document	Location
Hardware Peripheral Users Guide	User Guide ^[53]
LLD Users Guide	\$(TI_SA_LLD_<ver>_INSTALL_DIR)\sasetup\docs\UserGuide_SA_LLD.pdf
API Reference Manual	\$(TI_SA_LLD_<ver>_INSTALL_DIR)\sasetup\packages\ti\drv\sa\docs\doxygen\sa_llc_docs.chm
Release Notes	\$(TI_SA_LLD_<ver>_INSTALL_DIR)\sasetup\packages\ti\drv\sa\docs\ReleaseNotes_SA_LLD.pdf

Packet Accelerator (PA)

The PA LLD is used to configure the hardware PA and provides an abstraction layer between an application and the PA firmware. This does not include a transport layer. Commands and data are exchanged between the PA and an application via the Mutlicore Navigator.

Additional documentation can be found in:

Document	Location
Hardware Peripheral Users Guide	User Guide ^[54]
LLD Users Guide	\$(TI_PDK_INSTALL_DIR)\packages\ti\drv\pa\docs\pa_sds.pdf
API Reference Manual	\$(TI_PDK_INSTALL_DIR)\packages\ti\drv\pa\docs\paDocs.chm
Release Notes	\$(TI_PDK_INSTALL_DIR)\docs\ReleaseNotes_PA_LLD.pdf

I/O and Buses

Serial RapidIO (SRIO)

The SRIO Low Level Driver provides a well defined standard interface which allows application to send and receive messages via the SRIO peripheral.

Additional documentation can be found in:

Document	Location
Hardware Peripheral Users Guide	User Guide ^[55]
LLD Users Guide	\$(TI_PDK_INSTALL_DIR)\packages\ti\drv\srio\docs\SRIO_SDS.pdf
API Reference Manual	\$(TI_PDK_INSTALL_DIR)\packages\ti\drv\sa\docs\srioDocs.chm
Release Notes	\$(TI_PDK_INSTALL_DIR)\docs\ReleaseNotes_SRIODriver.pdf

Peripheral Component Interconnect Express(PCIe)

The PCIe module supports dual operation mode: End Point (EP or Type0) or Root Complex (RC or Type1). This driver focuses on EP mode but it also provides access to some basic RC configuration/functionality. The PCIe subsystem has two address spaces. The first (Address Space 0) is dedicated for local application registers, local configuration accesses and remote configuration accesses. The second (Address Space 1) is dedicated for data transfer. This PCIe driver focuses on the registers for Address Space 0.

Additional documentation can be found in:

Document	Location
Hardware Peripheral Users Guide	User Guide [56]
API Reference Manual	\$(TI_PDK_INSTALL_DIR)\packages\ti\drv\pcie\docs\pcieDocs.chm
Release Notes	\$(TI_PDK_INSTALL_DIR)\docs\ReleaseNotes_PCIE_LLD.pdf

Hyperlink

The Hyperlink peripheral provides a high-speed, low-latency, and low-power point-to-point link between two Keystone (SoC) devices. The peripheral is also known as vUSR and MCM. Some chip-specific definitions in CSL and documentation may have references to the old names. The LLD provides a well defined standard interface which allows application to configure this peripheral.

Additional documentation can be found in:

Document	Location
Hardware Peripheral Users Guide	User Guide [57]
API Reference Manual	\$(TI_PDK_INSTALL_DIR)\packages\ti\drv\hyplnk\docs\hyplnkDocs.chm
Release Notes	\$(TI_PDK_INSTALL_DIR)\docs\ReleaseNotes_HYPLNK_LLD.pdf

Co-processors

Bit-rate Coprocessor (BCP)

The BCP driver is divided into 2 layers: Low Level Driver APIs and High Level APIs. The Low Level Driver APIs provide BCP MMR access by exporting register read/write APIs and also provides some useful helper APIs in putting together BCP global and sub-module headers required by the hardware. The BCP Higher Layer provides APIs useful in submitting BCP requests and retrieving their results from the BCP engine.

Additional documentation can be found in:

Document	Location
Hardware Peripheral Users Guide	User Guide [58]
LLD Users Guide	\$(TI_PDK_INSTALL_DIR)\packages\ti\drv\bcp\docs\BCP_SDS.pdf
API Reference Manual	\$(TI_PDK_INSTALL_DIR)\packages\ti\drv\bcp\docs\bcpDocs.chm
Release Notes	\$(TI_PDK_INSTALL_DIR)\packages\ti\drv\bcp\docs\ReleaseNotes_BCPDriver.pdf

Turbo Coprocessor Decoder (TCP3d)

The TCP3 decoder driver provides a well defined standard interface which allows application to send code blocks for decoding and receive hard decision and status via EDMA3 transfers.

Additional documentation can be found in:

Document	Location
Hardware Peripheral Users Guide	User Guide ^[59]
LLD Users Guide	\$(TI_PDK_INSTALL_DIR)\packages\ti\drv\tcp3d\docs\TCP3D_DriverSDS.pdf
API Reference Manual	\$(TI_PDK_INSTALL_DIR)\packages\ti\drv\tcp3d\docs\TCP3D_DRV_APIIF.chm
Release Notes	\$(TI_PDK_INSTALL_DIR)\packages\ti\drv\tcp3d\docs\ReleaseNotes_TCP3DDriver.pdf

FFT Accelerator Coprocessor(FFTC)

The FFTC driver is divided into 2 layers: Low Level Driver APIs and High Level APIs. The Low Level Driver APIs provide FFTC MMR access by exporting register read/write APIs and also provides some useful helper APIs in putting together FFTC control header, DFT size list etc. required by the hardware. The FFTC Higher Layer provides APIs useful in submitting FFT requests and retrieving their results from the FFTC engine without having to know all the details of the Multicore Navigator.

Additional documentation can be found in:

Document	Location
Hardware Peripheral Users Guide	User Guide ^[60]
LLD Users Guide	\$(TI_PDK_INSTALL_DIR)\packages\ti\drv\fftc\docs\FFTC_SDS.pdf
API Reference Manual	\$(TI_PDK_INSTALL_DIR)\packages\ti\drv\fftc\docs\fftcDocs.chm
Release Notes	\$(TI_PDK_INSTALL_DIR)\packages\ti\drv\fftc\docs\ReleaseNotes_FFTCDriver.pdf

Resource Manager (RM)

The RM low level driver provides the Integrator a mechanism for assigning DSP initialization and usage permissions to various device resources. For more information on how to utilize the RM and which resources are covered by the RM please see the Resource Manager (RM) LLD section.

Additional documentation can be found in:

Document	Location
API Reference Manual	\$(TI_PDK_INSTALL_DIR)\packages\ti\drv\rm\docs\rmllldDocs.chm
Release Notes	\$(TI_PDK_INSTALL_DIR)\docs\ReleaseNotes_RM_LLD.pdf

Instrumentation

The instrumentation directory contains utilities that can be used for the purposes of DSP logging and debug.

Fault Management Library

The fault management library defines a standard interface for a DSP to inform the Linux kernel of a fault and provide crash dump information. On a fault, a DSP configured to use the fault management library can shut down all data transfer IO and then write crash dump information to a remoteproc region accessible to the Linux kernel in ELF Note section format. Also, during the fault, the dsp core under fault has an ability to send NMI pulse to other DSP cores to initiate crash on other remote DSP cores via NMI generation registers. Upon completion of the write the DSP will signal the ARM core it has crashed via the ARM IPC interrupt generation register.

1. release library (ti.instrumentation.fault_mgmt.ae66) - located under \instrumentation\fault_mgmt\lib, should be used normally for the best performance of the cycles since the code is compiled with the full optimization.



Note: Please note that sending NMI pulse to remote DSP cores to initiate the crash remotely can be optional. Please do not initiate a call to this function from the exception hook function, if the feature to crash remote DSP cores is not needed.

Fault Management Library Summary	
Component Type	Library
Install Package	PDK for TCI6614
Install Directory	pdk_tci6614_<version>\packages\ti\instrumentation\fault_mgmt
Project Type	CCS ^[35]
Endian Support	Little & Big
Library Name	Select for the TCI6614 EVM ti.instrumentation.fault_mgmt.ae66 (little) ti.instrumentation.fault_mgmt.ae66e (big)
Linker Path	\$(TI_PDK_INSTALL_DIR)\packages\ti\instrumentation\fault_mgmt\lib - for release version
Linker Sections	crash
Section Preference	none
Include Paths	\$(TI_PDK_INSTALL_DIR)\packages\ti\instrumentation\fault_mgmt fault_mgmt.h defines the interface
Reference Guides	See docs under Install Directory
Support	Technical Support
Additional Resources	Texas Instruments Embedded Processors Wiki ^[61]
Downloads	Product Updates
License	BSD ^[51]

Remoteproc Types

The remoteproc directory contains a set of Remoteproc types that are common with the Remoteproc types defined in the Linux kernel release. These types can be utilized by applications wishing to implement DSP to ARM logging and debug features via remoteproc. The instrumentation library example and test applications provide examples on how to utilize the remoteproc types to allow the Linux kernel to interpret information passed to it by the DSP via remoteproc.

Remoteproc Interface Summary	
Component Type	Interface
Install Package	PDK for TCI6614
Install Directory	pdk_tci6614_<version>\packages\ti\instrumentation\remoteproc
Project Type	CCS ^[35]
Endian Support	Little & Big
Include Paths	\$(TI_PDK_INSTALL_DIR)\packages\ti\instrumentation\remoteproc remoteproc.h defines the interface
Reference Guides	See docs under Install Directory
Support	Technical Support

Additional Resources	Texas Instruments Embedded Processors Wiki ^[61]
Downloads	Product Updates
License	BSD ^[51]

Trace Framework

Trace framework is an engine that relays information between producer and consumers. Below configurations are supported

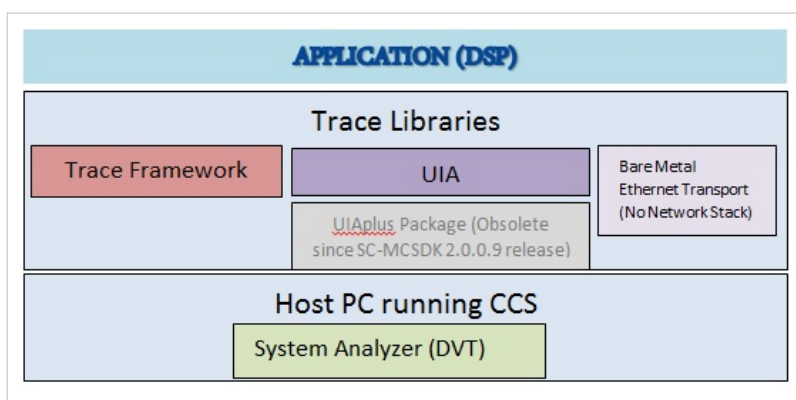
Producer	Consumers	Supported
DSP	ARM	Yes
DSP	DSP	Yes
ARM	ARM	Yes
ARM	DSP	No

- The information is conveyed using shared memory that is configured as contracts. Producers convey the information to consumers via Ring Buffers.
- Each Contract memory needs to be created before any producer or consumers are attached to it
- Trace Framework supports upto 64 contracts
- Every producer instance needs to be attached to an unique contract
- Each Contract can have maximum of 4 consumer instances at any given time.

The trace framework in PDK supports the ring producer and consumer libraries.

1. Please refer to `pdk_tci6614_1_00_00_##\packages\ti\instrumentation\traceframework\docs\html\index.html` for traceframework library API information
2. Please refer to `\pdk_tci6614_1_00_00_##\packages\ti\instrumentation\traceframework\docs\trace_frmwrk_ug.pdf` for the traceframework concept and example applications.

The system level representation for the trace framework library is as below.



Note:

1. System Analyzer's Tutorial 5B is now available at [System_Analyzer_Tutorial_5B](#) ^[62] link - it covers using LoggerStreamer and the PDK Trace Framework 'producer/consumer' model.
2. Trace Framework on DSP depends on UIA, UIA is a DSP component

that is included in the MCSDK installer. The documentation for the UIA packages are in the docs folder, that contains the system analyzer's user guide. UIA's API documentation can be located from CCS's help window (Help->Help Contents).

Trace Framework Library Summary	
Component Type	Library
Install Package	PDK for TCI6614
Install Directory	pdk_tci6614_<version>\packages\ti\instrumentation\traceframework
Project Type	CCS ^[35]
Endian Support	Little & Big
Library Name	Select for the TCI6614 EVM ti.instrumentation.traceframework.ae66 (little) ti.instrumentation.traceframework.ae66e (big)
Linker Path	\$(TI_PDK_INSTALL_DIR)\packages\ti\instrumentation\traceframework\lib - for release version
Linker Sections	.tf_genLogBuffersNull
Section Preference	none
Include Paths	\$(TI_PDK_INSTALL_DIR)\packages\ti\instrumentation\traceframework\traceframework.h
Reference Guides	See docs under Install Directory
Support	Technical Support
Additional Resources	Texas Instruments Embedded Processors Wiki ^[61]
Downloads	Product Updates
License	BSD ^[51]

Watchdog Timer Module

The Watchdog Timer Module provides APIs for using the local DSP timers as watchdog timers. Each DSP has a local timer, typically DSP n's local timer is system timer n, that can be configured and used as a 64-bit watchdog timer for that DSP. The Watchdog Timer Module provides APIs for an application to configure and "feed" a DSP's watchdog timer as an exception failsafe. If the DSP were to crash or become corrupted the watchdog timer will timeout. The Watchdog Timer Module provides options for if the watchdog times out. The timer can be configured to perform a hard/soft reset, generate an NMI, and generate an exception via the BIOS Exception module.

If the Watchdog Timer module is configured to generate an exception through the BIOS Exception module it can be plugged into the Fault Management library to generate a core dump on exception.

1. release library (ti.instrumentation.wdtimer.ae66) - located under \instrumentation\wdtimer\lib

Watchdog Timer Module Summary	
Component Type	Library
Install Package	PDK for TCI6614
Install Directory	pdk_tci6614_<version>\packages\ti\instrumentation\wdtimer
Project Type	CCS ^[35]
Endian Support	Little & Big
Library Name	Select for the TCI6614 EVM ti.instrumentation.wdtimer.ae66 (little) ti.instrumentation.wdtimer.ae66e (big)
Linker Path	\$(TI_PDK_INSTALL_DIR)\packages\ti\instrumentation\wdtimer\lib - for release version
Linker Sections	none

Section Preference	none
Include Paths	\$(TI_PDK_INSTALL_DIR)\packages\ti\instrumentation\wdtimer WatchdogTimer.h and WatchdogTimer.xdc define the interface
Reference Guides	See docs under Install Directory
Support	Technical Support
Additional Resources	Texas Instruments Embedded Processors Wiki ^[61]
Downloads	Product Updates
License	BSD ^[51]

Platform Library

The platform library defines a standard interface for platform utilities and functionality and provides sample implementations for the EVM platform. These include things such as reading and writing to EEPROM, FLASH, UART, etc. Platform library supports three libraries

1. debug library (ti.platform.evmtdci6614.ae66) - located under \platform_lib\lib\debug, needed only when a debug is needed on the platform library since the source is compiled with full source debugging.
2. release library (ti.platform.evmtdci6614.ae66) - located under \platform_lib\lib\release, should be used normally for the best performance of the cycles since the code is compiled with the full optimization.

Platform Library Summary	
Component Type	Library
Install Package	PDK for TCI6614
Install Directory	pkg_tci6614_<version>\packages\ti\platform\evmtci6614\platform_lib
Project Type	CCS ^[35]
Endian Support	Little
Library Name	Select for the TCI6614 EVM ti.platform.evmtdci6614.ae66 (little)
Linker Path	\$(TI_PDK_INSTALL_DIR)\packages\ti\platform\evmtci6614\platform_lib\lib\debug - for debug version \$(TI_PDK_INSTALL_DIR)\packages\ti\platform\evmtci6614\platform_lib\lib\release - for release version
Linker Sections	platform_lib
Section Preference	none
Include Paths	\$(TI_PDK_INSTALL_DIR)\packages\ti\platform platform.h defines the interface
Reference Guides	See docs under Install Directory
Support	Technical Support
Additional Resources	Texas Instruments Embedded Processors Wiki ^[61]
Downloads	Product Updates
License	BSD ^[51]

Note: For Alpha-0 and Alpha-1 release, only the simulator project is supported
\$(TI_PDK_INSTALL_DIR)\packages\ti\platform\simtdci6614\platform_lib)

Transport

The Transports are intermediate drivers that sit between either the NDK or IPC sub-systems and interface them to the appropriate EVM peripherals. The Transport supported by SC-MCSDK are:

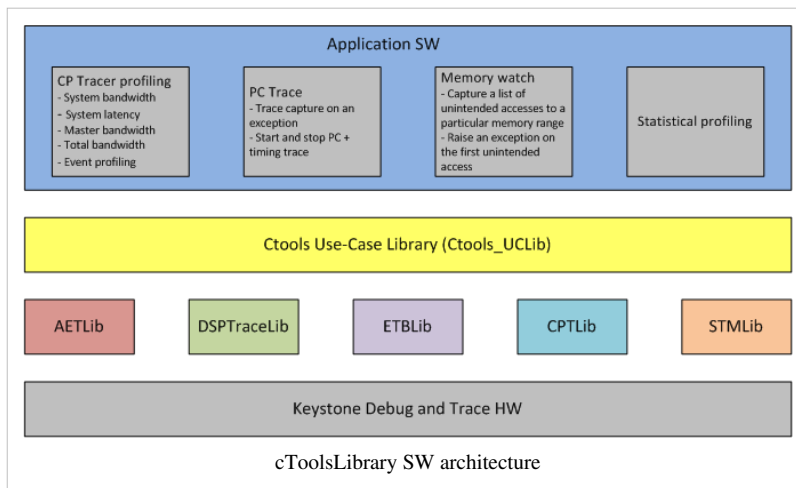
- NDK transport - Network Interface Management Unit (NIMU) Driver
- QMSS IPC transport - IPC MessageQ transport utilizing QMSS
- SRIO IPC transport - IPC MessageQ transport utilizing SRIO
- BMET - Bare Metal Ethernet Transport utilizing Netcp

More information on these can be found in the BMET or NDK or IPC sections of this guide.

cToolsLibrary

The cTools library provides APIs for using the individual instrumentation libraries for advanced users and also few use case based libraries that are built on top of the individual instrumentation libraries.

As shown in the figure below, the *ctoolslib_sdk* module provided in the package is intended to provide the glue between the use case based libraries, individual instrumentation libraries and application:



The *ctoolslib_sdk* is a collection of libraries for Advanced Event Triggering(AET), Common Platform Tracers(CPT), Ctools use case library(Ctools_UC), DSPTTrace, Embedded Trace Buffer(ETB) and System Trace Module(STM) - located under \aet\lib, \CPTLib\lib, \Ctools_UCLib\lib, \DSPTTraceLib\lib, \ETBLib\lib and \STMLib\lib

Ctools Library Package Summary	
Component Type	Library
Install Package	CtoolsLibrary for TCI6614
Install Directory	ctoolslib_<version>
Project Type	CCS ^[35]
Endian Support	Little & Big
Library Name	Select for the TCI6614 EVM Please see <i>GettingStarted.htm</i> for details
Linker Path	\$(TI_CTOOLSLIB_INSTALL_DIR)\packages\ti\LIBRARY_NAME\lib - for release/debug version, where LIBRARY_NAME = aet, CPTLib, Ctools_UCLib, DSPTTraceLib, ETBLib and STMLib.
Linker Sections	none
Section Preference	none

Include Paths	\$(TI_CTOOLSLIB_INSTALL_DIR)\packages\ti\LIBRARY_NAME\include*.h and ti\toolslib_sdk\evmtci6614\package.xdc define the interface for tci6614 use case library support
Reference Guides	See doc*html*\index.html file under respective libraries for details and CtoolsLib ^[63]
Support	Technical Support
Additional Resources	Texas Instruments Embedded Processors Wiki ^[61]
Downloads	Product Updates
License	BSD ^[51]

ctools Image processing Demo example

SC-MCSDK demonstrates Common Platform Tracer (CPT) capabilities using image processing demo as an application. Please refer to Image processing demo guide ^[64] for details on the demo and steps to be followed to run it on the TCI6614 EVM. The instrumentation examples are provided on the IPC version of the demo. Please refer to Run instructions and Expected output ^[65] for details on steps to be followed and expected output for use case based MCSDK instrumentation examples. The following are the supported use cases:

1. System Bandwidth Profile

1. Captures bandwidth of data paths from all system masters to the slave (or) slaves associated with one or more CP Tracers
2. Demo: For the Image demo, instrument the external memory (MSMC & DDR3) bandwidth used by all system masters

2. System Latency Profile

1. Captures Latency of data paths from all system masters to the slave (or) slaves associated with one or more CP Tracers
2. Demo: For the Image demo, instrument the external memory (MSMC & DDR3) Latency values from all system masters

3. The CP tracer messages (system trace) can be exported out for analysis in 3 ways:

1. System ETB drain using CPU (capture only 32KB (SYS ETB size in keystone devices) of system trace data)
2. System ETB drain using EDMA (ETB extension to capture more than 32KB (SYS ETB size in keystone devices))
3. External Emulator like XDS560 PRO or XDS560V2

4. Total Bandwidth Profile

1. The bandwidth of data paths from a group of masters to a slave is compared with the total bandwidth from all masters to the same slave.

