BIOS-MCSDK

User Guide

Contents

Articles

References

Article Licenses

BIOS MCSDK 2.0 User Guide

BIOS Multicore Software Development Kit
Version 2.x
User's Guide

Last updated: //

Introduction

The BIOS Multicore Software Development Kit (MCSDK) provides the core foundational building blocks that facilitate application software development on TI's high performance and multicore DSPs. 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
- Run-time libraries
- 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 MCSDK. 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.

Useful Tip

It is expected the user has gone through the *EVM Quick Start Guide* 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 MCSDK Getting Started Guide and have installed both CCS and the MCSDK.

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
MCSDK	Texas Instruments Multi-Core Software Development Kit
NDK	Texas Instruments Network Development Kit (IP Stack)
NIMU	Network Interface Management Unit
PDK	Texas Instruments Programmers 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, TMS320C6678 refers to the C6678 DSP processor and TMDSEVM6678L refers to the actual hardware EVM that the processor is on.

Supported Devices/Platforms

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

Platform Development Kit	Supported Devices	Supported EVM
C6657	TMS320C6657 [1]	TMDXEVM6657L [2], TMDXEVM6657LE [2]
C6670	TMS320C6670 [3], TMS320TCI6618 [4]	TMDSEVM6670L [5], TMDSEVM6670LE [5],TMDSEVM6670LXE [5], TMDSEVM6618LXE
C6678	TMS320C6678 [6], TMS320TCI6608 [7]	TMDSEVM6678L [8], TMDSEVM6678LE [8], TMDSEVM6678LXE [8]

Other Resources

Training

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

Link	Description
MCSDK Overview Online [9]	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. The **Mandarin** version of this training can be found here [10].
KeyStone Architecture Wiki [11]	KeyStone Architecture Overview Mediawiki
KeyStone Architecture Online [12]	C66x Multicore SOC Online Training for KeyStone Devices
SYS/BIOS Online [13]	SYS/BIOS Online Training
SYS/BIOS 1.5 Day [14]	SYS/BIOS 1.5-DAY Workshop
MCSA Online [15]	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 [16]	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 [17]	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
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.
MCSDK Getting Started Guide	Discusses how to install the BIOS-MCSDK and access the demonstration application.
TMDSEVM66xxL 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 *Hardware - EVM Overview*.

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 [18]	How to get up and running with CCS v5
XDS560 Emulator Information [19]	Information on XDS560 emulator
XDS100 Emulator Information [20]	Information on XDS100 emulator
TMS320C6000 Optimizing Compiler v 7.3 [21]	Everything you wanted to know about the compiler, assembler, library-build process and C++ name demangler.
TMS320C6000 Assembly Language Tools v 7.3 [22]	More in-depth information on the assembler, linker command files and other utilities.
Multi-core System Analyzer [23]	How to use and integrate the system analyzer into your code base.
Eclipse Platform Wizard [24]	How to create a platform for RTSC. The demo uses CCSv4 but the platform screens are the same in CCSv5.
Runtime Object Viewer [25]	How to use the Object Viewer for Eclipse Based Debugging.

Hardware - EVM Overview

The following resources provide information about the EVM.

Document	Description
Introducing the C66x Lite EVM Video [26]	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 C66x multicore DSPs based on the KeyStone architecture.
TMDSEVM6657L documentation and support	Discusses the technical aspects of your EVM including board block diagram, DIP Switch Settings, memory addresses and range, power supply and basic operation.
TMDSEVM6670L documentation and support [27]	
TMDSEVM6678L documentation and support [28]	
TMDSEVM6618LXE documentation and support (TBD)	

Hardware - Processor Overview

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

Document	Description
TMS320C6657 Data Manual [29]	Data manual for specific TI DSP
TMS320C6670 Data Manual [30]	
TMS320C6678 Data Manual [31]	
TMS320TCI6618 Data Manual [32]	