1. The following statistics are captured:

1. Percentage of total slave activity utilized by a selected group of masters
2. Slave Bus Bandwidth (bytes per second) utilized by the selected group of masters
3. Average Access Size of slave transactions (for all system masters)
4. Bus Utilization (transactions per second) (for all system masters)
5. Bus Contention Percentage (for all system masters)
6. Minimum Average Latency (for all system masters)

2. Demo: For Image Demo, compare DDR3 accesses by Core0 (master core) with the DDR3 accesses by all other masters (which includes Core1, Core2...)

5. Master Bandwidth Profile

1. The bandwidth of data paths from two different group of masters to a slave is measured and compared.
 1. The following statistics are captured:
 1. Slave Bus Bandwidth (bytes per second) utilized by master group 0
 2. Slave Bus Bandwidth (bytes per second) utilized by master group 1
 3. Average Access Size of slave transactions (for both the master groups)
 4. Bus Utilization (transactions per second) (for both the master groups)
 5. Bus Contention Percentage (for both the master groups)
 6. Minimum Average Latency (for both the master groups)
 2. Demo: For Image Demo, compare DDR3 accesses by Core0 and Core1 with the DDR3 accesses by Core2 and Core3
6. Event Profile
 1. Capture new request transactions accepted by the slave. Filtering based on Master ID or address range is supported.
 1. The following statistics are captured for every new request event: Master ID which initiated this particular transaction
 1. Bus Transaction ID
 2. Read/Write Transaction
 3. 10 bits of the address (Address export mask selects, which particular 10 bits to export)
 2. Demo: For Image Demo, implement a Global SoC level watch point on a variable in DDR3 memory. Only Core0 is intended to read/write to this DDR3 variable. Unintended accesses by other masters (or Cores) should be captured and flagged.
7. Statistical profiling provides a light weight(based on the number of Trace samples needed to be captured) and coarse profiling of the entire application.
 1. PC trace is captured at periodic sampling intervals. By analyzing these PC trace samples, a histogram of all functions called in the application, with the following information:
 1. percent of total execution time
 2. number of times encountered
 2. Demo: In the Image Demo, Statistically profile the process_rgb() function, which processes a single slice of the Image.

Ctools PC trace example

There is a hardware support for logging the PC trace in TCI6614 in the ETB (Extended Trace Buffer), which is 4Kb in size. The Program Counter Trace along with Timing trace examples are located under `\sc_mcsdk_bios_2_00_00_11\examples\ctools\evmtci6614` folder. The examples include the following.

PC Trace where ETB is drained by CPU

This demonstrates the PC trace where ETB is drained by CPU. The PC Trace can be logged in either circular mode or stop mode to get 4Kb encoded PC trace information. Please follow below steps.

1. Power Cycle the EVM, make sure the EVM is brought up with Linux and you get to root prompt on the console.
2. Open CCS
3. Connect to DSP Core 0
4. Load the DSP ELF executable provided in the Example's Debug folder on DSP Core 0.
5. After the program is executed, the trace bin file gets generated in the same location which has the DSP ELF executable.
6. CCS provides a trace decoder tool, which consumes the .bin file and provides the PC trace decoded information. Please follow the steps outlined for using the Trace Decoder utility ^[66].

PC Trace where ETB is drained by EDMA

This demonstrates the PC trace where ETB is drained by EDMA. This mode provides the capability to extend the 4Kb ETB buffer size using EDMA. The PC Trace can be logged in either circular mode or stop mode to get more than 4Kb encoded PC trace information. Please follow below steps to execute the program.

1. Power Cycle the EVM, make sure the EVM is brought up with Linux and you get to root prompt on the console.
2. Open CCS
3. Connect to DSP Core 0
4. Load the DSP ELF executable provided in the Example's Debug folder on DSP Core 0.
5. After the program is executed, the trace bin file gets generated in the same location which has the DSP ELF executable.
6. CCS provides a trace decoder tool, which consumes the .bin file and provides the PC trace decoded information. Please follow the steps outlined for using the Trace Decoder utility ^[66].

PC Trace where ETB is drained by CPU targetted for Exception debug

This demonstrates the PC trace when the application is hitting an exception. The ETB is drained by CPU. The test example code demonstrates the APIs provided in the ctools Use case library. The PC Trace is always logged in circular mode. Please follow below steps to execute the program.

1. Power Cycle the EVM, make sure the EVM is brought up with Linux and you get to root prompt on the console.
2. Open CCS
3. Connect to DSP Core 0
4. Load the DSP ELF executable provided in the Example's Debug folder on DSP Core 0.
5. After the program is executed, the trace bin file gets generated in the same location which has the DSP ELF executable.
6. CCS provides a trace decoder tool, which consumes the .bin file and provides the PC trace decoded information. Please follow the steps outlined for using the Trace Decoder utility ^[66].

Using the Trace Decoder Utility

Trace Decoder utility is available from CCS. Please refer to TraceDecoder ^[67] page for details. The below sample usage should provide the PC trace dump to the console.

```
'<Dir Containing the Bin file>'<CCS Install
DIR>'<ccs_base\emulation\analysis\bin\td -procid 66x -bin
<binfilename>.bin -app <DSP ELF Executable Name>.out -rcvr
ETB
```

Other Ctools Library examples

Please refer to CCSv5 CtoolsLib Examples ^[68] for more information and to download other supported Ctools library examples. The downloaded Examples.zip should be extracted into [<CTOOLSLIB INSTALL>\packages\ti\] location. All the examples are CCSv5 compatible.

EDMA3 Low Level Driver

EDMA3 Low Level Driver is targeted to users (device drivers and applications) for submitting and synchronizing EDMA3-based DMA transfers.

EDMA3 is a peripheral that supports data transfers between two memory mapped devices. It supports EDMA as well as QDMA channels for data transfer. This peripheral IP is re-used in different SoCs with only a few configuration changes like number of DMA and QDMA channels supported, number of PARAM sets available, number of event queues and transfer controllers etc. The EDMA3 peripheral is used by other peripherals for their DMA needs thus

the EDMA3 Driver needs to cater to the requirements of device drivers of these peripherals as well as other application software that may need to use DMA services.

The EDMA3 LLD consists of an EDMA3 Driver and EDMA3 Resource Manager. The **EDMA3 Driver** provides functionality that allows device drivers and applications for submitting and synchronizing with EDMA3 based DMA transfers. In order to simplify the usage, this component internally uses the services of the **EDMA3 Resource Manager** and provides one consistent interface for applications or device drivers.

EDMA3 Driver Summary	
Component Type	Library
Install Package	EDMA3 Low level drivers
Install Directory	<root_install_dir>/edma3_lld_02_11_01_02
Project Type	N/A
Endian Support	Little and Big
Library Name	edma3_lld_drv.ae66 (little endian) and edma3_lld_drv.ae66e (big endian)
Linker Path	N/A
Linker Sections	N/A
Section Preference	N/A
Include Paths	N/A
Reference Guides	See docs under install directory
Support	Technical Support
Additional Resources	Programming the EDMA3 using the Low-Level Driver (LLD) ^[69]
Downloads	Product Updates
License	BSD ^[51]

SYS/BIOS RTOS

SYS/BIOS is a scalable real-time kernel. It is designed to be used by applications that require real-time scheduling and synchronization or realtime instrumentation. SYS/BIOS provides preemptive multi-threading, hardware abstraction, real-time analysis, and configuration tools. SYS/BIOS is designed to minimize memory and CPU requirements on the target.

SYS/BIOS Summary	
Component Type	Libraries
Install Package	SYS/BIOS
Install Directory	bios_6_<version>\
Project Type	Eclipse RTSC ^[49]
Endian Support	Little and Big
Library Name	The appropriate libraries are selected for your device and platform as set in the RTSC build properties for your project and based on the use module statements in your configuration.
Linker Path	The appropriate path is selected to the libraries for your device and platform as set in the RTSC build properties for your project.
Linker Sections	N/A

Section Preference	N/A
Include Paths	BIOS_CG_ROOT is set automatically by CCS based on the version of BIOS you have checked to build with. \${BIOS_CG_ROOT}\packages\ti\bios\include
Reference Guides	See docs under Install Directory
Support	Technical Support
Additional Resources	SYS/BIOS Online Training ^[4] SYS/BIOS 1.5-DAY Workshop ^[5] Eclipse RTSC Home ^[49]
Downloads	SYS/BIOS Downloads ^[70]
License	BSD ^[51]

Inter-Processor Communication (IPC)

Inter-Processor Communication (IPC) provides communication between processors in a multi-processor environment, communication to other threads on same processor, and communication to peripherals. It includes message passing, streams, and linked lists.

IPC can be used to communicate with the following:

- Other threads on the same processor
- Threads on other processors running SYS/BIOS
- Threads on GPP processors running SysLink (e.g., Linux)

IPC Summary	
Component Type	Libraries
Install Package	IPC
Install Directory	ipc_<version>\
Project Type	Eclipse RTSC ^[49]
Endian Support	Little and Big
Library Name	The appropriate libraries are selected for your device and platform as set in the RTSC build properties for your project and based on the use module statements in your configuration.
Linker Path	The appropriate path is selected to the libraries for your device and platform as set in the RTSC build properties for your project.
Linker Sections	N/A
Section Preference	N/A
Include Paths	N/A
Reference Guides	See docs under Install Directory
Support	Technical Support
Additional Resources	Eclipse RTSC Home ^[49]
Downloads	IPC Downloads ^[71]
License	BSD ^[51]

Bare Metal Ethernet Transport (BMET)

Bare Metal Ethernet Transport (BMET) enables the ethernet transport using the netcp. Please note that this does not provide any network stack, but simply enables to send the information over ethernet with UDP packets

BMET Summary	
Component Type	Libraries
Install Package	BMET
Install Directory	bmet_eth\
Project Type	Eclipse RTSC ^[49]
Endian Support	Little and Big
Library Name	The appropriate libraries are selected for your device and platform as set in the RTSC build properties for your project and based on the use module statements in your configuration.
Linker Path	The appropriate path is selected to the libraries for your device and platform as set in the RTSC build properties for your project.
Linker Sections	N/A
Section Preference	N/A
Include Paths	N/A
Reference Guides	See docs under Install Directory
Support	Technical Support
Additional Resources	Eclipse RTSC Home ^[49]
Downloads	SC-MCSDK
License	BSD ^[51]

Network Development Kit (NDK)

The NDK is a platform for development and demonstration of network enabled applications on DSP devices and includes demonstration software showcasing DSP capabilities across a range of network enabled applications. The NDK serves as a rapid prototype platform for the development of network and packet processing applications, or to add network connectivity to existing DSP applications for communications, configuration, and control. Using the components provided in the NDK, developers can quickly move from development concepts to working implementations attached to the network.

The NDK provides an IPv6 and IPv4 compliant TCP/IP stack working with the SYS/BIOS real-time operating system. Its primary focus is on providing the core Layer 3 and Layer 4 stack services along with additional higher-level network applications such as HTTP server and DHCP.

The NDK itself does not include any platform or device specific software. The NDK interfaces through well-defined interfaces to the PDK and platform software elements needed for operation.

The functional architecture for NDK is shown below.

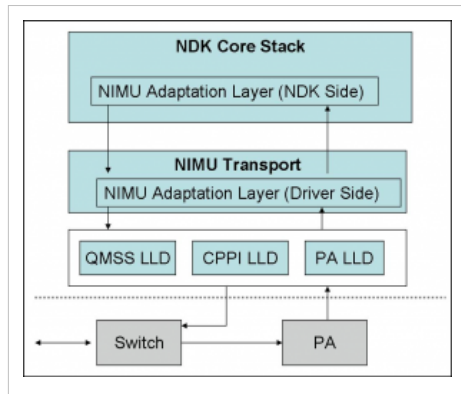
Extended Support	Eclipse RTSC Home ^[49] NDK User's Guide ^[72] NDK Programmer's Reference Guide ^[73] NDK Support Package Ethernet Driver Design Guide ^[74] NDK_FAQ ^[75] Rebuilding NDK Core ^[76]
Downloads	NDK Downloads ^[77]
License	BSD ^[51]

Network Interface Management Unit (NIMU) Driver

NIMU sits between NDK common software and the TCI6614 platform library and provides a common interface for NDK to communicate with. This package have NDK unit test examples for all supported platforms.

Note: This module is only intended to be used with NDK. As such, users should not tie up to its API directly.

The functional architecture for NIMU (taking TCI6614 platform as an example) is shown below.



NIMU Summary	
Component Type	Library
Install Package	PDK_INSTALL_DIR
Install Directory	sc_mcsdk_<version>\packages\ti\transport\ndk\nimu
Project Type	Eclipse RTSC ^[49]
Endian Support	Little
Library Name	ti.transport.ndk.nimu.ae66 (little)
Linker Path	\$(TI_PDK_INSTALL_DIR)\packages\ti\transport\ndk\nimu\lib\debug for debug version \$(TI_PDK_INSTALL_DIR)\packages\ti\transport\ndk\nimu\lib\release for release version
Linker Sections	nimu_eth_ll2
Section Preference	L2SRAM
Include Paths	\$(TI_PDK_INSTALL_DIR)\packages\ti\transport\ndk\nimu\include
Reference Guides	None
Support	Technical Support
Additional Resources	The NDK unit test examples are available in \$(TI_SCMCSDK_INSTALL_DIR)\examples\ndk\evm####
Downloads	[78]
License	BSD ^[51]

Syslib

This Syslib package includes the following four components:

- Resource manager library (Referred to as resmgr in this document)
- Packet library (Referred to as pktlib in this document)
- Message Communicator library (Referred to as msgcom in this document)
- Network Coprocessor Fast Path library (Referred to as netfp in this document)

For more details, please refer to SYSLIB User Guide and Release Notes.

Syslib Summary	
Component Type	Libraries
Install Package	SYSLIB
Install Directory	syslib_<version>\
Project Type	Eclipse RTSC ^[49]
Endian Support	Little and Big
Library Name	ti.runtime.msgcom.ae66 or ti.runtime.msgcom.ae66e and ti.runtime.msgcom-full.ae66 or ti.runtime.msgcom-full.ae66e and ti.runtime.netfp.ae66 or ti.runtime.netfp.ae66e and ti.runtime.pktlib.ae66 or ti.runtime.pktlib.ae66e and ti.runtime.resmgr.ae66 or ti.runtime.resmgr.ae66e
Linker Path	\$(SYSLIB_INSTALL_DIR)\packages\ti\runtime\msgcom\lib \$(SYSLIB_INSTALL_DIR)\packages\ti\runtime\netfp\lib \$(SYSLIB_INSTALL_DIR)\packages\ti\runtime\pktlib\lib \$(SYSLIB_INSTALL_DIR)\packages\ti\runtime\resmgr\lib
Linker Sections	
Section Preference	L2 Cache
Include Paths	\$(SYSLIB_INSTALL_DIR)\packages\ti\runtime\msgcom\include \$(SYSLIB_INSTALL_DIR)\packages\ti\runtime\netfp\include \$(SYSLIB_INSTALL_DIR)\packages\ti\runtime\pktlib\include \$(SYSLIB_INSTALL_DIR)\packages\ti\runtime\resmgr\include
Reference Guides	See docs under Install Directory
Support	Technical Support
Additional Resources	
Extended Support	Eclipse RTSC Home ^[49]
Downloads	SC-MCSDK
License	BSD ^[51]

Demonstration Software

The SC-MCSDK consist of demonstration software to illustrate device and software capabilities, benchmarks, and usage.

Note: The demo software is not supported in Alpha releases.

Tools

Multicore System Analyzer (MCSA)

Multicore System Analyzer (MCSA) is a suite of tools that provide real-time visibility into the performance and behavior of your code, and allow you to analyze information that is collected from software and hardware instrumentation in a number of different ways.

Advanced Tooling Features:

- Real-time event monitoring
- Multicore event correlation
- Correlation of software events, hardware events and CPU trace
- Real-time profiling and benchmarking
- Real-time debugging

Two key components of System Analyzer are:

- DVT: Various features of Data Analysis and Visualization Technology (DVT) provide the user interface for System Analyzer within Code Composer Studio (CCS)
- UIA: The Unified Instrumentation Architecture (UIA) target package defines APIs and transports that allow embedded software to log instrumentation data for use within CCS

MCSA Summary	
Component Type	Libraries
Install Package	UIA + DVT
Install Directory	ccsv5/uiia_<version>, ccsv5/eclipse, ccsv5/ccs_base_5.0.0.*/dvt\
Project Type	Eclipse RTSC ^[49]
Endian Support	Little
Library Name	The appropriate libraries are selected for your device and platform as set in the RTSC build properties for your project and based on the use module statements in your configuration.
Linker Path	The appropriate path is selected to the libraries for your device and platform as set in the RTSC build properties for your project.
Linker Sections	N/A
Section Preference	N/A
Include Paths	N/A
Reference Guides	See docs under Install Directory
Support	Technical Support
Additional Resources	Multicore System Analyzer ^[79]
Downloads	Installed as a part of SC-MCSDK installation
UIA License	BSD ^[51]

DVT License	TI Technology and Software Publicly Available (TSPA). See DVT Manifest in the install directory.
--------------------	--

Eclipse RTSC Tools (XDC)

RTSC is a C-based programming model for developing, delivering, and deploying Real-Time Software Components targeted for embedded platforms. The XDCtools product includes tooling and runtime elements for component-based programming using RTSC.

XDC Summary	
Component Type	Tools
Install Package	XDC
Install Directory	xdctools_<version>\
Project Type	Eclipse RTSC ^[49]
Endian Support	Little and Big
Library Name	The appropriate libraries are selected for your device and platform as set in the RTSC build properties for your project and based on the use module statements in your configuration.
Linker Path	The appropriate path is selected to the libraries for your device and platform as set in the RTSC build properties for your project.
Linker Sections	systemHeap
Section Preference	none
Include Paths	N/A
Reference Guides	See docs under Install Directory
Support	Technical Support
Additional Resources	Eclipse RTSC Home ^[49] Users Guide and Reference Manual ^[80]
Downloads	N/A
License	See XDC Manifest in the install directory

Build and Example Guide

The Build and Example Guide talks about setting up your build environment for SC-MCSDK, how to build the various components and then walks you through a set of example programs that are designed to teach you how to start writing programs using the software development kit.

Setting up the Build Environment

To set up the build environment you need to have:

- Installed Code Composer Studio.
- Installed the SC-MCSDK software.
- Have created a Target Configuration File to allow you to communicate with the EVM over JTAG.

The Getting Started Guide ^[81] talks about how to do this.

Once CCS and SC-MCSDK are installed, they provide you with both Debug and Release versions of the demos, examples and components. In addition, many of the components provide pre-built big endian versions as well. To re-build the demos and examples and components that do not provide pre-built big endian, see the section on

re-building for big endian in this Guide.

Creating a Target Configuration File

A Target Configuration File tells CCS how to connect to the EVM (they reference device specific files which have been supplied and contain information about the EVM, SoC and interface being used). Without one, you will not be able to load or debug applications over JTAG using CCS.

You create a Target Configuration File by selecting "New Target Configuration" in the Target Configurations Window of CCS. When creating a target configuration file you will also need to select the emulator interface being used: e.g. XDS 100 or BH 560. We generally use the emulator interface and EVM number in naming our target configurations. For example, if I am creating a target configuration for the TMD6614 I would create target configuration files with the names evmtci6614_xds100 or evmtci6614_bh560.

After you click Finish, the file will be displayed in the CCS main window for editing. It is here that you select the interface to use (e.g. XDS100, BH560, etc), whether a GEL file is being used and the device (e.g. TCI6614, etc).

XDS 100

You can find more detailed information about the XDS 100 interface here: <http://processors.wiki.ti.com/index.php/XDS100>

XDS 560

You can find more detailed information about the XDS 560 interface here:
http://processors.wiki.ti.com/index.php/Xds_560

Building the Software

Build in Place vs. Build in Workspace

The SC-MCSDK uses a "Build in Place" philosophy. This means you should not import the projects into the workspace. You can, but if you do, the projects may not re-build automatically and you may need to edit paths and other project settings to get them to build.

Note: It can be challenging to write a project that supports both build in place and build in workspace when the project is fairly rich and uses common source files (shared with other projects), etc.,.

Modifying a Library

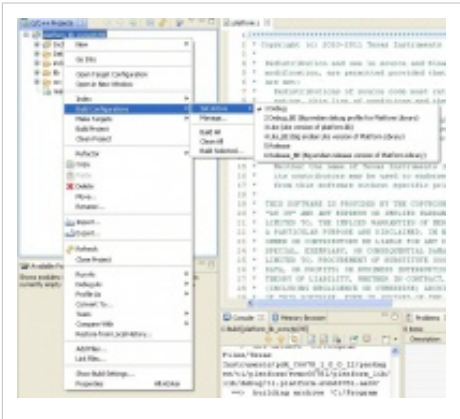
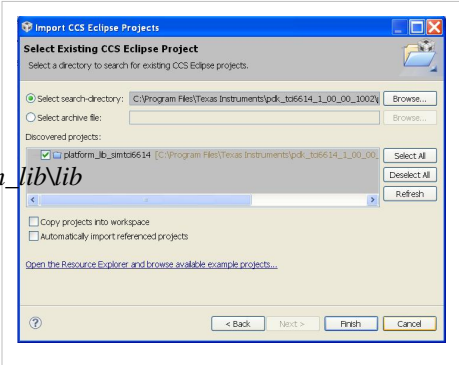
- If you want to modify and re-build a library its best not to copy it into your workspace. We suggest building it "in-place". When you build in-place you will not need to change build macros and so forth. You will also not have to edit the example projects as they will already have the correct paths to the library.
- If you want to experiment with a library routine, debug it or try some new functionality, add the file to your project and use it there. Once you are done with it, if its a change you need to add then you can re-build it in the library.
- You may want to make a backup copy of any library before you begin modifying it. This will allow you to get to the original more easily should you need to do so.

Platform Library

We will be building library in place which will allow other dependent application to pick up the library from usual place.

The following procedure assumes the SC-MCSDK is installed in *C:\ti*, the same directory where CCS is installed by default.

- Open CCS (preferably with a new workspace)
- Goto *Project->Import Existing CCS/CCE Eclipse project*
- In the *Select search-directory:* enter *C:\ti\pdk_tci6614_#_#_###packages\ti\platform\evmtci6614\platform_lib* and hit *Browse*. See Import Project Settings. This will import platform_lib_evmtci6614 into the workspace.
- Make sure the *Copy projects into workspace* is not checked. Then hit *Finish*.
- Now *Project->Clean* followed by *Project->Build All* should build the project and library is created in *C:\ti\pdk_tci6614_#_#_###packages\ti\platform\evmtci6614\platform_lib\lib* for a selected profile. Setting Profile for Project Settings. This will set the desired profile for platform_lib_evmtci6614 into the workspace.



Profile	Little endian Library name	Big Endian Library Name	Comment
Debug	/lib/debug/ti.platform.evmtci6614.ae66	/lib/debug/ti.platform.evmtci6614.ae66e	Full Symbol Debug Platform library
Release	/lib/release/ti.platform.evmtci6614.ae66	/lib/release/ti.platform.evmtci6614.ae66e	Optimized Full Platform library

See *platform_library_user_guide* located under *C:\Program Files\Texas Instruments\pdk_tci6614_#_#_###packages\ti\platform\docs\platform* for more information on platform APIs.

Building CSL and the Low Level Device Drivers

Please follow the instructions below to build CSL and LLDs.

- Open a command window inside of the \$(TI_PDK_INSTALL_DIR)\packages directory.
- Set the environment by running the batch file and follow the instructions as per the batch file output.

```
pdksetupenv.bat
```

- After configuring the environment successfully, the below message would appear.

```
... .. PDK BUILD ENVIRONMENT CONFIGURED
```

- To build the drivers run the below batch file.

```
.\ti\drv\pdkbuilder.bat
```

Building the Device Drivers Example Projects

The device drivers have example projects which can be verified after they are built with CCSV5. Please follow below steps to build the CCS projects for the example projects.

- Check Prerequisites

Please ensure that all dependent/pre-requisite packages are installed before proceeding with the examples and/or unit test.

- Configure CCS Environment

The CCS environment configuration step needs to be done only once for a workspace as these settings are saved in the workspace preferences. These settings only need to be modified if:

- New workspace is selected
- Newer version of the component is being used. In that case modify the paths of the upgraded component to the newer directory.

The procedure mentioned in this section is provided using <Managed Build Macro> option in CCS. Following are the steps:

- Create a macro file if not available from the PDK release. For the PDK release file:
<PDK_INSTALL_DIR>\packages\ti\drv\macros.ini can be used, where <PDK_INSTALL_DIR> refers to the location where PDK is installed.

Following environment would need to be available in the macros.ini file

```
PDK_INSTALL_PATH = <PDK_INSTALL_DIR>\packages
CSL_INSTALL_PATH = <PDK_INSTALL_DIR>\packages
CPPI_INSTALL_PATH = <PDK_INSTALL_DIR>\packages
QMSS_INSTALL_PATH = <PDK_INSTALL_DIR>\packages
PASS_INSTALL_PATH = <PDK_INSTALL_DIR>\packages
SA_INSTALL_PATH = <PDK_INSTALL_DIR>\packages
MAS_INSTALL_PATH = <PDK_INSTALL_DIR>\packages
SRIO_INSTALL_PATH = <PDK_INSTALL_DIR>\packages
```

- Import macros.ini located under \pdk_tci6614_1_0_0_XX\packages\ti\drv
 - This can be done as Click on CCS File menu option->Import->CCS->Managed Build Macros
 - Click on Next and Browse to open the macros.ini located in the above mentioned path
 - Click Finish
- Import the desired Example project and Build it under CCS to continue test.

Compiling Big Endian SC-MCSDK Demos and Examples

The pre-compiled platform libraries, NIMU drivers, and NDK examples provided in the package are Little Endian only. If Big Endian binaries are needed, they need to be rebuilt by changing the CCS build options. This section covers how to build and run the NDK Network Client example, NDK Network HelloWorld example, and HUA demo in big endian.

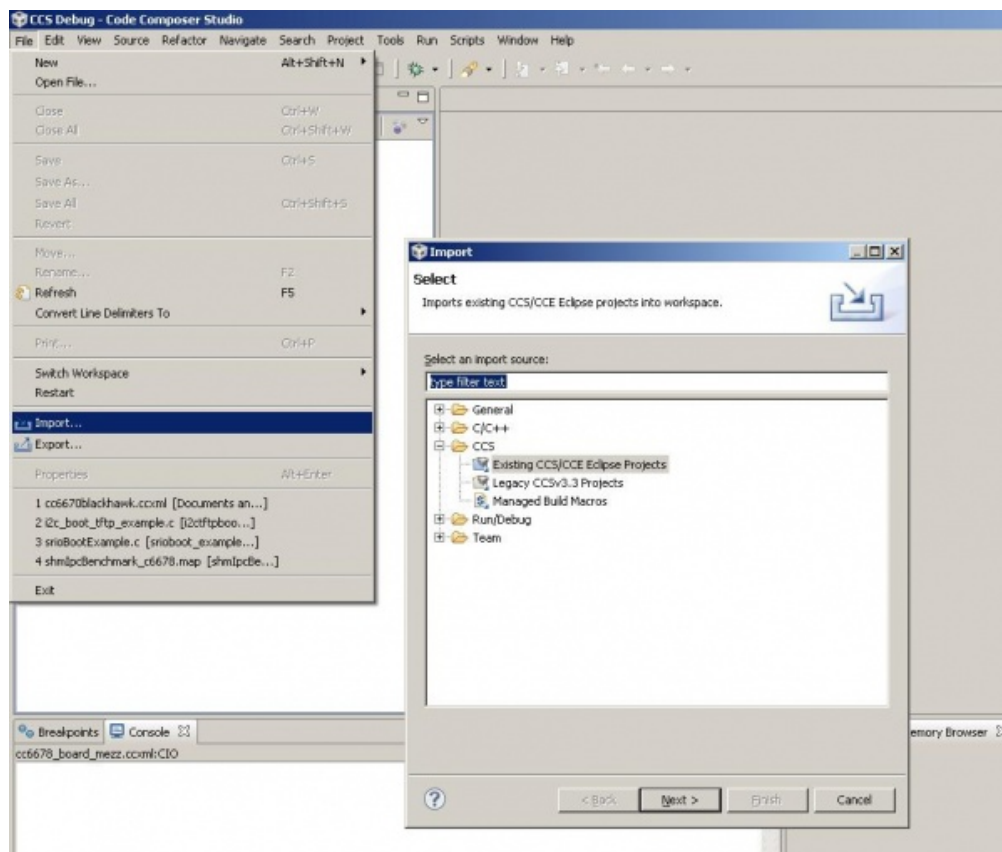
Note: The following images describing the steps to build the Big Endian libraries portray tci6614 projects.

Warning: Make sure to execute the EVM initialization GEL on the core the examples will be run on. The GEL's Global_Default_Setup function should be executed prior to loading and running any of the clients and examples. The GEL can be found under "CCSv5 installation path" \ccsv5\ccs_base_w.x.y.zzzzz\emulation\boards\evmtci6614\gel\evmtci6614.gel.

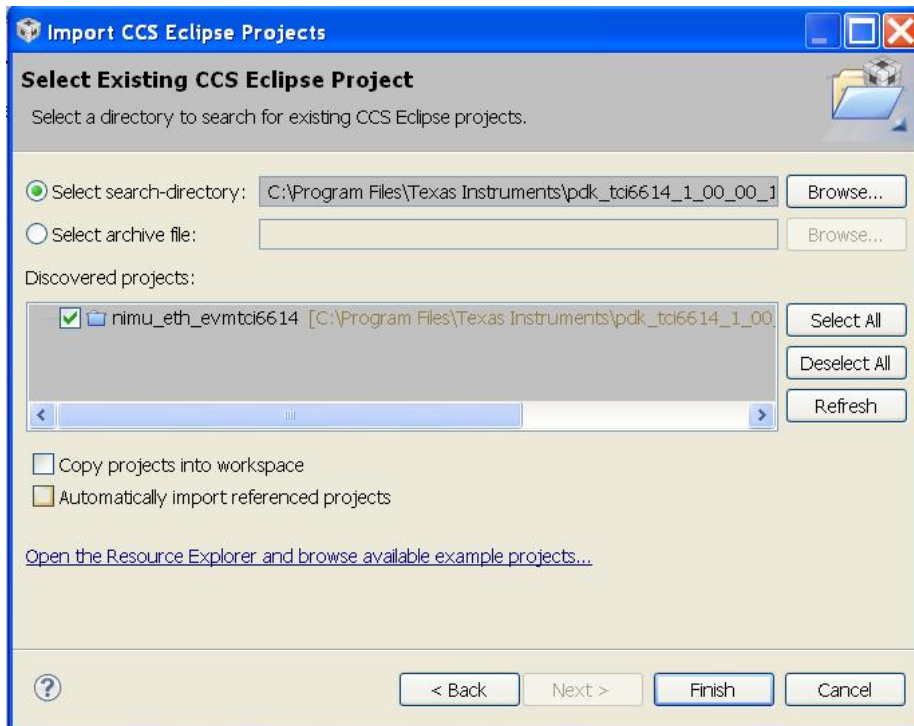
Recompile Big Endian NDK NIMU Driver

- The NIMU driver is required for all NDK examples. This must be recompiled in Big Endian prior to recompiling any example or demo in Big Endian.

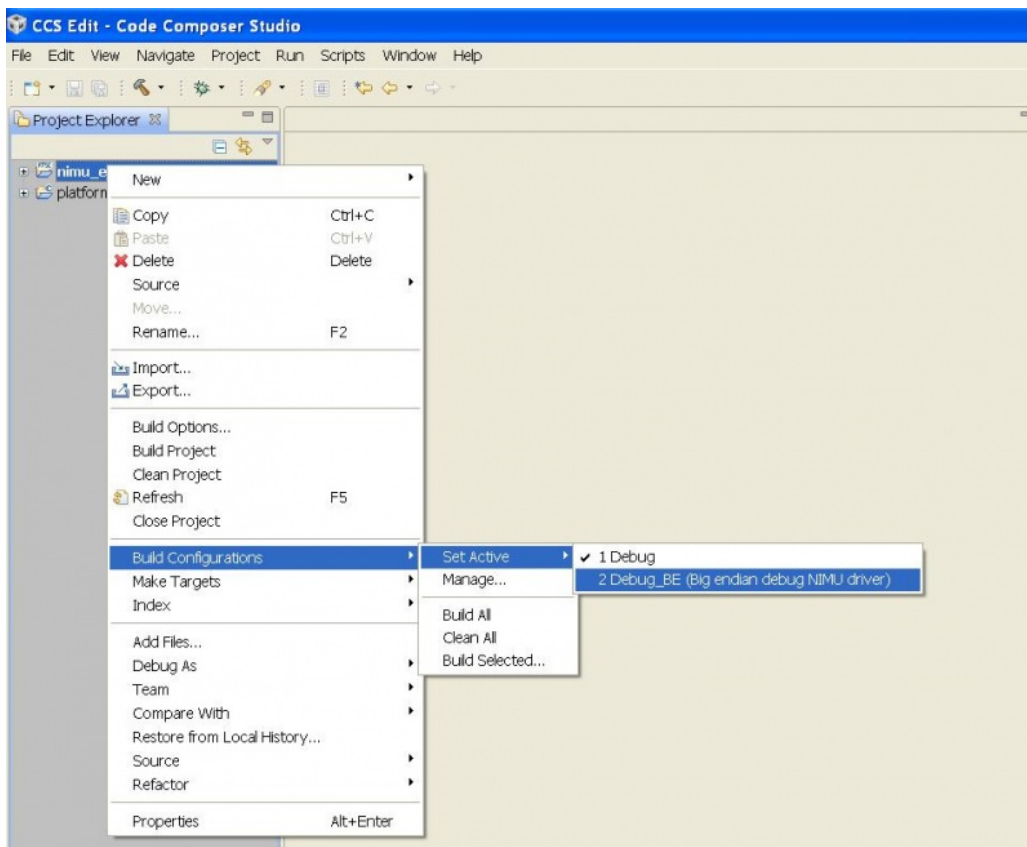
1. **Open the CCSv5 Project Import Wizard:** In CCSv5, click on File -> Import... to open the Project Import Wizard. Subsequently, select "Existing CCS/CCE Eclipse Projects" and click on the "Next" button as shown:



2. **Select and Import the NIMU Project:** Click the browse button to open a directory browser. Navigate to the PDK transport directory and select the NIMU transport project. Click "Finish" to import the nimu_eth_evmtci6614 project into CCS.



3. **Change the NIMU project active build configuration to Big Endian (Debug or Release):** In the C/C++ Projects window, right-click on the `nimu_eth_evmtci6614` RTSC project folder, click on Build Configurations -> Set Active -> Debug_BE (or Release_BE for release).



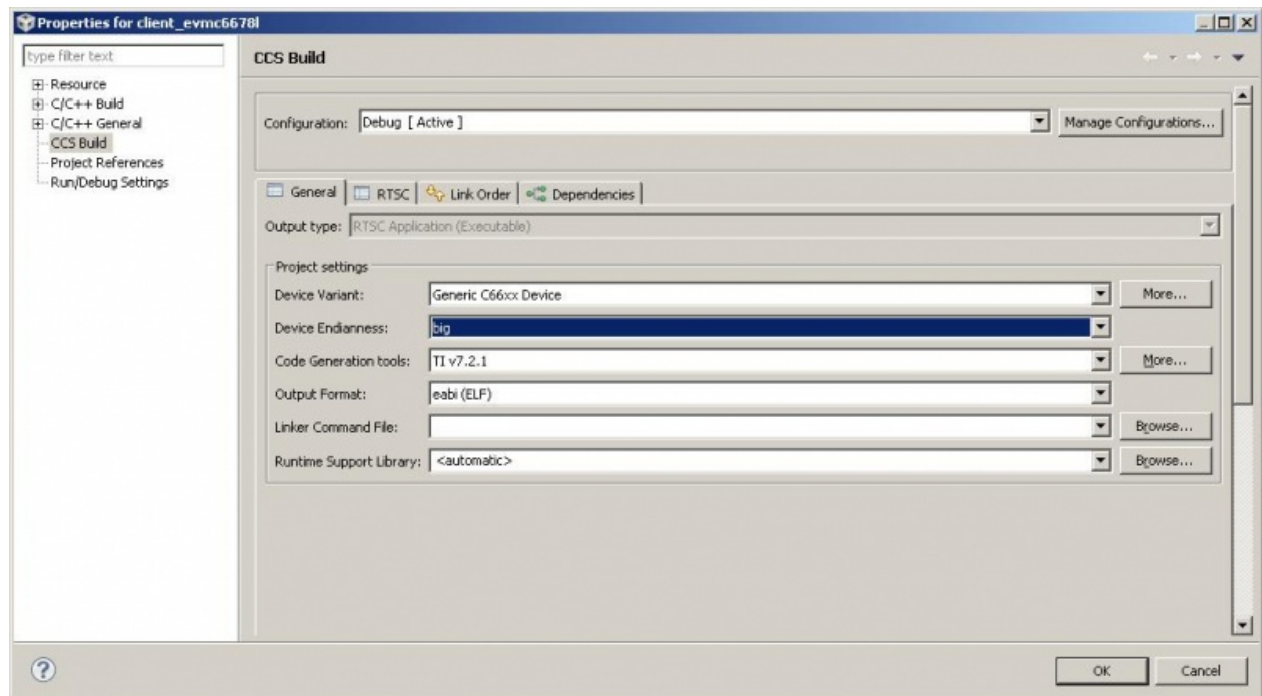
4. **Clean and Build the NIMU driver:** The NIMU driver will be rebuilt in Big Endian format and can now be linked by rebuilt Big Endian NDK examples.

Recompile Big Endian Platform Library

5. **Import the Platform library project:** Repeat steps 1. and 2. from above to import the platform_lib_evmtci6614 project. This project should be located within the PDK installation directory, under ti\platform\evmtci6614\platform_lib.
6. **Change the Platform project active build configuration to Big Endian (Debug or Release):** Repeat step 3. from above to set the big endian build configuration.
7. **Clean and Build the Platform library:** The Platform library will be rebuilt in Big Endian format and can now be linked by rebuilt Big Endian NDK examples and the HUA demo.

Recompile Big Endian NDK Client Example

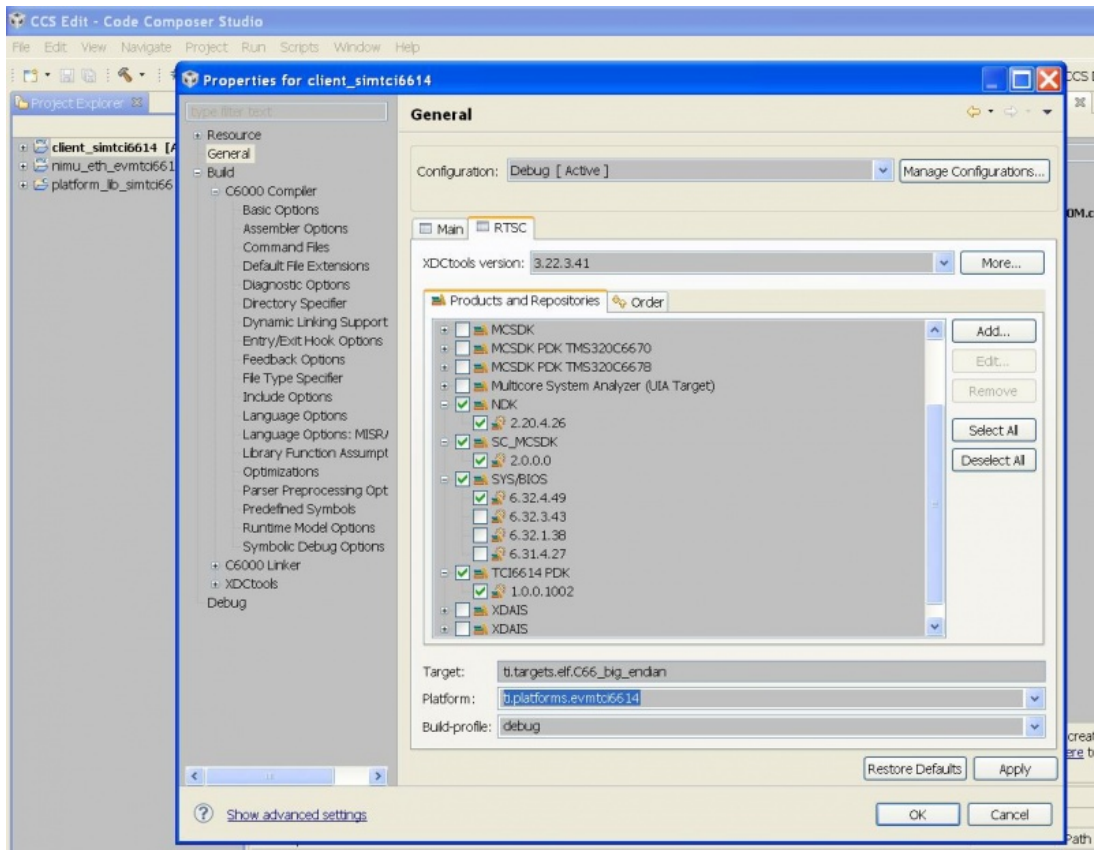
8. **Import the NDK Client example project:** Repeat steps 1. and 2. from above to import the client_evmtci6614 project. This project should be located within the SC-MCSDK installation directory, under examples\ndk\client\evmtci6614.
9. **Reconfigure the Client example for Big Endian:** With the client_evmtci6614 project selected, click on Project -> Properties and then select the "CCS Build" pane. In the "General" tab set "Device Endianness" to "big". Click "Apply".