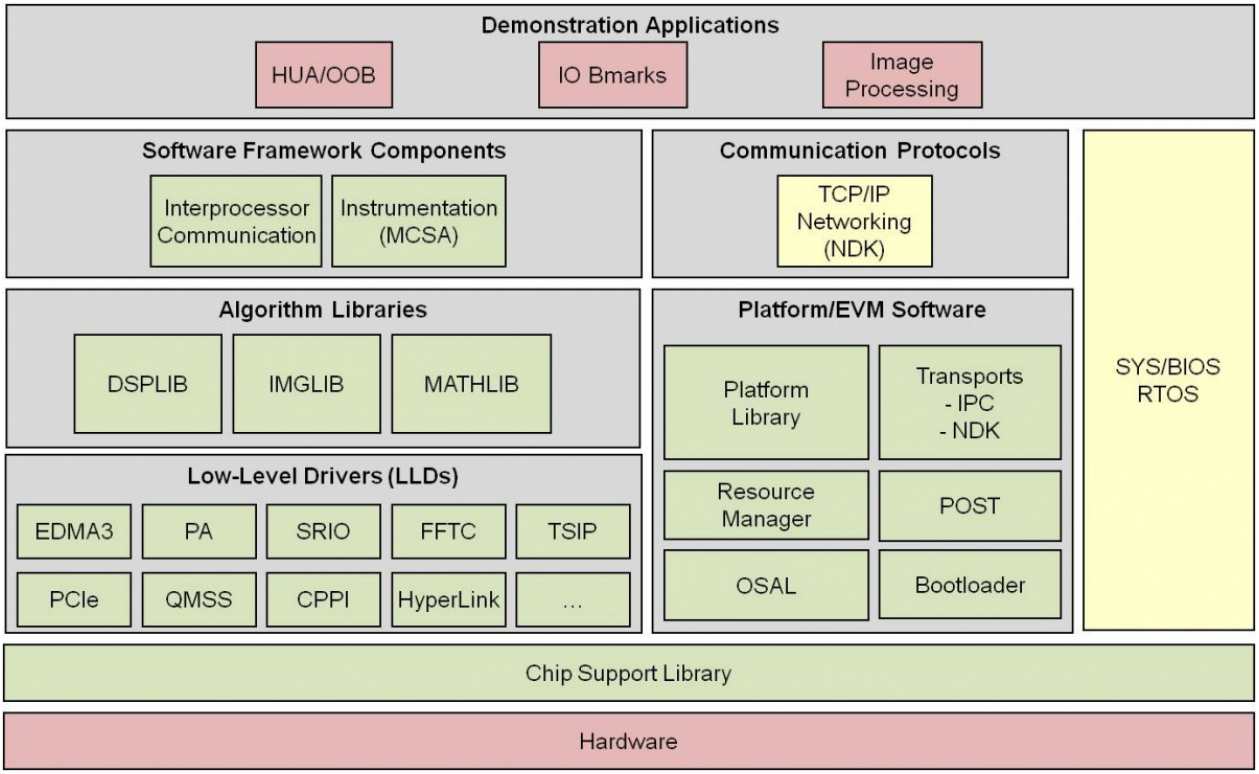
Related Software

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

Link	Description
Security Accelerator LLD [33]	Download page for Security Accelerator (SA) low level driver
C6x DSP Linux Project [34]	Community site for C6x DSP Linux project
Telecom Libraries [35]	TI software folder for information and download of Telecom Libraries (Voice, Fax, etc) for TI processors.
c66x Speech and Video Codecs [36]	TI software folder for information and download of Speech and Video codecs for c66x.
Medical Imaging Software Tool Kits [37]	TI software folder for information and download of medical imaging software tool kits for TI processors.
c6x Software Libraries [38]	Mediawiki providing an overview of available software libraries for TI's c6x family of DSP processors.
Multicore Video Infrastructure Demonstration Application [39]	TI software folder for information and download of multicore video infrastructure demonstration application using the BIOS-MCSDK.

Software Overview

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



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	pdk_<platform>_w_xx_yy_zz/packages/ti/csl/
All LLD (except EDMA3)	pdk_<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_ww_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 Libary	pdk_<platform>_w_xx_yy_zz/packages/ti/platform/<device>/platform_lib
Resource Manager	pdk_<platform>_w_xx_yy_zz/packages/ti/platform/resource_mgr.h (Note: There is also a RM LLD provided for resource management)
Platform OSAL	pdk_<platform>_w_xx_yy_zz/packages/ti/platform/platform.h
Transports	pdk_<platform>_w_xx_yy_zz/packages/ti/transport/ipc/qmss/ - QMSS IPC Transport
	pdk_<platform>_w_xx_yy_zz/packages/ti/transport/ipc/srio/ - SRIO IPC Transport
	pdk_<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/
Demonstration Applications	
HUA "Out of Box" Demo	mcsdk_w_xx_yy_zz/demos/hua/
Image Processing	mcsdk_w_xx_yy_zz/demos/image_processing/

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 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 pdk_C####_#_#_#_#/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 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

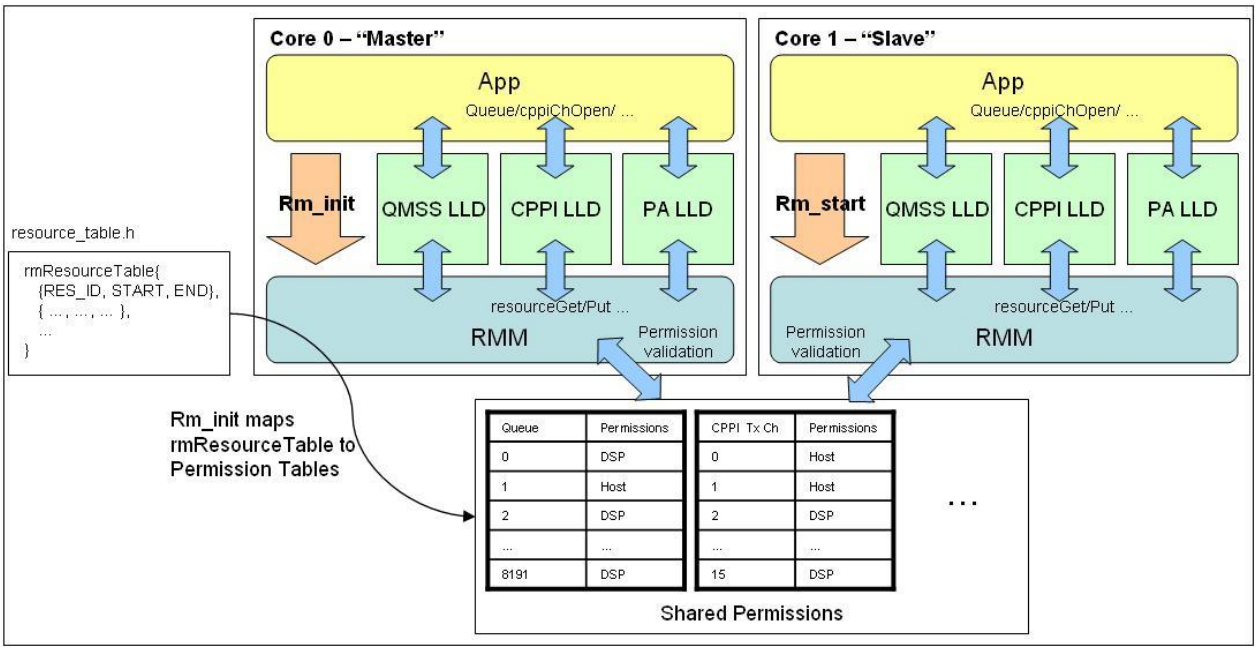
- Transmit Channels
- Receive Channels
- Receive Flows