In addition, click on the "RTSC" tab and configure the following and click "Apply" when finished:

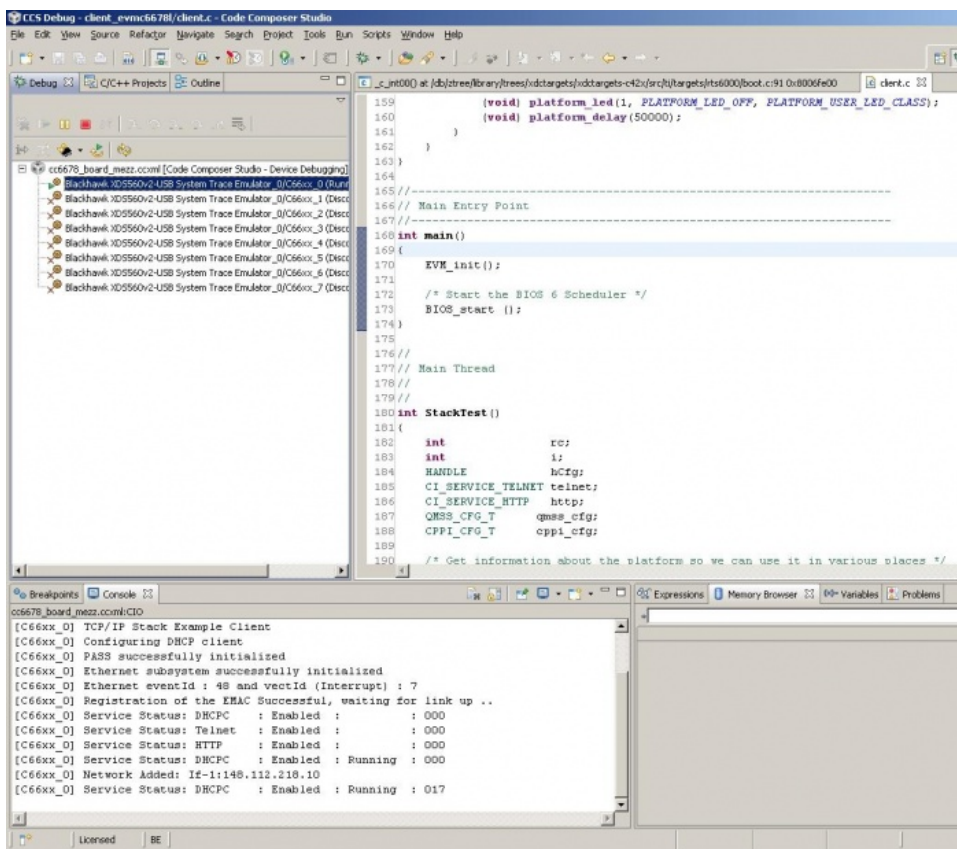
RTSC Target: ti.targets.elf.C66_big_endian

RTSC Platform: ti.platforms.evmtci6614



10. Clean and build the Client example: Clean and rebuild the Client example project from the project context menu.

Note: When the client example is executed the IP address negotiated with DHCP will be displayed backwards. As shown below the IP address reported is 148.112.218.10. The correct IP address is 10.218.112.148.



Recompile Big Endian NDK HelloWorld Example

11. Reconfigure NDK HelloWorld Example as Big Endian and rebuild: Follow step 8. through 10. to rebuild the NDK HelloWorld Example in Big Endian.

Building and running NDK client example with simulator

Setup RGMII/EMAC Adaptor in the CCS EMAC simulator

- Open the target Configuration file located under CCS simulation directory (simulation_csp_ny). For example, if CCSv5 is installed to its default directory, i.e., C:\Program Files\Texas Instruments\ccsv5, then the configuration file can be found at C:\Program Files\Texas Instruments\ccsv5\ccs_base\simulation_csp_aptn\bin\configurations with name tisim_tci6614_pv.cfg
- Pick a NIC on the PC running simulation that you'd like to use to run the example. This will be the interface using which the packets will be sent/received by the example.
- Under "EMAC_ADAPTOR" section look for USER_INPUTS sub-section, locate the following line of code,

```
INPUT2      ADAPTOR, OFF;
```

Modify the above line of code to:

```
INPUT2      ADAPTOR, ON;
```

This will turn on the EMAC adapter in simulator so as to send/receive packets.

- Under the same section, locate and modify the following line of code as follows:

```
INPUT4      NETWORK_ADAPTOR, Broadcom;
```

Modify the above line of code to include the name of the NIC card you are using, for example if the interface you are using for the test on your PC is a "Realtek" card, modify the above line to:

```
INPUT4      NETWORK_ADAPTOR, Realtek;
```

- If the following lines are uncommented, please comment them:

```
CONNECT11
System.C66XX_S.SHARED_SYSTEM.SWITCHSS.switchss_sgmiio_tx_data_gen_opin,

System.C66XX_S.SHARED_SYSTEM.SWITCHSS.switchss_sgmiil_rx_data_gen_ipin;
CONNECT12
System.C66XX_S.SHARED_SYSTEM.SWITCHSS.switchss_sgmiio_rx_data_gen_ipin,

System.C66XX_S.SHARED_SYSTEM.SWITCHSS.switchss_sgmiil_tx_data_gen_opin;
```

as follows:

```
//CONNECT11
System.C66XX_S.SHARED_SYSTEM.SWITCHSS.switchss_sgmiio_tx_data_gen_opin,

System.C66XX_S.SHARED_SYSTEM.SWITCHSS.switchss_sgmiil_rx_data_gen_ipin;
//CONNECT12
System.C66XX_S.SHARED_SYSTEM.SWITCHSS.switchss_sgmiio_rx_data_gen_ipin,

System.C66XX_S.SHARED_SYSTEM.SWITCHSS.switchss_sgmiil_tx_data_gen_opin;
```

This disables loopback at EMAC adapter level (PHY simulation) in the simulator.


```

C:\WINDOWS\system32\cmd.exe
C:\Program Files\Texas Instruments\ndk_2_20_04_26\packages\ti\ndk>dir/w
Volume in drive C has no label.
Volume Serial Number is 10F5-1C87

Directory of C:\Program Files\Texas Instruments\ndk_2_20_04_26\packages\ti\ndk

[.]                [..]                [build]             [config]            config.bld.default
[docs]             doxyfile             [inc]               [lib]               [package]
package.bld        package.xdc          package.xs          [productview]       Settings.h
Settings.xdc       [src]               [winapps]
                   7 File(s)          75,019 bytes
                   11 Dir(s)    63,680,442,368 bytes free

C:\Program Files\Texas Instruments\ndk_2_20_04_26\packages\ti\ndk>

```

- You will see a file called config.bld.default. You will need to edit this file.
- Make a *copy* of the file and call it config.bld.
- You will need to edit some settings in config.bld as discussed below. **Note:** These are the paths I am using. Yours may be different depending on where you installed CCS and/or SC-MCSDK.

Change the BIOS 6 path to where you have BIOS installed: var bios6path = "C:/Program Files/Texas Instruments/bios_6_32_01_38/packages";

Change the location for the Code Generation tools: var rootDir = "C:/Program\ Files/Texas\ Instruments/ccsv5/tools/compiler/c6000"

You can remove the ARM path if you are not building NDK for ARM or did not install ARM support. If you need ARM libraries built then make sure this has the right path: var rootDirArm = "C:/Program\ Files/Texas\ Instruments/ccsv4/tools/compiler/tms470"

Remove targets you do not need built. You should see our C66 targets. The others for ARM or C64 can be removed if you do not need to build for them. Build.targets = [

```

elfTargets.C66,
elfTargets.C66_big_endian,
];

```

Compile for Debug if you need debug by Changing the compiler options line C6xSuffix and adding a -g to it as below. var c6xSuffix = "-mi10 -mo -pdr -pden -pds=238 -pds=880 -pds1110 -g ";

- Save the file with your changes.
- Type xdc at the command line to build. Note that the xdc command must be run in the same directory as the config.bld.

```

C:\WINDOWS\system32\cmd.exe - xdc
C:\Program Files\Texas Instruments\ndk_2_20_04_26\packages\ti\ndk>dir/w
Volume in drive C has no label.
Volume Serial Number is 10F5-1C87

Directory of C:\Program Files\Texas Instruments\ndk_2_20_04_26\packages\ti\ndk

[.]                [..]                [build]             [config]            config.bld
[docs]             doxyfile             [inc]               [lib]               [package]
[package]          package.bld        package.xdc          [productview]
Settings.h         Settings.xdc       [src]               [winapps]
                   8 File(s)          77,278 bytes
                   11 Dir(s)    63,667,646,464 bytes free

C:\Program Files\Texas Instruments\ndk_2_20_04_26\packages\ti\ndk>xdc
making package.mak (because of package.bld) ...

```


Examples

The example programs are designed to take you from writing a simple "hello world" type program to progressively more complicated applications. At each step, various methodologies and ways of working with the SC-MCSDK are introduced. It is highly recommended that you do them.

Note: The following examples assume you installed SC-MCSDK in *C:\Program Files\Texas Instruments*. If you did not, then you will need to alter the paths used in this example to the location of where you installed it.

Note: The example programs make use of components contained in the PDK so you will need to specify the processor number and substitute it into the various paths and names as needed.

For example, a typical path might be:

```
"C:\Program Files\Texas Instruments\pdk_tci6614_1_0_0_xx\packages"
```

To specify that for the 6614 on the 1.0.0.1 release you would do:

```
"C:\Program Files\Texas Instruments\pdk_tci6614_1_0_0_01\packages"
```

Example 1 - Building and running a simple single core application

This is the first example program. Its purpose is to get you used to creating projects in CCS, building an executable and then running it on your EVM. The application executes out of shared memory on the EVM and does not use the external DDR.

Note: Please note that the simple platform library application code is assuming that everything is running from shared memory (MSMCRAM) - so no GEL file is needed. It is preferred to run the respective CCS GEL file for that platform before loading and running any application.

1. The first step is to create a project in CCS for this example. To do so follow the steps below.

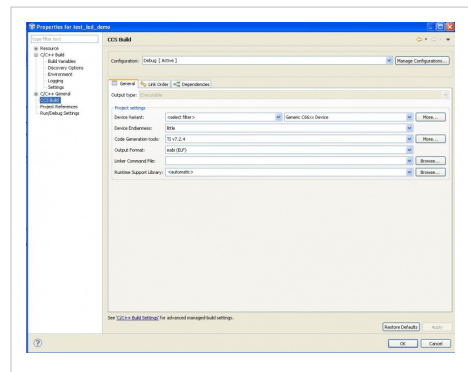
- Open CCS (preferably with a new workspace).
- Open *File->New->CCS Project* and in the project name field enter *led_play*", then hit Next.
- In the CCS project window, select *Project Type:* as *C6000* and hit *Next* and hit *Next* again to skip the next page for "Additional Project Settings.
- In the *New CCS Project*, select *Device Variant:* as *Generic C66xx Device* and hit *Next*. See Project Settings.
- In the *Project Templates* window select *Empty Project* and hit *Next*.
- It should open an empty project with name *led_play*.

2. Now that we have a project, we are going to create a source file that will use the SC-MCSDK Platform Library to a.) initialize our EVM at start-up, b.) write a simple string to the UART (console port) and c.) will blink the EVM LED's.

- Select *File->New->Source File*, enter *Source File* name as *led_play.c*, then hit *Finish*.
- It should open *led_play.c* empty file in the eclipse editor. Paste following source code in the editor

```
include <cerrno> include <stdio.h> include <stdlib.h> include <string.h>
include "ti\platform\platform.h" include "ti\platform\resource_mgr.h"

/* OSAL functions for Platform Library */ uint8_t *Osal_platformMalloc (uint32_t num_bytes, uint32_t alignment)
{
return malloc(num_bytes);
}
```



```
void Osal_platformFree (uint8_t *dataPtr, uint32_t num_bytes) {  
  
    /* Free up the memory */  
    if (dataPtr)  
    {  
        free(dataPtr);  
    }  
  
}  
  
void Osal_platformSpiCsEnter(void) {  
  
    /* Get the hardware semaphore.  
    *  
    * Acquire Multi core CPPI synchronization lock  
    */  
    while ((CSL_semAcquireDirect (PLATFORM_SPI_HW_SEM)) == 0);  
  
    return;  
  
}  
  
void Osal_platformSpiCsExit (void) {  
  
    /* Release the hardware semaphore  
    *  
    * Release multi-core lock.  
    */  
    CSL_semReleaseSemaphore (PLATFORM_SPI_HW_SEM);  
  
    return;  
  
}  
  
void main(void) {  
  
    platform_init_flags init_flags;  
    platform_init_config init_config;  
    platform_info p_info;  
    uint32_t led_no = 0;  
    char message[] = "\r\nHello World.....\r\n";  
    uint32_t length = strlen((char *)message);  
    uint32_t i;  
  
    /* Initialize platform with default values */  
    memset(&init_flags, 0x01, sizeof(platform_init_flags));  
    memset(&init_config, 0, sizeof(platform_init_config));  
    if (platform_init(&init_flags, &init_config) != Platform_EOK) {  
        return;  
    }  
  
    platform_uart_init();  
    platform_uart_set_baudrate(115200);  
  
    platform_get_info(&p_info);
```

```

/* Write to the UART */
for (i = 0; i < length; i++) {
    if (platform_uart_write(message[i]) != Platform_EOK) {
        return;
    }
}

/* Play forever */
while(1) {
    platform_led(led_no, PLATFORM_LED_ON, PLATFORM_USER_LED_CLASS);
    platform_delay(30000);
    platform_led(led_no, PLATFORM_LED_OFF, PLATFORM_USER_LED_CLASS);
    led_no = (++led_no) % p_info.led[PLATFORM_USER_LED_CLASS].count;
}
}

```

3. Our project now needs a linker command script. The linker command script defines the memory map for the platform (where internal, shared and external memory start, etc.) and where we want our code and data sections to be placed. We are going to put them in the shared memory region on the processor.

- Select *File->New->File from Template*, enter *File Name* as *led_play.cmd* and hit *Finish*.
- It would open *led_play.cmd* file in the editor, paste following linker command file in the editor

```
-c -heap 0x41000 -stack 0xa000
```

```
/* Memory Map */ MEMORY {
```

```

L1PSRAM (RWX) : org = 0x0E00000, len = 0x7FFF
L1DSRAM (RWX) : org = 0x0F00000, len = 0x7FFF
L2SRAM (RWX) : org = 0x0800000, len = 0x080000
MSMCSRAM (RWX) : org = 0xc000000, len = 0x200000
DDR3 (RWX) : org = 0x80000000, len = 0x10000000

```

```
}
```

```
SECTIONS {
```

```

.csl_vect > MSMCSRAM
.text > MSMCSRAM
GROUP (NEAR_DP)
{
    .neardata
    .rodata
    .bss
} load > MSMCSRAM
.stack > MSMCSRAM
.cinit > MSMCSRAM
.cio > MSMCSRAM
.const > MSMCSRAM
.data > MSMCSRAM
.switch > MSMCSRAM
.systemem > MSMCSRAM
.far > MSMCSRAM

```

```
.testMem > MSMCSRAM
.fardata > MSMCSRAM
platform_lib > MSMCSRAM

}
```


4. Were almost done. We have some code to execute and a memory map. Now we need to build the executable we will load and run. Before we build though, we will need to define a few include paths and specify the library for the Platform Library.

- Select *Project->Properties*, it should open Properties window for led_play project, select *C/C++ Build* from the left pane.
- Select *Settings* in the left pane after opening the *C/C++ Build* sub menu.
- In the *Tool Settings* tab, select *Include Options*, add following items in the *Add dir to #include search path...*

"C:\Program Files\Texas Instruments\pdk_tci6614_1_0_0_xx\packages"

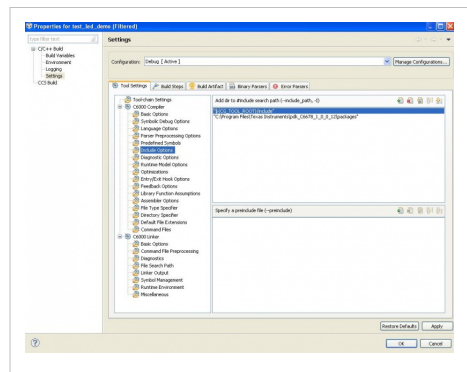
See Include Path

- Select *File Search Path* from *C6000 Linker* section. Add following items in *Include library...* section

ti.platform.tci6614.ae66  **Note:** Please note that the above library is the little endian debug version library of the platform library. This is needed for the application built for Little Endian. Please refer to the above table for including the appropriate library for the particular platform library application.

And add following items in *Add <dir> to library...* section

"C:\Program Files\Texas Instruments\pdk_tci6614_1_0_0_xx\packages\ti\platform\evmtci6614\platform_lib\lib\debug"

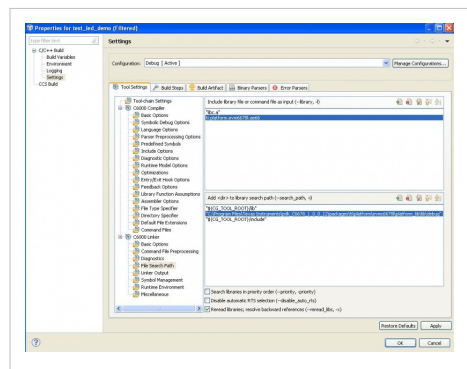


See Linker Input.

- Select *OK* to close the properties dialog box.
- Select *Project->Build Project* to build the project.

5. We should have an executable. Likely it was built as Debug since that is the default option to build unless it was changed. You can now follow the steps below to load and run your first example.

- Select *View->Target Configurations* to open target configuration tab in the left pane (this step assumes you have followed Getting Started Guide to create target configuration for your setup).
- Right click on the configurations file (#####.ccxml) and select *Launch Selected Configuration*.
- It should change the CCS prospective to *Debug* and load the configuration.
- After loading is complete select *Device* for core 0 (e.g. *C66XX_0*).
- Select *Target->Connect Target* to connect to the core.
- After core 0 is connected, select *Run->Load->Load Program*, then hit *Browse Project...*
- It should open *Select program to load* dialog, then select *led_play.out [...]* and hit *OK* and another *OK* to load the program to core 0.
- After loading completes, select *Target->Run* to run the application.
- The application should print *Hello World* if UART is connected to the board at 115200 baud rate and should flash LEDs.



Example 2 - Building and running your first tasking application using SC-MCSDK and BIOS

This example essentially re-does the first example and takes the LED code and puts it into a task. Note that while the steps may look similar there is a significant leap being made with BIOS and Eclipse RTSC being introduced.

1. The first step is to create an Eclipse RTSC project. To do that:

- Open CCS (preferably with a new workspace).
- Open *File->New->CCS Project* and in the project name field enter *led_play*, then hit *Next*.
- In the CCS project window, select *Project Type:* as *C6000* and hit *Next* and hit *Next* again to skip the next page for "Additional Project Settings."
- In the *New CCS Project*, select *Device Variant:* as *Generic C66xx Device* and hit *Next*.
- In the *Project Templates* screen, select an *Empty RTSC Project* and hit *Next*.
- In the *RTSC Configuration Settings* screen, check the *Repositories* (i.e. components) you want to use. All of them will be checked by default. Select only BIOS and the appropriate PDK for your EVM. Before you're done with this screen you need to select the *RTSC Platform* you are using. Select the *ti.platforms.evmtci6614* from the list box (note it will be empty, but just click on it and values will be filled in to select from).
- Hit *Finish*

2. Now we have an Eclipse RTSC project but nothing in it. Our next step is to create a *.cfg* file and the source file we want to use. The *.cfg* is essential to this project and serves many purposes: 1.) It replaces the *linker.cmd* file 2.) Allows you to include the various modules from BIOS and other Components you wish to use and 3.) allows you to configure default settings within them.

If you followed along in Example one you should know how to add files to a project. Add a C source file called *led_play.c*. Now we need to add the configuration file called *led_play.cfg* to the project. Do *File->New->RTSC Configuration File* and then name is *led_play.cfg*. You should now have both files as shown in the figure to the right called BIOS LED Example Project.

Note: Do not select a regular text file or a BIOS 5 configuration file when creating the *.cfg*.

3. Lets add the code we need to the *led_play.c* file:

1. `include <cerno>`
2. `include <stdio.h>`
3. `include <stdlib.h>`
4. `include <string.h>`
5. `include <ti/sysbios/BIOS.h>`
6. `include <ti/sysbios/hal/Hwi.h>`
7. `include <ti/bios/include/swi.h>`
8. `include "ti\platform\platform.h"`
9. `include "ti\platform\resource_mgr.h"`

```
/* OSAL functions for Platform Library */ uint8_t *Osai_platformMalloc (uint32_t num_bytes, uint32_t alignment)
{
```

```
    return malloc(num_bytes);
```

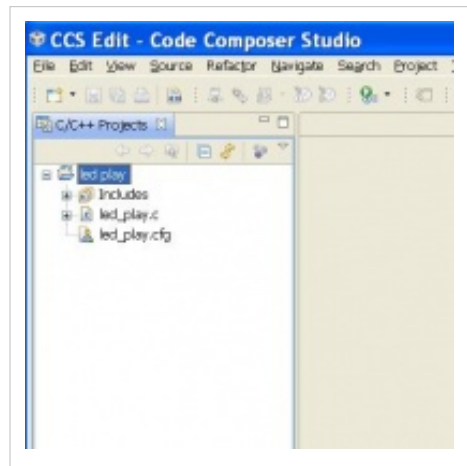
```
}
```

```
void Osai_platformFree (uint8_t *dataPtr, uint32_t num_bytes) {
```

```
    /* Free up the memory */
```

```
    if (dataPtr)
```

```
{
```



```

free(dataPtr);
}

}

void Osal_platformSpiCsEnter(void) {

/* Get the hardware semaphore.
*
* Acquire Multi core CPPI synchronization lock
*/
while ((CSL_semAcquireDirect (PLATFORM_SPI_HW_SEM)) == 0);

return;

}

void Osal_platformSpiCsExit (void) {

/* Release the hardware semaphore
*
* Release multi-core lock.
*/
CSL_semReleaseSemaphore (PLATFORM_SPI_HW_SEM);

return;

}

/*****

* main()
* Entry point for the application.
*****/

int main() {

/* Start the BIOS 6 Scheduler - it will kick off our main thread
ledPlayTask() */
platform_write("Start BIOS 6\n");

BIOS_start();

}

/*****

* EVM_init()
* Initializes the platform hardware. This routine is configured to
start in
* the evm.cfg configuration file. It is the first routine that BIOS
* calls and is executed before Main is called. If you are debugging
within
* CCS the default option in your target configuration file may be to
execute
* all code up until Main as the image loads. To debug this you should
disable

```

```

* that option.
*****/

void EVM_init() {

platform_init_flags sFlags;
platform_init_config sConfig;
int32_t pform_status;

/* Initialize the UART */
platform_uart_init();
platform_uart_set_baudrate(115200);
(void) platform_write_configure(PLATFORM_WRITE_ALL);

/*
 * You can choose what to initialize on the platform by setting the
 * following
 * flags. Things like the DDR, PLL, etc should have been set by the boot
 * loader.
 */
memset( (void *) &sFlags, 0, sizeof(platform_init_flags));
memset( (void *) &sConfig, 0, sizeof(platform_init_config));

sFlags.pll = 0; /* PLLs for clocking */
sFlags.ddd = 0; /* External memory */
sFlags.tcsl = 1; /* Time stamp counter */
sFlags.phy = 0; /* Ethernet */
sFlags.ecc = 0; /* Memory ECC */
sConfig.pllm = 0; /* Use libraries default clock divisor */

pform_status = platform_init(&sFlags, &sConfig);

/* If we initialized the platform okay */
if (pform_status != Platform_EOK) {
/* Initialization of the platform failed... die */
platform_write("Platform failed to initialize. Error code %d \n",
pform_status);
platform_write("We will die in an infinite loop... \n");
while (1) {
(void) platform_led(1, PLATFORM_LED_ON, PLATFORM_USER_LED_CLASS);
(void) platform_delay(50000);
(void) platform_led(1, PLATFORM_LED_OFF, PLATFORM_USER_LED_CLASS);
(void) platform_delay(50000);
}
}

return;

}

/*****

```

```

* ledPlayTask()
*
* This is the main task for the example. It will write send text
* messages to both the console and the UART using platform_write and
then
* twinkle the LEDs. This task is configured to start in led_play.cfg
* configuration file and it is called from BIOS.
*
***** /

```

```
int ledPlayTask (void) {
```

```
platform_info p_info;
```

```
uint32_t led_no = 0;
```

```
/* Get information about the platform */
```

```
platform_get_info(&p_info);
```

```
platform_write("Lets twinkle some LED's\n");
```

```
/* Play forever */
```

```
while(1) {
```

```
platform_led(led_no, PLATFORM_LED_ON, PLATFORM_USER_LED_CLASS);
```

```
platform_delay(30000);
```

```
platform_led(led_no, PLATFORM_LED_OFF, PLATFORM_USER_LED_CLASS);
```

```
led_no = (++led_no) % p_info.led[PLATFORM_USER_LED_CLASS].count;
```

```
}
```

```
}
```

4. Add the code to the cfg file *led_play.cfg* by opening it with a text editor. Note that if you double click it, it opens a tool you can use to edit the file but editing it via a text editor will be simpler.

```
/*
```

```
* led_play.cfg
```

```
*
```

```
* Memory Map and Program initialization for the BIOS
```

```
* LED example program.
```

```
*/
```

```
/* Include the various Modules we want to use */ var Memory = xdc.useModule('xdc.runtime.Memory'); var Startup
= xdc.useModule('xdc.runtime.Startup'); var BIOS = xdc.useModule('ti.sysbios.BIOS'); var Task =
xdc.useModule('ti.sysbios.knl.Task');
```

```
/* Configure the Modules */ BIOS.taskEnabled = true; /* Enable BIOS Task Scheduler */
```

```
/* Create our memory map - i.e. this is equivalent to linker.cmd */ Program.sectMap[".const"] = "MSMCSRAM";
Program.sectMap[".text"] = "MSMCSRAM"; Program.sectMap[".code"] = "MSMCSRAM";
Program.sectMap[".data"] = "MSMCSRAM"; Program.sectMap[".sysmem"] = "MSMCSRAM";
Program.sectMap[".platform_lib"] = "MSMCSRAM";
```

```
/* Lets register any hooks, tasks, etc that we want BIOS to handle */
```

```
/*
```


- Register an EVM Init handler with BIOS. This will initialize the hardware.
- BIOS calls before it starts.
- /

```
Startup.firstFxn.$add('&EVM_init');
```

```
/*
```

- Create the Main Thread Task for our application.
- /

```
var tskNdkMainThread = Task.create("&ledPlayTask"); tskNdkMainThread.stackSize = 0x2000;
tskNdkMainThread.priority = 0x5; tskNdkMainThread.instance.name = "ledPlayTask";
```


5. Now we need to configure a few project settings for the Platform Library (just like we did in the previous example).

- Select *Project->Properties*, it should open Properties window for led_play project, select *C/C++ Build* from the left pane.
- Select *Settings* in the left pane after opening the *C/C++ Build* sub menu.
- In the *Tool Settings* tab, select *Include Options*, add following items in the *Add dir to #include search path...*

```
"C:\Program Files\Texas Instruments\pdk_tci6614_1_0_0_xx\packages"
```

See Include Path

- Select *File Search Path* from *C6000 Linker* section. Add following items in *Include library...* section

ti.platform.evmtci6614.ae66  **Note:** Please note that the above library is the little endian debug version library of the platform library. This is needed for the application built for Little Endian. Please refer to the above table for including the appropriate library for the particular platform library application.

And add following items in *Add <dir> to library...* section

```
"C:\Program Files\Texas Instruments\pdk_tci6614_1_0_0_xx\packages\ti\platform\evmtci6614\platform_lib\lib\debug"
```

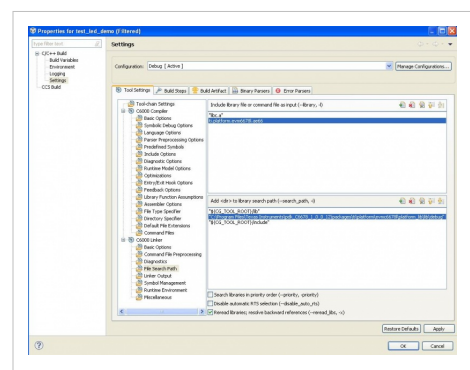
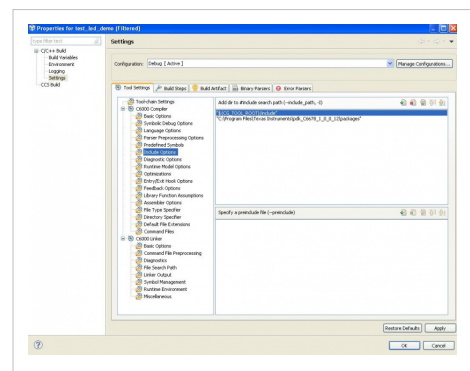
See Linker Input.

- Select *OK* to close the properties dialog box.
- Select *Project->Build Project* to build the project.

You maybe wondering why we do not need include/library paths or library names for BIOS? Any RTSC enabled component in the SC-MCSDK, provides its libraries and paths automatically during the build process. The appropriate libraries (big or little) and the paths are determined by the version of the component you selected in the CCS or RTSC Settings Screen. If you need to change any RTSC settings for an existing project, you can do so by highlighting the project name in CCS, then right clicking and selecting Properties and then selecting CCS from the menu.

6. Build the project.

7. Connect to your EVM with your Target Configuration file, then load and run the program!



Example 3 - Running from external memory (DDR)

Note: This topic is under construction - July 29, 2011

Example 4 - CTools PC Trace Use Case Demonstration Examples**DSP PC Trace during Exception**

TBD

DSP PC Trace, drain using CPU

TBD

DSP PC Trace, drain using EDMA

TBD

Using MAD to create a multi-core bootable Image

The Multicore Application Deployment (MAD) tool allows you to create a bootable image that can support multiple images and multiple cores. The premise behind MAD is to allow you to:

1. Deploy multiple applications on multiple cores.
2. Conserve memory by sharing common code.
3. Deploy an application dynamically on a core, if needed.

See the MAD Utils User Guide ^[82] for details.

An example of an SC-MCSDK application that uses MAD is the Image Processing Demo Guide ^[64].

Multi-core Programming Models

TBD.

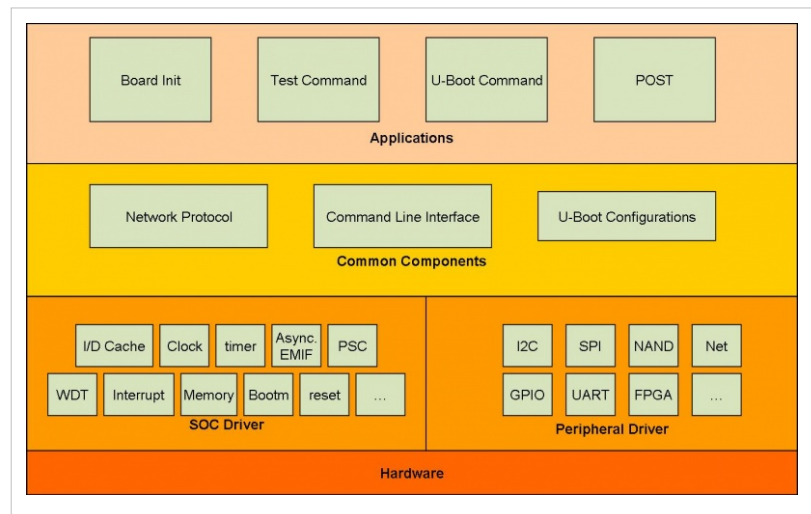
System**U-Boot Software Overview****Introduction**

- Appleton U-Boot is ported from Das U-Boot 2011.06 release.
- Das U-Boot wiki page: <http://www.denx.de/wiki/U-Boot>
- Free Software: full source code under GPL
- Hosted on git repository: <http://git.denx.de/cgi-bin/gitweb.cgi?p=u-boot.git;a=summary>
- Das U-Boot Documentation: <http://www.denx.de/wiki/U-Boot/Documentation>
- Das U-Boot Presentation: <http://www.denx.de/wiki/U-Bootdoc/Presentation>

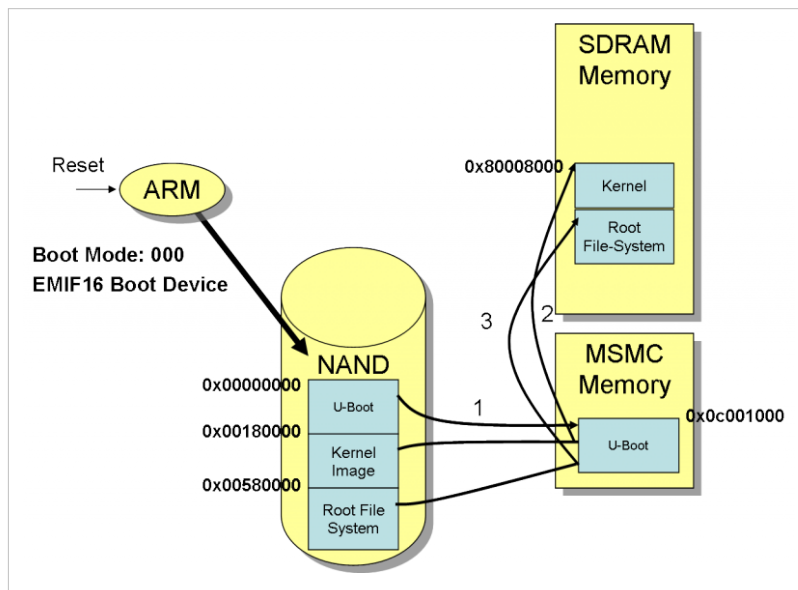
Architecture

U-Boot User Interface

- U-Boot uses a simple command line interface (CLI), over a serial console port.
- Configuration parameters and commands / command sequences (scripts) can be stored in "environment variables" which can be saved to NAND flash.
- Appleton U-Boot supports two different ways to load and boot an image.
- Ethernet: "tftp", "dhcp"
- NAND flash with JFFS2 filesystem: "nboot"



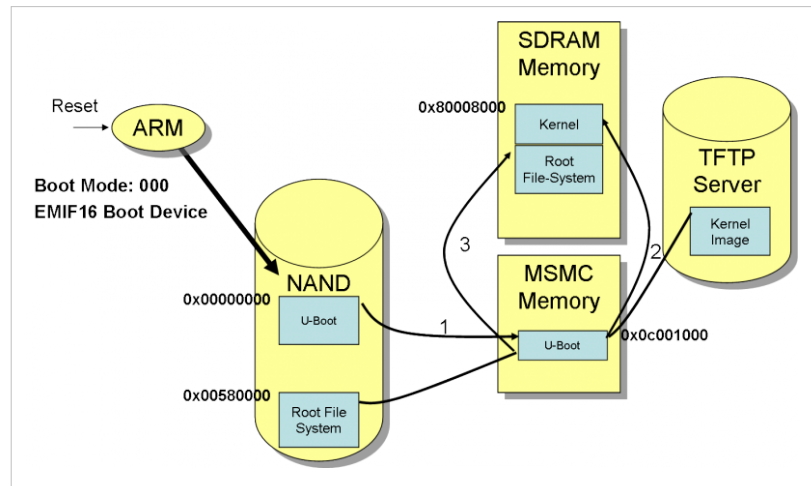
U-Boot Loading Kernel/FS from NAND



U-Boot Loading Kernel from TFTP Server

U-Boot Commands

- Memory Commands
 - mw
 - md
- Flash Memory Commands
 - nand
- Network Commands
 - Bootp
 - dhcp
 - ping
 - tftpboot
- Environment Variables Commands
 - printenv
 - saveenv
 - setenv
- Information Commands
 - coninfo
 - iminfo
 - imls
 - Help
 - version



Boot Kernel uImage

- Image:
 - Header + Payload
- Header:
 - Creation Timestamp
 - Data Load Address
 - Entry Point Address
 - Data CRC Checksum
 - Operating System
 - CPU architecture
 - Image Type
 - Compression Type
 - Image Name
- Actions:
 - test CPU architecture and OS
 - test checksum (optional)
 - if compressed, uncompress
 - copy to load address
 - prepare boot arguments

- start at entry point

Configuring U-Boot

- Configuration depends on the combination of board and CPU type; all such information is kept in a configuration file "include/configs/<board_name>.h". Example: For TCI6614 EVM, all configuration settings are in "include/configs/tci6614_evm.h".
- For all supported boards there are ready-to-use default configurations available; just type "make <board_name>_config". Example: For the TCI6614 EVM board type:
 - cd u-boot
 - make tci6614_evm_config
- For PG 1.1 devices, ethernet boot image size is limited to 256K. Current u-boot image cannot be used for Ethernet boot as the size is greater than this limit. To allow Ethernet boot of u-boot image, a smaller size image can be built using the configuration tci6614_evm_min_config. To build this image, following build commands can be used:-
 - cd u-boot
 - make tci6614_evm_min_config

Please note that this min version of u-boot cannot be used for Linux boot as there is no support for UBIFS and FDT. This can be used instead of loading and running u-boot from CCS very first time to flash the main u-boot on to the NAND flash.

Building U-Boot

- It is assumed that you have the GNU cross compiling tools available in your path and named with a prefix of "arm-linux-gnueabi-".
- U-Boot is intended to be simple to build. After installing the sources you must configure U-Boot for one specific board type. This is done by typing:
 - cd u-boot
 - make distclean;
 - make tci6614_evm_config or make tci6614_evm_min_config
 - make
- You should get some working U-Boot images ready for download to / installation on your system:
 - "u-boot.bin" is a raw binary image
 - "u-boot" is an image in ELF binary format
 - "u-boot.srec" is in Motorola S-Record format

U-Boot Utilities

Utility to format u-boot image for NAND boot

U-Boot image needs to be converted to the ROM boot loader compliant bootable image for NAND boot, there is a add_hdr_tlr utility provided in the Linux SC-MCSDK package (under host-tools/u-boot) to add a 2-word (8 bytes) header (load address and length) and a 2-word (8 bytes) trailer (both words 0) to the U-Boot image:

usage: ./add_hdr_tlr.out baseAddress infile outfile



Note: baseAddress is the load and entry address of the U-Boot image

E.g. ./add_hdr_tlr.out 0xc001000 u-boot.bin u-boot_hdr_tlr.bin

Utility to format u-boot image for Ethernet boot

U-boot image needs to be converted to the ROM boot loader compliant bootable image for Ethernet boot. Boot ROM execute the code from MSMC start address 0xc000000. This utility adds a branch instruction at the base address and fills rest of the area upto 0xc001000 with zeros. This is the image to be configured as bootfile in a DHCP server to download the image to the device for Ethernet boot. The utility is provided in the Linux SC-MCSDK package (host-tools/u-boot).

usage: ./uboot_eth infile outfile

E.g. ./uboot_eth u-boot.bin u-boot-eth.bin

U-Boot command to format the OOB

For TCI6614 EVM, U-Boot uses Linux standard ECC layout format in the OOB area for 4-bit HW ECC. For NAND device used on TCI6614 EVM, it uses the last 40 bytes of the 64-byte OOB area as the ECC bytes for a 2KB data page. RBL uses a different ECC layout format, it uses last 10 bytes of each 16 bytes in the OOB area as the ECC bytes. If user wants to burn the U-Boot image to the flash, it should re-format the ECC bytes in the oob area, below is the command to do the oob formatting:

Usage: oob fmt <start_addr in Hex> <size in Hex> - Re-format the OOB data from start offset byte addr with size bytes

If customers use a different NAND device with a different page size, they may need to modify the oob fmt command.

U-Boot debug symbol relocation

U-Boot supports code relocation on TCI6614 EVM, for debugging purpose, customers need to load/run symbol_reloc.gel (under host-tools/u-boot directory) to relocate the symbol.

For example, after you run u-boot.bin in CCS, once you enter the U-Boot command prompt, halt the CCS and run the symbol_reloc.gel to update the symbols that are relocated by the U-boot. Note that you may need to change the u-boot ELF image path and the relocated address based on your actual build path and the size of the u-boot image. You also need to set up the path mapping for the source and the binary image on your emulator target in CCS.

U-Boot does the code/symbol relocation in relocate_code() of arch/arm/cpu/armv7/start.S and jumps back to board_init_r() of arch/arm/lib/board.c after relocation. One way to figure out the relocated address in GEL_SymbolAddELFRel() is to set a breakpoint at 0xc00115c, and check the R1 register value, the relocated address = R1 - 0x1000.