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 a 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 */
0u,
/** No use permissions */
0u,
}
```

};

- 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 initalize 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 ".cppi" 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 compatability 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 cppiStartCfg;
```

```
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 (&cppiGblCfgParams);
```

```
/* 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 (the API of one module 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_C66##_INSTALL_DIR)\packages\ti\csl directory.

Note: The CSL is built 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_c6678x_<version>\packages\ti\csl pdk_c6670x_<version>\packages\ti\csl pdk_c6657x_<version>\packages\ti\csl
Project Type	Eclipse RTSC [40]
Endian Support	Little & Big
Linker Path	$(TI_PDK_C6678_INSTALL_DIR)\packages\ti\csl $(TI_PDK_C6670_INSTALL_DIR)\packages\ti\csl $(TI_PDK_C6657_INSTALL_DIR)\packages\ti\csl
Linker Sections	.vecs , .switch, .args, .cio
Section Preference	L2 Cache
Include Paths	$(TI_PDK_C6678_INSTALL_DIR)\packages\ti\csl $(TI_PDK_C6670_INSTALL_DIR)\packages\ti\csl $(TI_PDK_C6657_INSTALL_DIR)\packages\ti\csl
Reference Guides	See docs under Install Directory
Support	Technical Support
Additional Resources	Chip support library [41]
Downloads	Product Updates
License	BSD [42]

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_C66##_INSTALL_DIR)\packages\ti\drv directory.

The following table shows PDK LLD vs. SoC availability.

Driver	C6678	C6670/ TCI6618	C6657
CSL	X	X	X
RM	X	X	X
QMSS	X	X	X
PKTDMA (CPPI)	X	X	X
PA	X	X	
SA	X	X	
SRIO	X	X	X
PCIe	X	X	X
Hyperlink	X	X	X
TSIP	X		
EDMA3	X	X	X
FFTC		X	

TCP3d		X	X
TCP3e		X	
BCP		X	
AIF2		X	
EMAC			X

Driver Library Summary	
Component Type	Library
Install Package	PDK
Install Directory	pdk_c6678x_<version>\packages\ti\drv pdk_c6670x_<version>\packages\ti\drv pdk_c6657x_<version>\packages\ti\drv
Project Type	Eclipse RTSC [40]
Endian Support	Little & Big
Linker Path	$(TI_PDK_C6678_INSTALL_DIR)\packages\ti\drv $(TI_PDK_C6670_INSTALL_DIR)\packages\ti\drv $(TI_PDK_C6657_INSTALL_DIR)\packages\ti\drv
Linker Sections	N/A
Section Preference	N/A
Include Paths	$(TI_PDK_C6678_INSTALL_DIR)\packages\ti\drv $(TI_PDK_C6670_INSTALL_DIR)\packages\ti\drv $(TI_PDK_C6657_INSTALL_DIR)\packages\ti\drv
Reference Guides	See docs under Install Directory
Support	Technical Support
Additional Resources	Chip support library [41]
Downloads	Product Updates
License	BSD [42]

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_C66##_INSTALL_DIR)\packages\ti\drv\rm\docs\rmlldDocs.chm
Release Notes	$(TI_PDK_C66##_INSTALL_DIR)\docs\ReleaseNotes_RM_LLD.pdf

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 device driver requirements 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) [43]
Downloads	Product Updates
License	BSD [42]

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 [44]
LLD Users Guide	$(TI_PDK_C66##_INSTALL_DIR)\packages\ti\drv\cppi\docs\ CPPI_QMSS_LLD_SDS.pdf
API Reference Manual	$(TI_PDK_C66##_INSTALL_DIR)\packages\ti\drv\cppi\docs\cppilldDocs.chm
Release Notes	$(TI_PDK_C66##_INSTALL_DIR)\docs\ReleaseNotes_CPPI_LLD.pdf

Note: As of BIOS-MCSDK 2.0.8 applications that configure the CPPI OSAL to allocate memory from an IPC SharedRegion heap may need to change. Changes are required only if the Cppi_init() function executes prior to the Ipc_attach() function. If the latter case occurs the Cppi_init() will attempt to allocate a block of memory from the SharedRegion heap located in shared memory. However, because Ipc_attach() has not executed yet the SharedRegion will not be configured. This will cause a default to allocate from a local heap in L2. The block pointer returned by this local heap will at some point be used by a remote core, expecting the CPPI heap to be shared. This will corrupt anything located in the remote core's memory located at the value of the block pointer.

Applications which suffer from the latter issue must create a static heap at compile time for use by CPPI. The heap can be provided to the CPPI LLD via new APIs. In the application source code at the following:

```
define SIZE_CPPI_HEAP 1024 /* Should be sized large enough to fit all shared
```

```
* CPPI channel and flow objects */
```

/* Statically created shared heap for CPPI since IPC does create a

```
* shared heap for SharedRegion prior to Ipc_attach */
```

1. pragma DATA_SECTION (cppiHeap, ".cppi_heap");
2. pragma DATA_ALIGN (cppiHeap, 128)

UInt8 cppiHeap[SIZE_CPPI_HEAP];

Int32 systemInit (Void) {

```
Cppi_InitCfg cppiHeapInit; /* Static CPPI heap */
```

```
...
```

```
/* Configure Cppi_init() parameters to configure static heap */
cppiHeapInit.heapParams.staticHeapBase = &cppiHeap[0];
cppiHeapInit.heapParams.staticHeapSize = SIZE_CPPI_HEAP;
cppiHeapInit.heapParams.heapAlignPow2 = 7; /* Power of 7 (128 byte) */
cppiHeapInit.heapParams.dynamicHeapBlockSize = -1; /* Shut off malloc if block runs out */
result = Cppi_initCfg (&cppiGblCfgParams, &cppiHeapInit);
if (result != CPPI_SOK)
{
```

```
Error...
}
```

```
...
```

}

Int main(Int argc, Char* argv[]) {

```
Int32 result = 0;
```

```
selfId = CSL_chipReadReg (CSL_CHIP_DNUM);
```

```
/* System initializations for each core. */
if (selfId == 0)
{
/* SRIO, QMSS, and CPPI system wide initializations are run on
* this core */
result = systemInit();
}
```

```
...
```

}

In the application linker command file or XDC configuration place the static CPPI heap into shared memory.

If using XDC .cfg file to add sections to the linker command file: `Program.sectMap[".cppi_heap"] = new Program.SectionSpec(); Program.sectMap[".cppi_heap"] = "MSMCSRAM";`

If explicitly placing the heap in the application linker command file: `.cppi_heap: load >> MSMCSRAM`

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 a 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 [44]
LLD Users Guide	$(TI_PDK_C66##_INSTALL_DIR)\packages\ti\drv\qmss\docs\ CPPI_QMSS_LLD_SDS.pdf
API Reference Manual	$(TI_PDK_C66##_INSTALL_DIR)\packages\ti\drv\qmss\docs\qmsslldDocs.chm
Release Notes	$(TI_PDK_C66##_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 MCSDK. See download link in the Related Software [45] link above.

Additional documentation can be found in:

Document	Location
Hardware Peripheral Users Guide	User Guide [46]
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_lld_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 [47]
LLD Users Guide	$(TI_PDK_C66##_INSTALL_DIR)\packages\ti\drv\pa\docs\pa_sds.pdf
API Reference Manual	$(TI_PDK_C66##_INSTALL_DIR)\packages\ti\drv\pa\docs\paDocs.chm
Release Notes	$(TI_PDK_C66##_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 [48]
LLD Users Guide	$(TI_PDK_C66##_INSTALL_DIR)\packages\ti\drv\srio\docs\SRIO_SDS.pdf
API Reference Manual	$(TI_PDK_C66##_INSTALL_DIR)\packages\ti\drv\sa\docs\srioDocs.chm
Release Notes	$(TI_PDK_C66##_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 [49]
API Reference Manual	$(TI_PDK_C66##_INSTALL_DIR)\packages\ti\drv\pcie\docs\pcieDocs.chm
Release Notes	$(TI_PDK_C66##_INSTALL_DIR)\docs\ReleaseNotes_PCIE_LLD.pdf

Antenna Interface (AIF2)

This AIF2 low level driver aims at generalizing the configuration of AIF2 for different modes (CPRI/OBSAI/ABTLib/Generic packet, WCDMA/LTE/Dual mode). The AIF2 LLD makes use of Chip Support Library and CPPI/QMSS Low Level Drivers (LLDs). This driver is only supported in C6670.

Additional documentation can be found in:

Document	Location
Hardware Peripheral Users Guide	User Guide [50]
LLD Users Guide	$(TI_PDK_C6670_INSTALL_DIR)\packages\ti\drv\aif2\docs\AIF2-c6670_usersguide.pdf
API Reference Manual	$(TI_PDK_C6670_INSTALL_DIR)\packages\ti\drv\aif2\docs\AIF2-c6670_apireferenceguide.html
Release Notes	$(TI_PDK_C6670_INSTALL_DIR)\docs\ReleaseNotes_AIF2_LLD.pdf

TSIP

The TSIP is multi-link serial interface consisting of a maximum of eight transmit data signals (or links), eight receive data signals (or links), two frame-sync input signals, and two serial clock inputs. Internally, the TSIP offers multiple channels of time-slot data management and multi-channel DMA capability that allow individual time-slots to be selectively processed. The LLD provides a well-defined standard interface which allows application to configure the peripheral.

Additional documentation can be found in:

Document	Location