Power On Self Test (POST)

The Power-On Self Test (POST) boot is designed to execute a series of platform/EVM factory tests on reset and indicate a PASS/FAIL condition using the LEDs and write test result to UART. A PASS result indicates that the EVM can be booted.

U-Boot by default will run POST to perform the following functional tests:

- External memory read/write test
- NAND read test
- NOR read test
- EEPROM read test
- UART write test
- Ethernet loopback test
- LED test

Additionally, POST provides the following useful information:

- FPGA version
- Board serial number
- EFUSE MAC ID
- Indication of whether SA is available on SOC
- PLL Reset Type status register



Note: POST is not intended to perform functional tests of the DSP.

In the NAND boot mode, after power on, the ARM starts execution with bootrom which transfers execution to the U-Boot program from NAND. The POST will then run through a sequence of platform tests. Upon power on, all the 4 FPGA debug LEDs will be on by default, then turn OFF if all the tests complete successfully. If any of the tests fails, the LED(s) will blink.

Below is the LED status table showing the test status/result:

Test Result	LED1	LED2	LED3	LED4
Test in progress	on	on	on	on
All tests passed	off	off	off	off
External memory test failed	blink	off	off	off
I2C EEPROM read failed	off	blink	off	off
EMIF16 NAND read failed	off	off	blink	off
SPI NOR read failed	off	off	off	blink
UART write failed	blink	blink	off	off
EMAC loopback failed	off	blink	blink	off
PLL initialization failed	off	off	blink	blink
NAND initialization failed	blink	blink	blink	off
NOR initialization failed	off	blink	blink	blink
Other failures	blink	blink	blink	blink

POST can be disabled by setting an environment variable under the U-Boot command prompt:

```
setenv no_post 1
saveenv
```

To enable the POST again, set the following environment variable under the U-Boot command prompt:

```
setenv no_post
saveenv
```

```
Tera Term - COM3 VT
File Edit Setup Control Window Help

TMDXEVMTCI6614 POST Version 01.00.00.00
SOC Information
FPGA Version: 0001
Board Serial Number: 0123456789
EFUSE MAC ID is: 00 18 31 7E 46 AB
SA is disabled on this board.
PLL Reset Type Status Register: 0x10000000
Additional Information:
(0x02350014): 0BEF0000
(0x02350624): 000211FF
(0x02350678): 00123400
(0x0235063C): 000801FF
(0x02350640): 000801FF
(0x02350644): 000900DB
(0x02350648): 000A40DB
(0x0235064C): 000B10DB
(0x02350650): 000C00DB
(0x02350654): 000C00DB
(0x02350658): 000C00DB
(0x0235065C): 000D1800
(0x02350660): 000E1800
(0x02350668): 000F1800
(0x02350670): 00101800
(0x02620008): 0C00C000
(0x0262000C): 0401017A
(0x02620010): 00000000
(0x02620014): 5D2A0021
(0x02620018): 0B96202F
(0x02620180): 0602F000

Power On Self Test
POST running in progress ...
POST I2C EEPROM read test started!
POST I2C EEPROM read test passed!
POST SPI NOR read test started!SF: Detected M25P32 with page size 64 KiB, total 4 MiB
POST SPI NOR read test passed!
POST EMIF16 NAND read test started!
POST EMIF16 NAND read test passed!
POST EMAC loopback test started!
POST EMAC loopback test passed!
POST external memory test started!ddr_memory_test PASSED!
POST external memory test passed!
POST done successfully!
POST result: PASS
```

Loading and booting DSP from ARM (*remoteproc*)

Introduction

The Linux remoteproc module can load DSP images provided in the filesystem and boot/run the DSP images from ARM using SYSFS interface. This section will provide detail procedures for creating a DSP image which can be loaded using remoteproc, known limitations and workarounds. This section also explains how to get DSP log messages from ARM debugfs.

Note: A sample unit test example application is provided in `sc_mcsdk_bios_2_##_##_##/examples/remoteproc` directory.

Creating a DSP image which can be loaded from ARM

The DSP images for each core needs to have an uncompressed section called `.resource_table`.

All the sections of the DSP images should fall in following memory segments

Memory resources reserved for DSP images

Resource Name	Start Address	Length
L2 Local	0x0080 0000	1M
L2 Global	0x[1-4]080 0000	1M
MSMC	0x0C00 0000	2M
DDR3	0xA000 0000	256M

The Linux image uses top half of DDR3 memory. The sections above are configured by default, and can be changed using device tree (*tc16614-evm.dts*) present in Linux kernel.

Creating a DSP image

The DSP image needs to have an uncompressed section with name **.resource_table**. The Linux kernel looks for this table in the DSP ELF image before loading it. The *.resource_table* can be used to provide SYS/BIOS information (log buffer, exception handler, etc.) to ARM/Linux kernel.

Following steps would create a *.resource_table* section in the DSP image with log buffer information

- In the Configuro Script File of the application add following commands to create a section

```
/* * The SysMin used here vs StdMin, as trace buffer address is required for *
Linux trace debug driver, plus provides better performance. */
Program.global.sysMinBufSize = 0x8000; var System =
xdc.useModule('xdc.runtime.System'); var SysMin =
xdc.useModule('xdc.runtime.SysMin'); System.SupportProxy = SysMin;
SysMin.bufSize = Program.global.sysMinBufSize;

var ti_sysbios_family_c64p_Exception = xdc.useModule("ti.sysbios.family.c64p.Exception");
ti_sysbios_family_c64p_Exception.enablePrint = true;

/* Load and use the Fault Management package */ var Fault_mgmt =
xdc.loadPackage('ti.instrumentation.fault_mgmt');

/* Load the Exception and register a exception hook */ var Exception =
xdc.useModule('ti.sysbios.family.c64p.Exception'); Exception.exceptionHook = '&myExceptionHook';
Exception.enablePrint = true;

/* BIOS Resource Table: */ Program.sectMap[".resource_table"] = new Program.SectionSpec();
Program.sectMap[".resource_table"].type = "NOINIT"; Program.sectMap[".resource_table"] = "L2SRAM"; Note:
The above section need not be in L2SRAM, it can be in any location.
```

- In a source/header file, create an resource array as follows

```
/* Remoteproc include file */
```

```
1. include <ti/instrumentation/remoteproc/remoteproc.h>
```

```
/* Fault Management Include File */
```

```
1. include <ti/instrumentation/fault_mgmt/fault_mgmt.h>
```

```
/* Add trace buffer information to the resource table */ extern char *
xdc_runtime_SysMin_Module_State_0_outbuf__A;
```

```
1. define TRACEBUFADDR (uint32_t)&xdc_runtime_SysMin_Module_State_0_outbuf__A
```

```
2. define TRACEBUFSIZE sysMinBufSize
```

```
1. pragma DATA_SECTION(resources, ".resource_table")
```

```
2. pragma DATA_ALIGN(resources, 4096)
```

```
struct resource resources[] = {
```

```
{TYPE_TRACE, 0, TRACEBUFADDR,0,0,0, TRACEBUFSIZE, 0,
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,"trace:sysm3"},
{TYPE_COREDUMP_NOTE, 0, (uint32_t) &fault_mgmt_data[0],0,0,0,
FAULT_MGMT_DATA_SIZE, 0,
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,"exception_trace_dump"},
```

```
};
```

```
Void myExceptionHook(Void) {
```

```
uint32_t i, my_core_id, efr_val;
/* Copy register status into fault management data region for Host */
Fault_Mgmt_getLastRegStatus();
```

```
my_core_id = CSL_chipReadDNUM();
```

```
efr_val = CSL_chipReadEFR();
if (efr_val & 0x80000000) {
```

/* Exception is triggered to be assumed by other core for NMI exceptions * so, don't need to notify other cores since the parent core where the * exception has triggered by an event would notify them, thus eliminating * the recursive exceptions */

/* Notify Host of crash and return */

```
Fault_Mgmt_notify();
```

```
return;
```

```
}
```

```
for (i = 0; i < FAULT_MGMT_NOTIFY_NUM_CORES; i++) {
  /* Skip the notification if it is same core */
  if (i == my_core_id)
```

```
continue;
```

```
/* Notify remote DSP cores for exception - WARNING: This will
generate NMI
```

```
pulse to the remote DSP cores */
Fault_Mgmt_notify_remote_core(i);
}
```

```
/* Notify Host of crash */
Fault_Mgmt_notify();
```

```
}
```

```
void main(void) {
```

```
/* The output of System_printf will be available
from Linux trace debug driver */
System_printf("Main started on core %d\n", DNUM + 1);
```

```
..... }
```

Note:

- The CCS project by default turns on ELF RLE compression, this should be turned off. The project should be linked with **--ram_model** option.
- The remoteproc ELF loader loads those segments to DSP memory, whose *PT_LOAD* field is set. In order to skip loading of a particular section, set the type to *NOLOAD* in the command/cfg file.

```
/*      Section      not      to      be      loaded      by      remoteproc      loader      */
Program.sectMap[".noload_section"].type = "NOLOAD";
```

Naming convention for DSP images and location in filesystem

The DSP firmware images reside in the file system under the **/lib/firmware** directory.

DSP core image naming convention:

- DSP image name for Core 0: dsp-core0.out
- DSP image name for Core 1: dsp-core1.out
- DSP image name for Core 2: dsp-core2.out
- DSP image name for Core 3: dsp-core3.out

SYSFS command to load and run DSP image

Once the kernel is up and running, in order to load and boot a DSP firmware image, you will need to write to a state variable for each dsp-core which is represented as a sysfs entry in the file system.

There are sysfs entries for the 4 cores in the following directory in the filesystem

- Sysfs entry for core 0: */sys/class/remoteproc/dsp-core0*
- Sysfs entry for core 1: */sys/class/remoteproc/dsp-core1*
- Sysfs entry for core 2: */sys/class/remoteproc/dsp-core2*
- Sysfs entry for core 3: */sys/class/remoteproc/dsp-core3*

Use the following commands to load and run each core

- To load and run DSP core 0: **echo "running" > /sys/class/remoteproc/dsp-core0/state**
- To load and run DSP core 1: **echo "running" > /sys/class/remoteproc/dsp-core1/state**
- To load and run DSP core 2: **echo "running" > /sys/class/remoteproc/dsp-core2/state**
- To load and run DSP core 3: **echo "running" > /sys/class/remoteproc/dsp-core3/state**

Use the following commands to stop each core and place it in reset

- To load and run DSP core 0: **echo "offline" > /sys/class/remoteproc/dsp-core0/state**
- To load and run DSP core 1: **echo "offline" > /sys/class/remoteproc/dsp-core1/state**
- To load and run DSP core 2: **echo "offline" > /sys/class/remoteproc/dsp-core2/state**
- To load and run DSP core 3: **echo "offline" > /sys/class/remoteproc/dsp-core3/state**

Remoteproc Device Tree Configuration

The user can configure the DDR2 start addresses, memory sections, DSP image names from *<Linux Kernel Source>/arch/arm/boot/dts/tci6614-evm.dts* device tree file.

DSP log message trace from Linux

The DSP log messages can be read from following locations

- DSP log entry for core 0: */debug/remoteproc/dsp-core0/trace0*
- DSP log entry for core 1: */debug/remoteproc/dsp-core1/trace0*
- DSP log entry for core 2: */debug/remoteproc/dsp-core2/trace0*
- DSP log entry for core 3: */debug/remoteproc/dsp-core3/trace0*

```
root@tci6614-evm:~# cat /debug/remoteproc/dsp-core0/trace0 Main started on core 1 .... root@tci6614-evm:~#
```

Note: The log buffer is a simple circular buffer intended for logging boot messages. This buffer will reset to the beginning on overflow. If the user desires a post boot snapshot at a prescribed interval after boot, the user should save a snapshot of these buffers as a part of the boot process.

DSP runtime snapshot and capturing of extra segments/registers

The user can configure remoteproc to capture extra segments in the coredump. The segments configured, will appear as extra segments in the coredump image. The extra segments can be global peripheral registers user want to capture for debugging purposes. But, care should be taken not to set the registers which changes system behavior on read (like queue pop registers).

Note that, the coredump can be captured while DSP is not crashed. So in a running system user can take snapshot of DSP memory and load it to CCS to analyze for values in program variables/structures.

The command configures remoteproc/coredump to capture extra segments in coredump image is as follows

```
echo    "<extra      segment      start      address>      <extra      segment      length>"    >
/debug/remoteproc/dsp-core<core-id>/coredump
```

For example: Following command configures remoteproc/coredump to capture *Ethernet switch subsystem configuration* in TCI6614

```
root@tci6614-evm:~# echo "0x02090000 0x30000" > /debug/remoteproc/dsp-core0/coredump
```

```
root@tci6614-evm:~# cp /debug/remoteproc/dsp-core0/coredump ./coredump
```

Please see *Generating_DSP_coredump_in_ARM* and subsequent sections in the user guide to further process the coredump image.

Another example is, the NOLOAD sections does not appear in program segment table of the image. If there are any such segments, user can run following command in Linux PC to generate the script which can be used to capture the extra sections.

```
readelf --sections dsp-core0.out | grep .no_load_section | awk '{print "echo \"0x"$4" 0x"$6"\" >
/debug/remoteproc/dsp-core0/coredump"}' >> coredump_no_load_segment.sh
```

Note that, if you capture coredump when DSP is not crashed, the coredump will not have information on DSP local registers like PC, Ax, Bx, etc. as the DSP is running.

Loading and running setup (preload) binary before loading ELF image

There can be situations, where user need to run a firmware image before loading actual application. For example, in a secure application running on a secure device, there may be a requirement to preload a monitor firmware which sets up necessary vector table. In order to preload a firmware image before loading the actual application, the location and start address of the firmware image need to be added to the device tree in the *rproc* structure:

```
preload-firmware = "preload-dsp.bin"; /* Preload image name (it should be present in /lib/firmware directory) */
preload-firmware-addr = <0x0c0f8000>; /* Preload image address */
```

The remoteproc command to run with pre-loading is given below

```
echo "runningwithpreload" > /sys/class/remoteproc/dsp-core0/state
```

The remoteproc will load and run the binary image provided in *preload-firmware* parameter in device tree at the address *preload-firmware-addr*. Then it will load and run the DSP application image pointed by device tree. The *runningwithpreload* is not a state, it moves the state machine to *running* state but it provides an indication to remoteproc to pre-load and run the binary image before loading and running the application image.

Following are the properties expected from *preload-firmware* image:

- The image needs to be a flat binary.
- The boot address and load address of the image need to be same, which is *preload-firmware-addr*
- The image should poll *boot magic address* to load actual application image

DSP Fault Management (FM)

This section describes the Fault Management APIs and provides information on how instrument a DSP image for exception, capture the DSP core dump, and analyze the fault using the core dump in CCS.

FM APIs

A DSP application can create and install an exception hook configured with the following features using the APIs provided by FM:

- Retrieve DSP register status for a core dump
- Halt system IO, such as DMAs
- Inform remote DSPs an exception has occurred
- Inform Host a DSP exception has occurred

Retrieving Last Register Status

An exception hook function that intends to create a core dump must call the following API:

```
void Fault_Mgmt_getLastRegStatus(void);
```

The API stores relevant DSP core register values into the memory region mapped to the *fault_mgmt_data* data array for the core dump that is provided to the Host.

Halting System IO

When a DSP exception is detected, invoking the FM instrumented exception hook function, system IO that offload data transfers can be stopped using the following API:

```
Fault_Mgmt_haltIoProcessing(Fm_GlobalConfigParams *fmGblCfgParams, Fm_HaltCfg *haltCfg)
```

The API provides the ability to disable all or subsets of the following system IO:

- AIF2 PE (TX) and PD (RX) channels
- EDMA3 DMA, QDMA, and INT channels

- CPDAM tx/rx channels
- SGMII switch

The exception hook can invoke this function in order to stop DSP enabled IO from continuing data transfers that may wipe out data important to deciphering the source of the DSP exception. However, care must be taken to not disable resources in use by the Host. If this occurs the Host (ARM) may experience unstable behavior preventing it from properly receiving and handling the DSP core dump.

Global Configuration Parameters

The `Fm_GlobalConfigParams` structure informs the IO halt function of the system specifics for the IO it will be halting. The user should not create their own version of this structure. Instead, the version of the structure provided within `pdk_tci6614_#_##_##_##\packages\instrumentation\fault_mgmt\device\tci6614src\fm_device.c` should be used. This structure has been statically created based on the TCI6614 platform IO parameters.

IO Halt Configuration Parameters

The `Fm_HaltCfg` structure is used to define which system IO will be disabled upon invocation of the IO halt API. A user can specify that an entire IO be disabled or a subset of an IO be disabled. The `Fm_HaltCfg` structure has the following parameters:

- `int32_t haltAif` - If set to a non-zero value, will halt all AIF2 PE and PD channels except those specified within the `Fm_ExcludedResource`'s list. All AIF2 PE and PD channels will be disabled if the `Fm_ExcludedResource`'s list is NULL.
- `int32_t haltSGMII` - If set to a non-zero value, will halt the SGMII ethernet switch. **Note:** This will prevent ARM Linux ethernet from working properly.
- `int32_t haltEdma3` - If set to a non-zero value, will halt all EDMA3 DMA, QDMA, and INT channels except those specified within the `Fm_ExcludedResource`'s list. All EDMA3 DMA, QDMA, and INT channels will be disabled if the `Fm_ExcludedResource`'s list is NULL. **Note:** Some EDMA3 channels are used by ARM Linux to access NAND in UBIFS. UBIFS will not work correctly if these channels are halted.
- `int32_t haltCpdma` - If set to a non-zero value, will halt all CPDMA channels except those specified within the `Fm_ExcludedResource`'s list. All CPDMA channels will be disabled if the `Fm_ExcludedResource`'s list is NULL. **Note:** Some CPDMA channels are used by ARM Linux which will exhibit unknown behavior if the relevant CPDMA channels are halted.
- `Fm_ExcludedResource *excludedResources;` - List of specific system IO values that should not be disabled.

A customized version of the `Fm_ExcludedResource`'s list can be created. However, a version has been created and supplied in `pdk_tci6614_#_##_##_##_##\packages\instrumentation\fault_mgmt\device\tci6614src\fm_device.c` which already accounts for all resources used the ARM Linux kernel delivered with the latest SC-MCSDK 2.x release.

Note: Following the IO halt configuration defined in the FM test source code will disable all IO except that used by ARM Linux UBIFS and NFS.

Informing Remote DSP Core's of Exception

Remote DSP cores can be informed of a DSP exception and told to halt functioning using the following API:

```
void Fault_Mgmt_notify_remote_core(uint32_t core_id);
```

The DSP local to the exception will interrupt remote cores via their NMI causing them to enter their own exception handling routine. This allows all DSPs to be brought down when a single DSP exception occurs. In multi-core application this may help preserve information relevant to deciphering the root cause of the original DSP exception.

Informing Host of DSP Exception

A DSP core can inform the ARM Host an exception has occurred via the following API:

```
void Fault_Mgmt_notify(void);
```

This function will notify the ARM Host an exception has occurred via Remoteproc and should be the last FM API called within the instrumented exception hook function.

Instrumenting a DSP Application with FM

Create and install an exception hook in the DSP application that utilizes the DSP FM APIs and adds the `fault_mgmt_data` to the Remoteproc `resource_table`.

- In the `.cfg` (Configuro Script) file of the application add following commands to create a section

```
/* Load and use the Fault Management package */ var Fault_mgmt =
xdc.loadPackage('ti.instrumentation.fault_mgmt');

/* Load the Exception and register a exception hook */ var Exception =
xdc.useModule('ti.sysbios.family.c64p.Exception'); Exception.exceptionHook = '&myExceptionHook';
Exception.enablePrint = true;
```

- In a source/header file, create a resource array and exception hook function as follows

```
/* Fault Management Include File */ include
<ti/instrumentation/fault_mgmt/fault_mgmt.h>
```

1. `pragma DATA_SECTION(resources, ".resource_table")`
2. `pragma DATA_ALIGN(resources, 4096)`

```
struct rproc_resource resources[] = {
```

```
{TYPE_COREDUMP_NOTE, 0, (uint32_t) &fault_mgmt_data[0], 0, 0, 0,
FAULT_MGMT_DATA_SIZE, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, "exception_trace_dump"},
```

```
};
```

```
Void myExceptionHook(Void) {
```

```
uint32_t i;
Fm_HaltCfg haltCfg;
uint32_t efr_val;
```

```
/* Copy register status into fault management data region for Host */
Fault_Mgmt_getLastRegStatus();
```

```
memset(&haltCfg, 0, sizeof(haltCfg));
efr_val = CSL_chipReadEFR();
```

```
/* If triggered exception originates from another core through
 * NMI exception don't need to halt processing and notify other
cores
 * since the parent core where the exception originally triggered
via
 * event would notify them. This eliminates recursive exceptions */
if (!(efr_val & 0x80000000)) {
    /* Halt all processing - Only need to be done on one core */
```

```
haltCfg.haltAif = 1;
haltCfg.haltCpdma = 1;
```

1. if EXCLUDE_LINUX_RESOURCES_FROM_HALT

```
haltCfg.haltSGMII = 0;
/* EDMA used by kernel to copy data to/from NAND in UBIFS */
haltCfg.haltEdma3 = 0;
haltCfg.excludedResources = &linuxResources[0];
```

1. else

```
haltCfg.haltSGMII = 1;
haltCfg.haltEdma3 = 1;
haltCfg.excludedResources = NULL;
```

```
1. endif
```

```
Fault_Mgmt_haltIoProcessing(&fmGblCfgParams, &haltCfg);

    for (i = 0; i < fmGblCfgParams.maxNumCores; i++) {
        /* Notify remote DSP cores of exception - WARNING: This will
generate NMI
        * pulse to the remote DSP cores */
        if (i != CSL_chipReadDNUM()) {
            Fault_Mgmt_notify_remote_core(i);
        }
    }

}

/* Notify Host of crash */
Fault_Mgmt_notify();

}
```

A sample test application is provided in `pdk_tci6614_#_##_###_###\packages\instrumentation\fault_mgmt\test`. The test application uses the default resource exclusion list provided with FM in `pdk_tci6614_#_##_###_###\packages\instrumentation\fault_mgmt\device`. The default list has been configured to exclude all Linux owned IO from the halting on exception. This allows the Linux kernel to remain operational after DSP exception so that the core dump can be processed.

Note: It is recommended that the IO halt configuration defined within `#if EXCLUDE_LINUX_RESOURCES_FROM_HALT` be used in addition to halting AIF and CPDMA if Linux must remain active after a DSP exception occurs. This IO halt configuration has been tested with both UBIFS and NFS. The documented configuration shuts down all IO except those needed by Linux to operate, such as EDMA3 (for access to NAND), the SGMII (for Ethernet), and Linux owned CPPI DMAs.

Detecting Crash Event in ARM

The DSP crash detection and notification happens in the remoteproc module running on the ARM. In case of a DSP exception, the Linux kernel (remoteproc module) calls the script `/sbin/dsp_crash_info`. Included in the MCSDK Linux filesystem is a sample script that displays information; a user can customize this script to suit user notification needs.

Generating DSP coredump in ARM


The DSP exceptions can be following

- Software-generated exceptions
- Internal/external exceptions
- Watchdog timer expiration

The DSP produces an ELF formatted core dump in debugfs. This core dump file can be copied from `/debug/remoteproc/dsp-core[0-x]/coredump`.

`cp /debug/remoteproc/dsp-core0/coredump ./coredump` The above commands will generate coredump file with name *coredump* for the DSP.


DSP coredump files can be large and may not fit in the filesystem, thus the cp command can fail. The coredump can be compressed on the fly in the live system, which should reduce amount of space and time needed to capture the coredump. `gzip -c /debug/remoteproc/dsp-core0/coredump > coredump.gz` In this scheme, the coredump need to be uncompressed using *gunzip* after transferring out of the system and before using *dspcoreparse* as described below.

 **Note:** DSP coredump files can be large and may not fit in the filesystem, thus the cp command can fail. In such scenarios, a partial coredump can be captured (upto the size available in the filesystem) using the dd command. An example is shown below: `dd if=/debug/remoteproc/dsp-core3/coredump of=/dev/shm/coredump bs=100M count=1`


Converting and loading core dump image to CCS

The current version of CCS (5.1.1) does not support ELF core dump. The dspcoreparse utility provided in `sc_mcsdk_linux_#_##_###_###/host-tools`, is used to parse the ELF format coredump and generate the crash dump file that can be uploaded to CCS for further analysis.

Copy the *coredump* file generated above to `sc_mcsdk_linux_#_##_###_###/host-tools/dspcoreparse` and run the following command to get crash dump file for CCS.

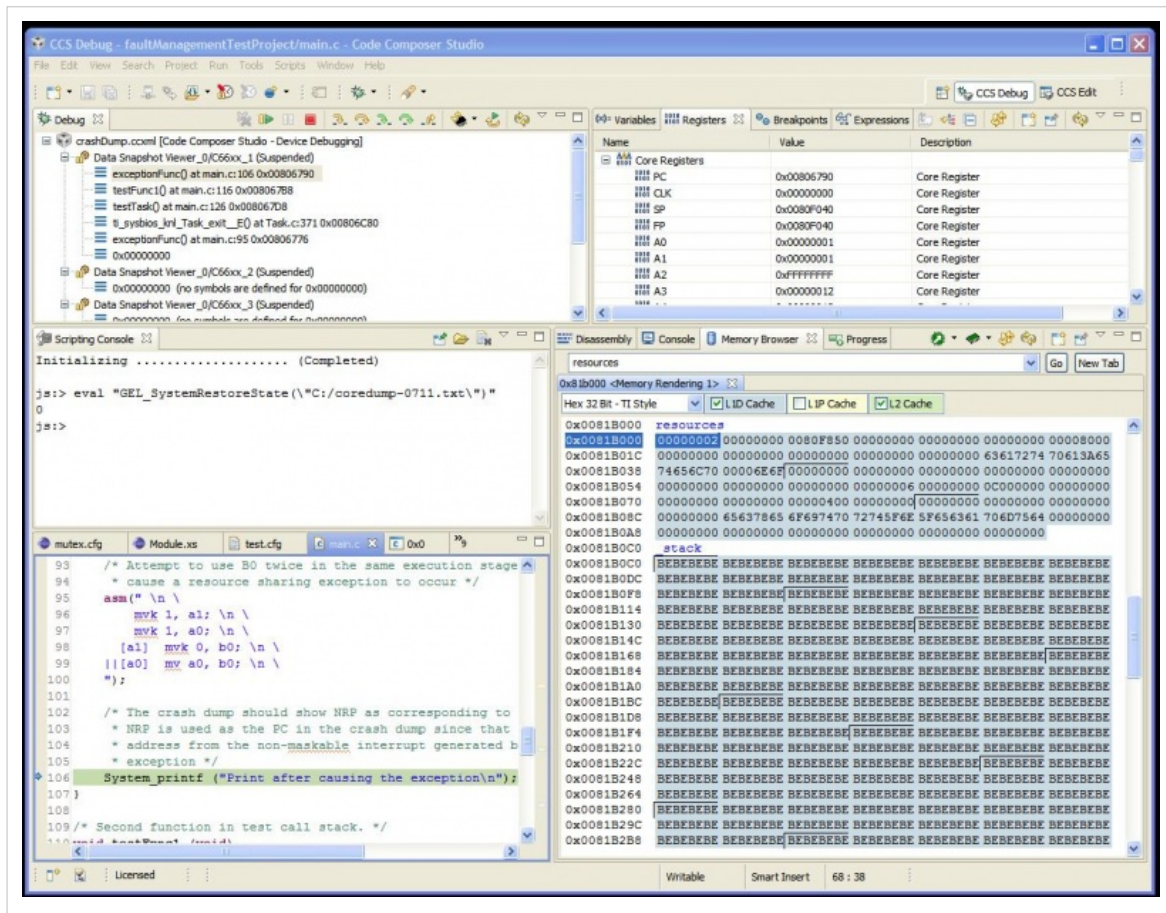
 **Note:** In CCS, a core dump is referred to as a "crash dump".

`user@dspcoreparse $./dspcoreparse -o coredump.txt coredump` This utility can also be used to display the parsed content. However, using CCS provides a graphical view of the information.

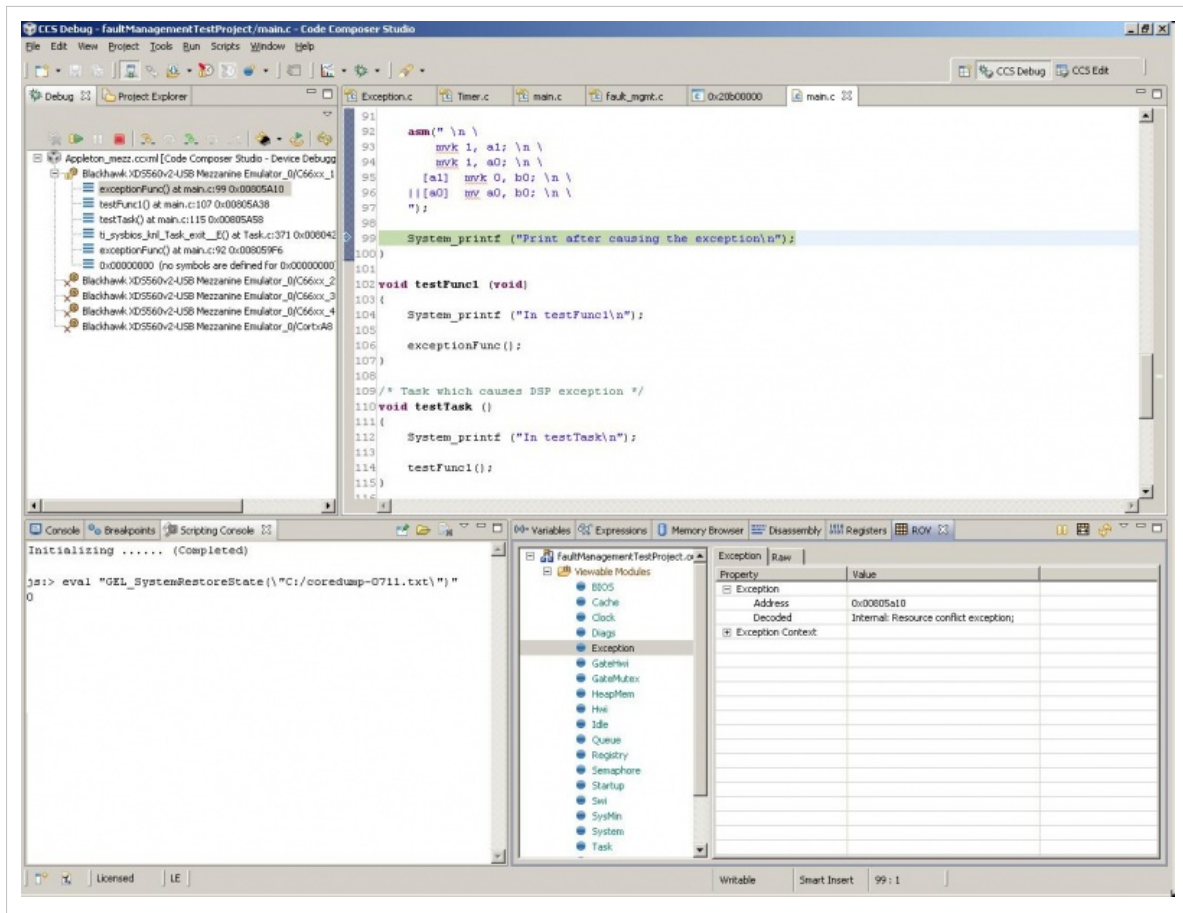
 **Note:** The Windows version of the binary is also available for *dspcoreparse* utility in the SC-MCSDK release package. Look for `sc_mcsdk_linux_#_##_###_###/host-tools\dspcoreparse\dspcoreparse.exe`.

Analyzing DSP crash in CCS

The crash dump file can be loaded to CCS with the symbols of the DSP executable to see register content, stack, memory and other information. Please see Crash_Dump_Analysis^[83] for more information on loading and analyzing a CCS crashdump file.



Further analysis of the crash can be done in CCS by opening ROV. Please see Runtime Object Viewer^[42] for more details on ROV.



Note: The scripting console of CCS sometimes does load the crashdump file with path other than base directory. As an workaround please following command to load the coredump file from CCS scripting console.
`activeDS.expression.evaluate('GEL_SystemRestoreState("/home/ubuntu/coredump.txt")')`

Generating DSP exception from ARM

The remoteproc sysfs can be used to generate exception in DSP from ARM. If the image is hooked with crash indication facility, then the ARM can crash indication from DSP. Use following command to generate an NMI exception to a DSP

echo "exception" > /sys/class/remoteproc/dsp-core<core-id>/state

For example: Following command generates NMI exception to DSP core 3, running a DSP image which is hooked with exception handler as described above.

`root@tc16614-evm:~# echo "exception" > /sys/class/remoteproc/dsp-core3/state`

Technical Support and Product Updates

Technical Support and Forums

This release is not broad market; technical support is directly from TI support team.

Product Updates

This release is not broad market; product updates are available via CDDS.

Frequently Asked Questions, Tips, Hints and Tricks

Using JTAG and CCS

Did you know that CCS will execute all code up to the cinit when loading an out file through the JTAG? This is an option that is enabled, by default, in the Target Configuration file. Initialization code may sometimes execute before this. For example if you hook a function into the SYS/BIOS startup function list it will execute before cinit. If you need to debug that code or it is causing your load to hang (i.e. you do not get the run button highlighted) change the default setting.

Solving the Verify_Init: warnings when executing Demos/NDK Examples from CCS

If you get *Verify_Init*: warnings while executing the Demos/NDK examples (the sample warning output is shown below)

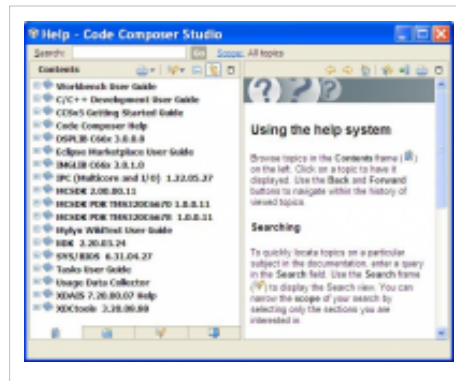
```
[C66xx_0] Verify_Init: Expected 16 entry count for gTxFreeQHnd queue
736, found 62 entries
[C66xx_0] Verify_Init: Expected 0 entry count for gRxQHnd= 704, found
22 entries
[C66xx_0] Verify_Init: Expected 0 entry count for Queue number = 0,
found 1 entries
[C66xx_0] Verify_Init: Expected 0 entry count for Queue number = 704,
found 22 entries
[C66xx_0] Verify_Init: Expected 0 entry count for Queue number = 4095,
found 1 entries
[C66xx_0] Verify_Init: Expected 0 entry count for Queue number = 8192,
found 1 entries
[C66xx_0] Warning:Queue handler Verification failed
```

Please make sure the following when an application is run from CCS environment.

1. SW3, SW4, SW5 and SW5 switches are all set to (ON, ON, ON, ON) mode, the only exception is the SW3[1] switch which is intended to control the endian mode of the EVM. This selects **EMIF16 or Emulation Boot** mode and bypasses the iBL interfering with the CCS executable loaded via CCS.
2. Do a system reset between multiple load and executes of the demo/ndk examples programs
3. Please make sure the corresponding GEL file is executed before the program gets loaded and executed from CCS.


Access to documents

Once SC-MCSDK is installed in the system, many of the documents can be accessed from *CCS->Help->Help Contents*.



Un-installing SC MCSDK

The SC-MCSDK installer installs the un-installer in `sc_mcsdk_xx_00_00_##` directory. The name of the un-installer is *Uninstall-SC-MCSDK-xx.00.00.##.exe*. It also adds links of the un-installer in **Programs->Texas Instruments->BIOS Multi-Core SDK** program menu and in Windows *Add and Remove Programs* menu with name **TI Small Cell Multi-Core SDK**. Selecting any one of the links will start the un-installer and remove the SC-MCSDK components from the system.

 **Note:** Some packages are installed as separate packages (e.g., EDMA3 LLD, DSPLIB, IMGLIB, MATHLIB, SYS/BIOS, IPC) in the system. Due to this, some of the component package installers are not removed after the MCSDK installer is complete; also, to uninstall these packages, please run the corresponding uninstaller.

 **Note:** The un-installer for MCSA will be under CCSv5 installation directory with name *uninstall_dvt.exe*.

Example and documentation of various device peripherals

GPIO

1. The GPIO documentation for KeyStone devices is available from the link General-Purpose Input/Output (GPIO) forKeyStone Devices User's Guide ^[84]
2. The GPIO implementation is provided in file
`pdk_C66##_1_0_0_##\packages\ti\platform\evmc66##\platform_lib\src\evmc66x_gpio.c`
3. The FPGA implementation is provided in file
`pdk_C66##_1_0_0_##\packages\ti\platform\evmc66##\platform_lib\src\evmc66x_fpga.c`
4. In particular the LED operations are in function `fpgaControlUserLEDs()` of file
`pdk_C66##_1_0_0_##\packages\ti\platform\evmc66##\platform_lib\src\evmc66x_fpga.c`

Timer

1. The link [SYSBIOS_Training:Timers and Clocks](#) ^[85] provides detail presentation on configuring timer to get periodic interrupt
2. An older document on SYSBIOS timer implementation is in [DSP/BIOS Timers and Benchmarking Tips](#) ^[86]

DDR3

1. The DDR3 controller users guide is in DDR3 Memory Controller for KeyStone Devices User's Guide ^[87]
2. The DDR3 initialization can be found in the GEL file of the evm

3. The C implementation is in `pdk_C66##_1_0_0_##\packages\ti\platform\evmtci6614\platform_lib\src\platform.c`, function `platform_init()`; Look for `if (p_flags->ddr)` section in the function for the sample code

UART

1. The UART users guide is in Universal Asynchronous Receiver/Transmitter (UART) for KeyStone Devices UG [88]
2. The sample code is in `pdk_tci6614_1_0_0_##\packages\ti\platform\evmtci6614\platform_lib\src\evmc66x_uart.c`

How to submit patches for Linux/U-Boot?

When users require enhancements to support new features or have bug fixes in U-Boot or Linux kernel code, they can submit patches to the `linux-keystone@list.ti.com` mailing list. To add subscription to the list, please send a request to the list and you will be able to send patches to the list once approved. Patches that are enhancing TCI6614 specific code or bug fixes that are TCI6614 related will be merged and maintained by TI. But if a patch touches an upstream code, it is the responsibility of the user to maintain the patch so that it is compatible with new upstream releases.

How to change versions of Linux or U-Boot build with Arago recipe?

In `arago-tci6614.git`, there is a bit bake file (.bb) to build Linux kernel for EVM and Simulator. Following are the files for tci6614 EVM and Simulator.

For EVM

```
recipes/linux/linux-tci6614-evm_git.bb
```

For Simulator

```
recipes/linux/linux-tci6614-sim_git.bb
```

This file has the Linux kernel git specific information that is used by the Arago build. If user wants to build with a different Linux kernel version than what is provided in the release, modify the files given below in `arago-tci6614.git`. For example if there is a patch release that affects only Linux kernel, there will be an updated Linux Kernel tag provided in the patch release. User may want to build with this patch tag.

```
git clone git://arago-project.org/git/projects/arago-tci6614.git
git reset --hard <release tag>
```

Now update the following before attempting Arago build

```
SRCREV = "DEV.SC-MCSDK-02.00.00.07"
```

Change this to the Kernel tag you want to use for build.

```
BRANCH = "releases/02.00.00.07"
```

Change this to the branch on which the kernel tag above is applied

```
KVER = "3.2"
```

This is changed to reflect the upstream kernel version used for the Linux kernel base. Use the commit log to find this version.

Similarly for U-Boot, there is a bit bake file (.bb) to build U-Boot. Following file is used.

```
recipes/u-boot/u-boot-tci6614-evm_git.bb
```

User needs to update the following variable to build with a new U-Boot version than that is provided.

```
SRCREV = "DEV.SC-MCSDK-02.00.00.07"
```

Change this to the u-boot tag user wants Arago to use to build U-Boot.

The tags are provided in the Release Notes.

Why is no timer interrupt when running Linux on Simulator?

The simulator timers are running very slow and there would generate interrupt at a very slow rate. Following work around will force the timer to run at a higher clock rate

```
diff --git a/arch/arm/mach-davinci/tci6614.c b/arch/arm/mach-davinci/tci6614.c index
ad53671..39aed1f 100644 --- a/arch/arm/mach-davinci/tci6614.c +++
b/arch/arm/mach-davinci/tci6614.c @@ -184,8 +184,6 @@
lpsc_clk_enabled(src3_pwr, main_div_chip_smreflex_clk, SRC3_PWR); *
automatically enabled by gpac */ lpsc_clk_enabled(emif4f, main_div_chip_clk1,
EMIF4F); -lpsc_clk_enabled(timer0, clk_modrst0, MODRST0);
-lpsc_clk_enabled(timer1, clk_modrst0, MODRST0); lpsc_clk_enabled(uart0,
clk_modrst0, MODRST0); lpsc_clk_enabled(uart1, clk_modrst0, MODRST0);
lpsc_clk_enabled(aemif, clk_modrst0, MODRST0); @@ -222,6 +220,18 @@
lpsc_clk(rsax2_0, main_div_chip_clk1, RSAX2_0, GEM2); lpsc_clk(tcp2,
main_div_chip_clk2, TCP2, TCP2); lpsc_clk(dxb, main_div_chip_clk3, DXB, DXB);
+static struct clk clk_timer0 = { + .name = "timer0", + .rate = 1000000, +
.flags = ALWAYS_ENABLED, +}; + +static struct clk clk_timer1 = { + .name =
"timer1", + .rate = 1000000, + .flags = ALWAYS_ENABLED, +}; + static struct
clk_lookup clks[] = { CLK(NULL, "ref_clk", &ref_clk), CLK(NULL, "main_pll",
&main_pll),
```

How to integrate the Resource Manager LLD into an app to manage QMSS, CPPI, and PA LLD resources

For instructions on how to manage QMSS, CPPI, and PA LLD resources using the RM LLD please see Resource Manager (RM) LLD.