Hardware Peripheral Users Guide	User Guide [51]
API Reference Manual	$(TI_PDK_C66##_INSTALL_DIR)\packages\ti\drv\tsip\docs\tsipDocs.chm
Release Notes	$(TI_PDK_C66##_INSTALL_DIR)\docs\ReleaseNotes_TSIP_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.

Note: Hyperlink is a point-to-point peripheral, so can only support communication between two devices.

Additional documentation can be found in:

Document	Location
Hardware Peripheral Users Guide	User Guide [52]
API Reference Manual	$(TI_PDK_C66##_INSTALL_DIR)\packages\ti\drv\hyplnk\docs\hyplnkDocs.chm
Release Notes	$(TI_PDK_C66##_INSTALL_DIR)\docs\ReleaseNotes_HYPLNK_LLD.pdf

Ethernet Media Access Controller (EMAC)

The device driver exposes a set of well defined API which is used by the application layer to send and receive data packets via the EMAC peripheral, and configure and monitor PHY via the MDIO peripheral. The driver also exposes a set of well defined OS abstraction API which is used to ensure that the driver is OS independent and portable. The EMAC driver uses the CSL EMAC functional layer for all EMAC MMR accesses.

Additional documentation can be found in:

Document	Location
Hardware Peripheral Users Guide	User Guide
API Reference Manual	$(TI_PDK_C66##_INSTALL_DIR)\packages\ti\drv\emac\docs\doxygen\emac.chm
Release Notes	$(TI_PDK_C66##_INSTALL_DIR)\docs\ReleaseNotes_EMAC_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 [53]
LLD Users Guide	$(TI_PDK_C66##_INSTALL_DIR)\packages\ti\drv\bcp\docs\BCP_SDS.pdf
API Reference Manual	$(TI_PDK_C66##_INSTALL_DIR)\packages\ti\drv\bcp\docs\bcpDocs.chm
Release Notes	$(TI_PDK_C66##_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 the 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 [54]
LLD Users Guide	$(TI_PDK_C66##_INSTALL_DIR)\packages\ti\drv\tcp3d\docs\TCP3D_DriverSDS.pdf
API Reference Manual	$(TI_PDK_C66##_INSTALL_DIR)\packages\ti\drv\tcp3d\docs\TCP3D_DRV_APIIF.chm
Release Notes	$(TI_PDK_C66##_INSTALL_DIR)\packages\ti\drv\tcp3d\docs\ReleaseNotes_TCP3DDriver.pdf

Turbo Coprocessor Encoder (TCP3e)

The TCP3 Encoder driver provides a well-defined standard interface which allows the application to send code blocks for encoding and receive encoded bits via EDMA3 transfers.

Additional documentation can be found in:

Document	Location
Hardware Peripheral Users Guide	User Guide [55]
LLD Users Guide	$(TI_PDK_C66##_INSTALL_DIR)\packages\ti\drv\tcp3e\docs\TCP3E_DriverSDS.pdf
API Reference Manual	$(TI_PDK_C66##_INSTALL_DIR)\packages\ti\drv\tcp3e\docs\TCP3E_DRV_APIIF.chm
Release Notes	$(TI_PDK_C66##_INSTALL_DIR)\packages\ti\drv\tcp3e\docs\ReleaseNotes_TCP3EDriver.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. as 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 [56]
LLD Users Guide	$(TI_PDK_C66##_INSTALL_DIR)\packages\ti\drv\fftc\docs\FFTC_SDS.pdf
API Reference Manual	$(TI_PDK_C66##_INSTALL_DIR)\packages\ti\drv\fftc\docs\fftcDocs.chm
Release Notes	$(TI_PDK_C66##_INSTALL_DIR)\packages\ti\drv\fftc\docs\ReleaseNotes_FFTCDriver.pdf

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 (e.g., ti.platform.evm6678l.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 (e.g., ti.platform.evm6678l.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.
3. lite library (e.g., ti.platform.evm6678l.lite.lib) - \platform_lib\lib\debug, not needed for regular platform development - this is used to link for the Power On Self Test (POST) application.

Platform Library Summary	
Component Type	Library
Install Package	PDK for C66X
Install Directory	pdk_c6657_<version>\packages\ti\platform\evm6657l\platform_lib pdk_c6670_<version>\packages\ti\platform\evm6670l\platform_lib pdk_c6678_<version>\packages\ti\platform\evm6678l\platform_lib
Project Type	CCS [18]
Endian Support	Little
Library Name	Select for the C6678L EVM ti.platform.evm6678l.ae66 (little)
Linker Path	$(TI_PDK_C6678_INSTALL_DIR)\packages\ti\platform\evmc6678l\platform_lib\lib\debug - for debug version $(TI_PDK_C6678_INSTALL_DIR)\packages\ti\platform\evmc6678l\platform_lib\lib\release - for release version (similar paths for C6670, C6657)
Linker Sections	platform_lib
Section Preference	none
Include Paths	$(TI_PDK_C6678_INSTALL_DIR)\packages\ti\platform (similar paths for C6670, C6657) platform.h defines the interface
Reference Guides	See docs under Install Directory
Support	Technical Support
Additional Resources	Texas Instruments Embedded Processors Wiki [57]

Downloads	Product Updates
License	BSD [42]

Platform Library Migration Information

The below table provides the migration information for the *platform library* for maintenance updates to the BIOS-MCSDK 2.0.0 production release:

Release	API Change	Migration Notes
BIOS-MCSDK 2.0.2	Added Platform_STATUS platform_get_emac_info(uint32_t port_num, PLATFORM_EMAC_EXT_info * emac_info)	
	Deprecated the efuse_mac_address[6], eeprom_mac_address[6] fields in EMAC_info structure as MAC address is now defined in the new data structure PLATFORM_EMAC_EXT_info	Use PLATFORM_EMAC_EXT_info structure for MAC address
	Added Platform_STATUS platform_get_macaddr(PLATFORM_MAC_TYPE type, uint8_t * mac_address);	
BIOS-MCSDK 2.0.5	No Platform library API change	Updated the main PLL, DDR3 PLL and PA PLL sequences. Please refer to *platform_lib\src\evm667#.c* file for the updates.

Transport

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

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

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 real-time 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 [40]
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 [13] SYS/BIOS 1.5-DAY Workshop [14] Eclipse RTSC Home [40]
Downloads	SYS/BIOS Downloads [58]
License	BSD [42]

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 [40]
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 [40]
Downloads	IPC Downloads [59]
License	BSD [42]

IPC Transports

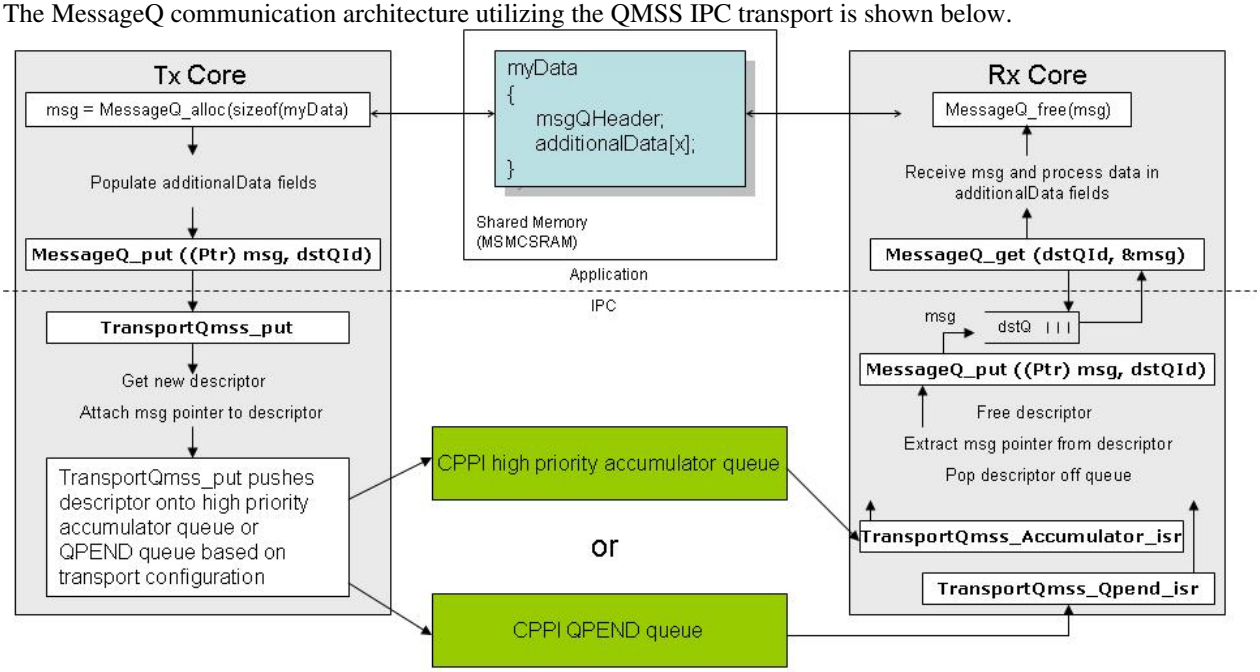
QMSS IPC Transport