Information on Arago build infrastructure and BitBake scripting

The Arago build infrastructure is an overlay on OpenEmbedded project. Please see for OpenEmbedded user manual ^[89] and BitBake user manual ^[90] for information on the OpenEmbedded infrastructure and BitBake scripting.

How to check if right content is going to rootfs, devkit and other tasks while writing a recipe

If you run a bitbake file, it generates packages in `~/sc-mcsdk/arago-tmp/deploy/glibc/ipk` directory. For eg: the trace-framework bitbake file generates following packages in armv7a sub-directory

```
ares-ubuntu:ipk$ find . -name *trace-framework*
./armv7a/ti-tci6614-trace-framework-dev_1.0-r9.6_armv7a.ipk
./armv7a/ti-tci6614-trace-framework-static_1.0-r9.6_armv7a.ipk
./armv7a/ti-tci6614-trace-framework-dbg_1.0-r9.6_armv7a.ipk
./armv7a/ti-tci6614-trace-framework_1.0-r9.6_armv7a.ipk
```

Note that, a single recipe can generate multiple packages with extension ipks.

These packages can be pulled into the tasks as needed by adding to the package lists of tasks. For eg:

```
ti-tci6614-trace-framework          in          ~/sc-mcsdk/arago/recipes/tasks/task-arago-sc-mcsdk-base.bb,
ti-tci6614-trace-framework-dev      and          ti-tci6614-trace-framework-static          in
~/sc-mcsdk/arago/recipes/tasks/task-arago-toolchain-scmcsdk-target.bb.
```

To see content of a ipk package use following command in an Linux bash shell

```
ares-ubuntu:armv7a$ dpkg-deb -c
ti-tci6614-trace-framework-static_1.0-r9.6_armv7a.ipk  drwxr-xr-x  root/root  0
2012-10-17 18:16 ./ drwxr-xr-x  root/root  0 2012-10-17 18:16 ./usr/ drwxr-xr-x
root/root  0 2012-10-17 18:16 ./usr/lib/ -rwxr-xr-x  root/root  54004 2012-10-17
18:12 ./usr/lib/libtraceframework.a
```

The above command will give you information on the content of your package and where each items will be placed.

References

- [1] <http://focus.ti.com/docs/training/catalog/events/event.jhtml?sku=OLT110048>
- [2] http://processors.wiki.ti.com/index.php/Keystone_Device_Architecture
- [3] <http://focus.ti.com/docs/training/catalog/events/event.jhtml?sku=OLT110027>
- [4] http://processors.wiki.ti.com/index.php/SYS/BIOS_Online_Training
- [5] http://processors.wiki.ti.com/index.php/SYS/BIOS_1.5-DAY_Workshop
- [6] http://processors.wiki.ti.com/index.php/Multicore_System_Analyzer_Tutorials
- [7] <http://www.ti.com/lit/wp/spry168a/spry168a.pdf>
- [8] http://focus.ti.com/general/docs/video/Portal.tsp?lang=en&entryid=0_xitw1jig
- [9] <http://arago-project.org/git/projects/?p=linux-tci6614.git;a=summary>
- [10] <http://arago-project.org/git/projects/?p=u-boot-tci6614.git;a=summary>
- [11] http://arago-project.org/wiki/index.php/Main_Page
- [12] <http://kernel.org/pub/software/scm/git/docs/git-config.html>
- [13] http://www.gnu.org/software/wget/manual/html_node/Sample-Wgetrc.html
- [14] http://ap-fpdsp-swapps.dal.design.ti.com/index.php/Appleton_Release_Page
- [15] <http://www.denx.de/wiki/DULG/LinuxBootArgs>
- [16] <http://www.linux-mtd.infradead.org/doc/ubi.html>
- [17] <http://www.linux-mtd.infradead.org/doc/general.html>
- [18] <http://www.linux-mtd.infradead.org/doc/ubifs.html>
- [19] http://processors.wiki.ti.com/index.php/MTD_Uutilities#MTD-Utils_Compilation
- [20] http://www.linux-mtd.infradead.org/faq/ubifs.html#L_mkubifs
- [21] <http://arago-project.org/git/projects/?p=linux-tci6614.git;a=blob;f=Documentation/devicetree/bindings/dma/keystone-pktdma.txt;h=92d53cda5c464fd6372dfecdd7f27b97ca956a94;hb=64d93f4a1bb7da4b7af650c9bc6dc599ac771f89>
- [22] <http://arago-project.org/git/projects/?p=linux-tci6614.git;a=blob;f=Documentation/devicetree/bindings/net/keystone-net.txt;h=28115ce03d74dc0a151015fb013d44554e425d8f;hb=64d93f4a1bb7da4b7af650c9bc6dc599ac771f89>
- [23] <http://arago-project.org/git/projects/?p=linux-tci6614.git;a=blob;f=Documentation/devicetree/bindings/hwqueue/keystone-hwqueue.txt;h=76e61d84e25563674da58bcc9980fdaabd71e839;hb=64d93f4a1bb7da4b7af650c9bc6dc599ac771f89>
- [24] <http://www.mjmwired.net/kernel/Documentation/networking/timestamping.txt>
- [25] <http://git.kernel.org/?p=linux/kernel/git/stable/linux-stable.git;a=tree;f=Documentation/networking/timestamping;h=9756a1d6b7fa1945c452e52b14c13cd0ca575090;hb=805a6af8dba5dfdd35ec35dc52ec0122400b2610>
- [26] http://processors.wiki.ti.com/index.php/CI_Dual_EVM_Break_Out_Card
- [27] [http://wfcache.advantech.com/support/DSPM-8303_EVM\(6614\)-3.0/TMDXEVM6614LXE_QSG_rev3.0.pdf](http://wfcache.advantech.com/support/DSPM-8303_EVM(6614)-3.0/TMDXEVM6614LXE_QSG_rev3.0.pdf)
- [28] [http://wfcache.advantech.com/support/DSPM-8303_EVM\(6614\)-3.0/TMDXEVM6614LXE_EVM_TRM_DVD_v12.pdf](http://wfcache.advantech.com/support/DSPM-8303_EVM(6614)-3.0/TMDXEVM6614LXE_EVM_TRM_DVD_v12.pdf)
- [29] <http://processors.wiki.ti.com/index.php/Ctprof>
- [30] <http://git.kernel.org/?p=utils/kernel/kexec/kexec-tools.git;a=summary>
- [31] <http://linux.die.net/man/8/makedumpfile>
- [32] https://access.redhat.com/knowledge/docs/en-US/Red_Hat_Enterprise_Linux/5/html/Deployment_Guide/ch-kdump.html
- [33] <http://wiki.stongswan.org/projects/strongswan/wiki/PKCS11Plugin>
- [34] <http://wiki.stongswan.org/projects/strongswan/wiki/PinSecret>
- [35] http://processors.wiki.ti.com/index.php/CCSV5_Getting_Started_Guide
- [36] http://processors.wiki.ti.com/index.php/Xds_560
- [37] <http://processors.wiki.ti.com/index.php/XDS100>

- [38] <http://focus.ti.com/lit/ug/spru187t/spru187t.pdf>
- [39] <http://focus.ti.com/lit/ug/spru186v/spru186v.pdf>
- [40] <http://processors.wiki.ti.com/index.php/MCSA>
- [41] http://rtsc.eclipse.org/docs-tip/Demo_of_the_RTSC_Platform_Wizard_in_CCsv4
- [42] http://rtsc.eclipse.org/docs-tip/Runtime_Object_Viewer
- [43] http://focus.ti.com/general/docs/video/Portal.tsp?entryid=0_55svdeqr&lang=en
- [44] <http://focus.ti.com/lit/ds/symlink/tms320tc6614.pdf>
- [45] http://www.linux-c6x.org/wiki/index.php/Main_Page
- [46] http://software-dl.ti.com/sdoemb/sdoemb_public_sw/salld/
- [47] <http://focus.ti.com/docs/toolsw/folders/print/telecomlib.html>
- [48] <http://www.ti.com/tool/s2meddus>
- [49] <http://www.eclipse.org/rtsc/>
- [50] <http://processors.wiki.ti.com/index.php/CSL>
- [51] <http://www.opensource.org/licenses/bsd-license.php>
- [52] <http://www.ti.com/lit/sprugr9c>
- [53] <http://www.ti.com/lit/sprugy6>
- [54] <http://www.ti.com/lit/sprugs4>
- [55] <http://www.ti.com/lit/sprugw1>
- [56] <http://www.ti.com/lit/sprugs6a>
- [57] <http://www.ti.com/lit/sprugw8>
- [58] <http://www.ti.com/lit/sprugz1>
- [59] <http://www.ti.com/lit/sprugs0>
- [60] <http://www.ti.com/lit/sprugs2b>
- [61] <http://processors.wiki.ti.com/>
- [62] http://processors.wiki.ti.com/index.php/System_Analyzer_Tutorial_5B
- [63] <http://processors.wiki.ti.com/index.php/CToolsLib>
- [64] http://processors.wiki.ti.com/index.php/MCSDK_Image_Processing_Demonstration_Guide
- [65] http://processors.wiki.ti.com/index.php/MCSDK_Image_Processing_Demonstration_Guide#CToolsLib_IPC-Based
- [66] http://wiki.ci.dal.design.ti.com/index.php/SC-MCSDK_2.0_User_Guide#Using_the_Trace_Decoder_Utility
- [67] <http://processors.wiki.ti.com/index.php/TD>
- [68] <http://processors.wiki.ti.com/index.php/CToolsLib#Examples>
- [69] http://processors.wiki.ti.com/index.php/Programming_the_EDMA3_using_the_Low-Level_Driver_%28LLD%29
- [70] http://software-dl.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/bios/index.html
- [71] http://software-dl.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/ipc/index.html
- [72] <http://www-s.ti.com/sc/techlit/spru523.pdf>
- [73] <http://www-s.ti.com/sc/techlit/spru524.pdf>
- [74] <http://www-s.ti.com/sc/techlit/sprufp2.pdf>
- [75] http://processors.wiki.ti.com/index.php/Network_Developers_Kit_FAQ
- [76] http://processors.wiki.ti.com/index.php/Rebuilding_the_NDK_Core
- [77] http://software-dl.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/ndk/index.html
- [78] <http://focus.ti.com/docs/toolsw/folders/print/bioslinuxmcsdk.html>
- [79] http://processors.wiki.ti.com/index.php/Multicore_System_Analyzer
- [80] http://rtsc.eclipse.org/docs-tip/Main_Page
- [81] http://processors.wiki.ti.com/index.php/SC_MCSDK_2.0_Getting_Started_Guide
- [82] http://processors.wiki.ti.com/index.php/MAD_Utils_User_Guide
- [83] http://processors.wiki.ti.com/index.php/Crash_Dump_Analysis
- [84] <http://www.ti.com/litv/pdf/sprugv1>
- [85] http://processors.wiki.ti.com/index.php/SYS/BIOS_Training:_Timers_and_Clocks
- [86] <http://focus.ti.com/lit/an/spra829/spra829.pdf>
- [87] <http://www.ti.com/litv/pdf/sprugv8b>
- [88] <http://www.ti.com/litv/pdf/sprugp1>
- [89] <http://docs.openembedded.org/usermanual/usermanual.html>
- [90] <http://docs.openembedded.org/bitbake/html/>

Article Sources and Contributors

SC-MCSDK 2.0 User Guide *Source:* <http://wiki.ci.dal.design.ti.com/index.php?oldid=5736> *Contributors:* A0221004, A0271519, A0343404, A0868405, A0875039, AravindBatni, Cchemparathy, Hao, Icesar, Justin32, Karthik5624, M-karicheri2, Pkuttiyam, RajSivarajan, SajeshSaran, Spaulraj, Stater, Tmannan, Wingmankwok

Image Sources and Contributors

Image:C66x-multicore-dsp-arm.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:C66x-multicore-dsp-arm.jpg> *License:* unknown *Contributors:* RajSivarajan

Image:Sc-mcsdk-2.0.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Sc-mcsdk-2.0.jpg> *License:* unknown *Contributors:* M-karicheri2

File:Light_bulb_icon.png *Source:* http://wiki.ci.dal.design.ti.com/index.php?title=File:Light_bulb_icon.png *License:* unknown *Contributors:* RajSivarajan

File:CCS-Help.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:CCS-Help.jpg> *License:* unknown *Contributors:* AravindBatni

File:uia-api.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Uia-api.jpg> *License:* unknown *Contributors:* AravindBatni

Image:Screenshot1.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Screenshot1.jpg> *License:* unknown *Contributors:* M-karicheri2

Image:Screenshot-2.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Screenshot-2.jpg> *License:* unknown *Contributors:* M-karicheri2

Image:Screenshot-3.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Screenshot-3.jpg> *License:* unknown *Contributors:* M-karicheri2

Image:Screenshot-4.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Screenshot-4.jpg> *License:* unknown *Contributors:* M-karicheri2

Image:Tci6614 target config.jpg *Source:* http://wiki.ci.dal.design.ti.com/index.php?title=File:Tci6614_target_config.jpg *License:* unknown *Contributors:* Hao

Image:Launching-target-config-evm.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Launching-target-config-evm.jpg> *License:* unknown *Contributors:* M-karicheri2

Image:Load-gel-evm.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Load-gel-evm.jpg> *License:* unknown *Contributors:* M-karicheri2

Image:Target-connected.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Target-connected.jpg> *License:* unknown *Contributors:* M-karicheri2

Image:Running-dss-script.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Running-dss-script.jpg> *License:* unknown *Contributors:* M-karicheri2

Image:Arm-ready-to-run.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Arm-ready-to-run.jpg> *License:* unknown *Contributors:* M-karicheri2

Image:Linux-mcsdk-2.0-arch.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Linux-mcsdk-2.0-arch.jpg> *License:* unknown *Contributors:* M-karicheri2

Image:Netstack.png *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Netstack.png> *License:* unknown *Contributors:* Spaulraj

File:Qos-tree.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Qos-tree.jpg> *License:* unknown *Contributors:* Spaulraj

File:Qos-new-shaper.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Qos-new-shaper.jpg> *License:* unknown *Contributors:* Spaulraj

File:Shaper-config-details.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Shaper-config-details.jpg> *License:* unknown *Contributors:* Spaulraj

File:Hwqueue1.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Hwqueue1.jpg> *License:* unknown *Contributors:* Spaulraj

File:Hwqueue2.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Hwqueue2.jpg> *License:* unknown *Contributors:* Spaulraj

File:Hwqueue3.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Hwqueue3.jpg> *License:* unknown *Contributors:* Spaulraj

File:Debugfs1.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Debugfs1.jpg> *License:* unknown *Contributors:* Spaulraj

File:Debugfs2.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Debugfs2.jpg> *License:* unknown *Contributors:* Spaulraj

File:Debugfs3.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Debugfs3.jpg> *License:* unknown *Contributors:* Spaulraj

File:Debugfs4.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Debugfs4.jpg> *License:* unknown *Contributors:* Spaulraj

File:Debugfs5.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Debugfs5.jpg> *License:* unknown *Contributors:* Spaulraj

File:Screenshot-Gnuplot tci6614 BW L2.png *Source:* http://wiki.ci.dal.design.ti.com/index.php?title=File:Screenshot-Gnuplot_tci6614_BW_L2.png *License:* unknown *Contributors:* A0343404

File:TraceFramework_ARM.jpg *Source:* http://wiki.ci.dal.design.ti.com/index.php?title=File:TraceFramework_ARM.jpg *License:* unknown *Contributors:* AravindBatni

Image:secure_store.png *Source:* http://wiki.ci.dal.design.ti.com/index.php?title=File:Secure_store.png *License:* unknown *Contributors:* A0875039, SajeshSaran

Image:MCSDK200SoftwareStack.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:MCSDK200SoftwareStack.jpg> *License:* unknown *Contributors:* Hao

Image:Rmm structure overview.JPG *Source:* http://wiki.ci.dal.design.ti.com/index.php?title=File:Rmm_structure_overview.JPG *License:* unknown *Contributors:* Justin32

Image:TraceFramework_DSP.jpg *Source:* http://wiki.ci.dal.design.ti.com/index.php?title=File:TraceFramework_DSP.jpg *License:* unknown *Contributors:* AravindBatni

File:Ctools uclib bitmap.png *Source:* http://wiki.ci.dal.design.ti.com/index.php?title=File:Ctools_uclib_bitmap.png *License:* unknown *Contributors:* Karthik5624

Image:Ndkarch.png *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Ndkarch.png> *License:* unknown *Contributors:* Hao

Image:Importplatformlibproject.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Importplatformlibproject.jpg> *License:* unknown *Contributors:* Hao

Image:Setprofileplatformlibproject.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Setprofileplatformlibproject.jpg> *License:* unknown *Contributors:* Hao

File:Warning.png *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Warning.png> *License:* unknown *Contributors:* RajSivarajan

Image:Import Project.JPG *Source:* http://wiki.ci.dal.design.ti.com/index.php?title=File:Import_Project.JPG *License:* unknown *Contributors:* Hao

Image:Import NIMU.JPG *Source:* http://wiki.ci.dal.design.ti.com/index.php?title=File:Import_NIMU.JPG *License:* unknown *Contributors:* Hao

Image:NIMU debug be set active.JPG *Source:* http://wiki.ci.dal.design.ti.com/index.php?title=File:NIMU_debug_be_set_active.JPG *License:* unknown *Contributors:* Hao

Image:Client big endian.JPG *Source:* http://wiki.ci.dal.design.ti.com/index.php?title=File:Client_big_endian.JPG *License:* unknown *Contributors:* Hao

Image:Client big endian RTSC.JPG *Source:* http://wiki.ci.dal.design.ti.com/index.php?title=File:Client_big_endian_RTSC.JPG *License:* unknown *Contributors:* Hao

Image:Client running.JPG *Source:* http://wiki.ci.dal.design.ti.com/index.php?title=File:Client_running.JPG *License:* unknown *Contributors:* Hao

Image:Ndkdosbox.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Ndkdosbox.jpg> *License:* unknown *Contributors:* Hao

Image:Ndkdosboxbuild.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Ndkdosboxbuild.jpg> *License:* unknown *Contributors:* Hao

Image:Ndkdosboxbuilding.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Ndkdosboxbuilding.jpg> *License:* unknown *Contributors:* Hao

Image:Projectsettingshellworld.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Projectsettingshellworld.jpg> *License:* unknown *Contributors:* Hao

Image:Includepathhellworld.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Includepathhellworld.jpg> *License:* unknown *Contributors:* Hao

Image:Linkerinputhellworld.jpg *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Linkerinputhellworld.jpg> *License:* unknown *Contributors:* Hao

Image:LedRtscProject.JPG *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:LedRtscProject.JPG> *License:* unknown *Contributors:* Hao

Image:Tci6614 uboot arch diagram.jpg *Source:* http://wiki.ci.dal.design.ti.com/index.php?title=File:Tci6614_uboot_arch_diagram.jpg *License:* unknown *Contributors:* Hao

Image:Tci6614 uboot nand boot.png *Source:* http://wiki.ci.dal.design.ti.com/index.php?title=File:Tci6614_uboot_nand_boot.png *License:* unknown *Contributors:* Hao

Image:Tci6614 uboot tftp boot.png *Source:* http://wiki.ci.dal.design.ti.com/index.php?title=File:Tci6614_uboot_tftp_boot.png *License:* unknown *Contributors:* Hao

Image:Post.png *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Post.png> *License:* unknown *Contributors:* Hao

Image:DSP_crash_stack.jpg *Source:* http://wiki.ci.dal.design.ti.com/index.php?title=File:DSP_crash_stack.jpg *License:* unknown *Contributors:* SajeshSaran

Image:Crash dump with ROV.JPG *Source:* http://wiki.ci.dal.design.ti.com/index.php?title=File:Crash_dump_with_ROV.JPG *License:* unknown *Contributors:* Justin32

Image:Ccs-help.png *Source:* <http://wiki.ci.dal.design.ti.com/index.php?title=File:Ccs-help.png> *License:* unknown *Contributors:* Hao