The QMSS transport is an additional transport for IPC. The QMSS transport can be used by MessageQ to send data between tasks and cores via the QMSS IP block. This package has a QMSS transport unit test/benchmark example for all supported platforms.

Note: This module is only intended to be used with IPC MessageQ. As such, users should not tie up to its API directly.

QMSS IPC Transport Summary	
Component Type	Library
Install Package	PDK_C6678_INSTALL_DIR
Install Directory	mcsdk_<version>\packages\ti\transport\ipc\qmss
Project Type	Eclipse RTSC [40]
Endian Support	Little, Big
Library Name	ti.transport.ipc.qmss.transports.ae66 (little) ti.transport.ipc.qmss.transports.ae66e (big)
Linker Path	$(TI_PDK_C6678_INSTALL_DIR)\packages\ti\transport\ipc\qmss\transports\lib\whole_program_debug
Reference Guides	None
Support	Technical Support
Additional Resources	The QMSS IPC Transport benchmark example is available in $(TI_PDK_C6678_INSTALL_DIR)\packages\ti\transport\ipc\examples\qmssIpcBenchmark
Downloads	[60]
License	BSD [42]

The MessageQ communication architecture utilizing the QMSS IPC transport is shown below.



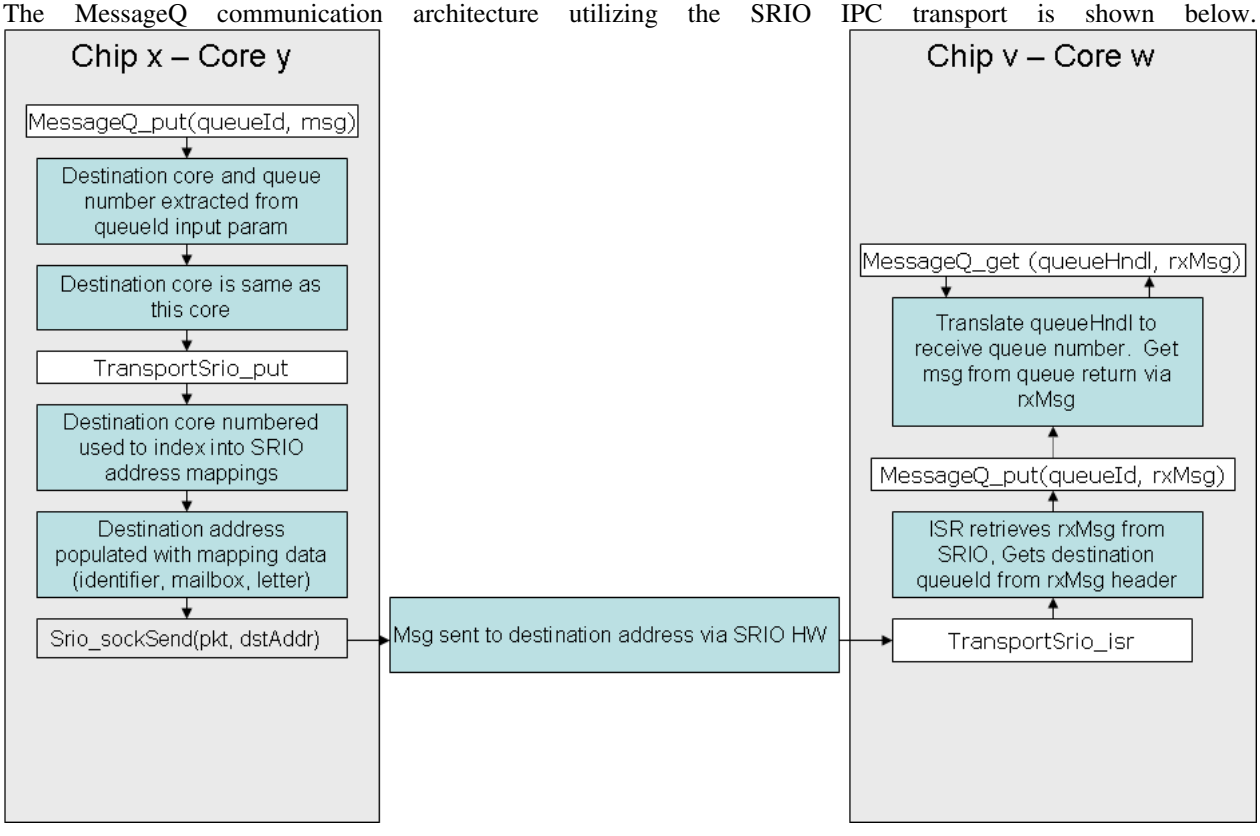
SRIO IPC Transport

The SRIO transport is an additional transport for IPC. The SRIO transport can be used by MessageQ to send data between tasks, cores, and chips via the SRIO IP block. This package has SRIO transport unit test and benchmark examples for all supported platforms.

Note: This module is only intended to be used with IPC MessageQ. As such, users should not tie up to its API directly.

SRIO IPC Transport Summary	
Component Type	Library
Install Package	PDK_C6678_INSTALL_DIR
Install Directory	mcsdk_<version>\packages\ti\transport\ipc\srio
Project Type	Eclipse RTSC [40]
Endian Support	Little, Big
Library Name	ti.transport.ipc.srio.transports.ae66 (little) ti.transport.ipc.srio.transports.ae66e (big)
Linker Path	$(TI_PDK_C6678_INSTALL_DIR)\packages\ti\transport\ipc\srio\transports\lib\whole_program_debug
Reference Guides	None
Support	Technical Support
Additional Resources	The SRIO IPC Transport benchmark example is available in $(TI_PDK_C6678_INSTALL_DIR)\packages\ti\transport\ipc\examples\srioIpcBenchmark The SRIO IPC Transport Chip to Chip example is available in $(TI_PDK_C6678_INSTALL_DIR)\packages\ti\transport\ipc\examples\srioIpcChipToChipExample
Downloads	[60]
License	BSD [42]

The MessageQ communication architecture utilizing the SRIO IPC transport is shown below.

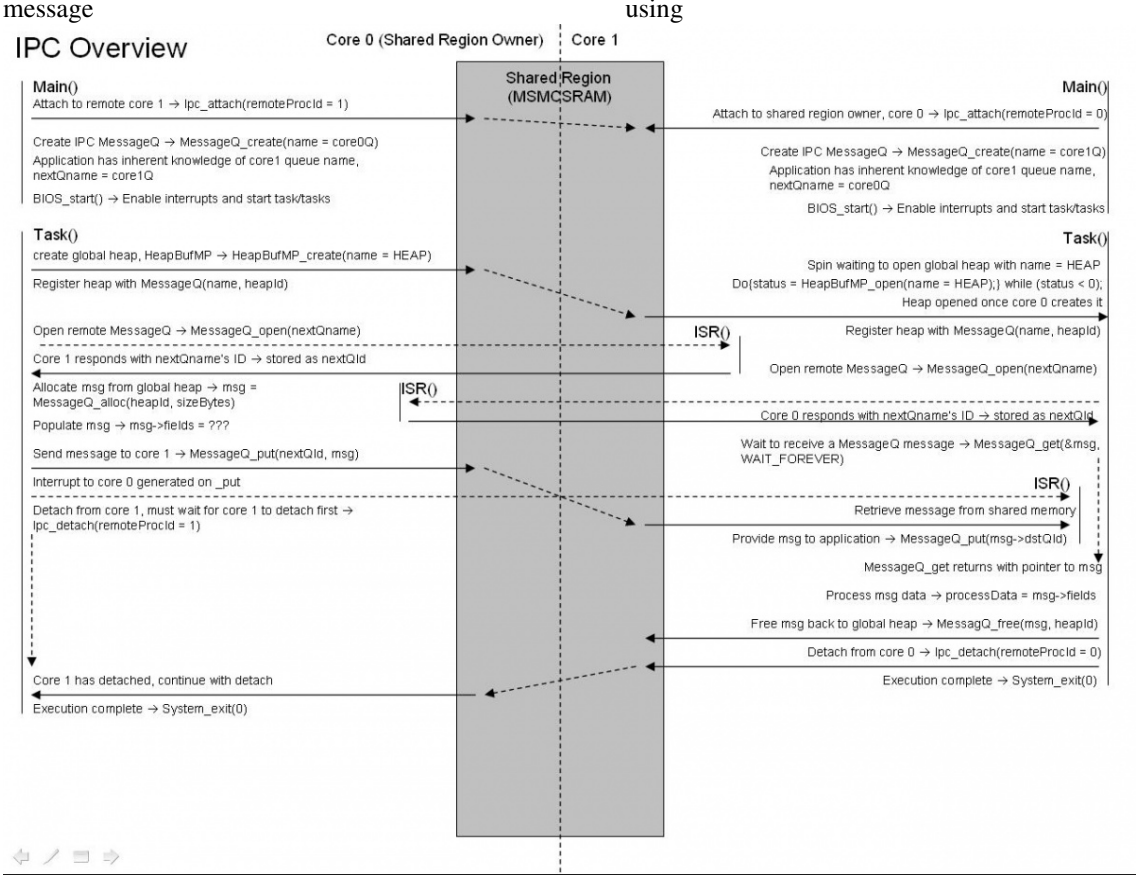


IPC Flow

This section provides ladders diagrams showing the execution flow that takes place when multiple cores access shared resources or exchange data using IPC. Not all function calls and input parameters are described in the ladder diagram. However, enough detail is provided to show how different cores share resources without stepping on one another.

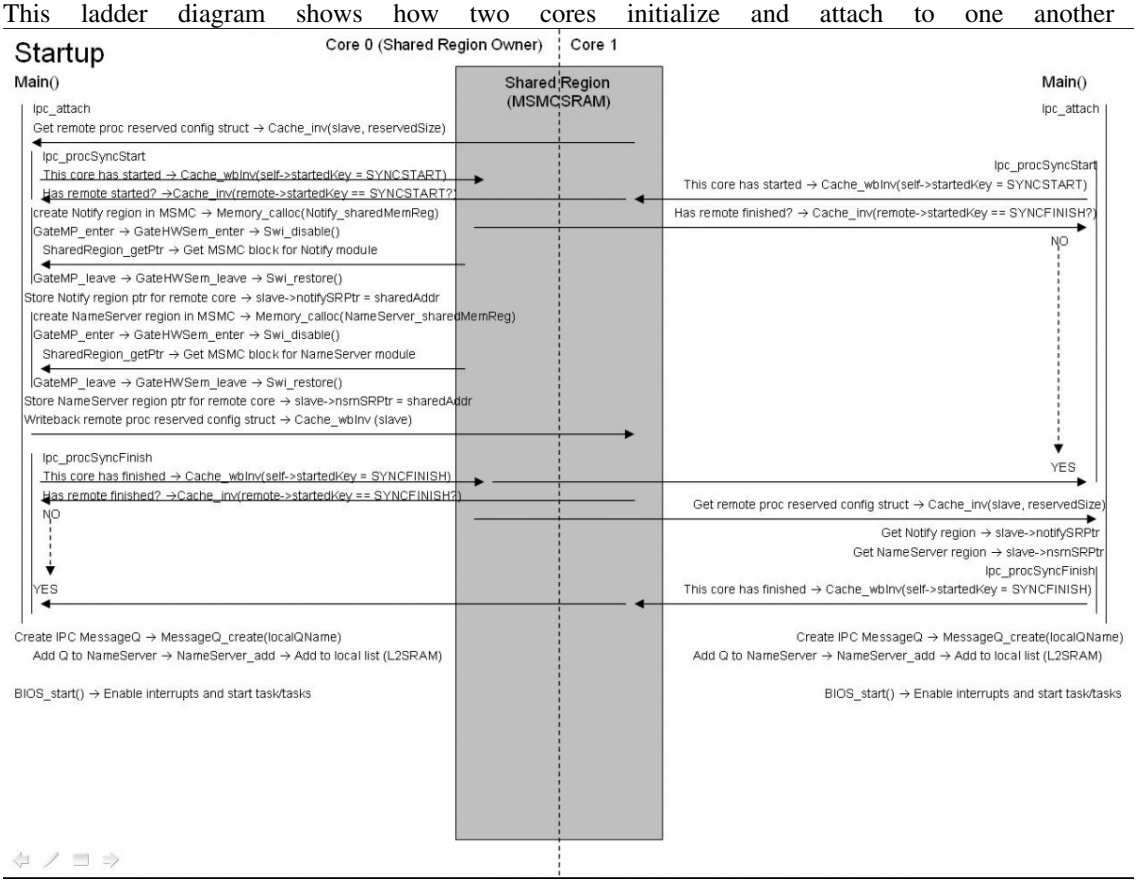
IPC Overview

A high-level ladder diagram showing how two cores would share a heap, MessageQ queues, and exchange a message using IPC



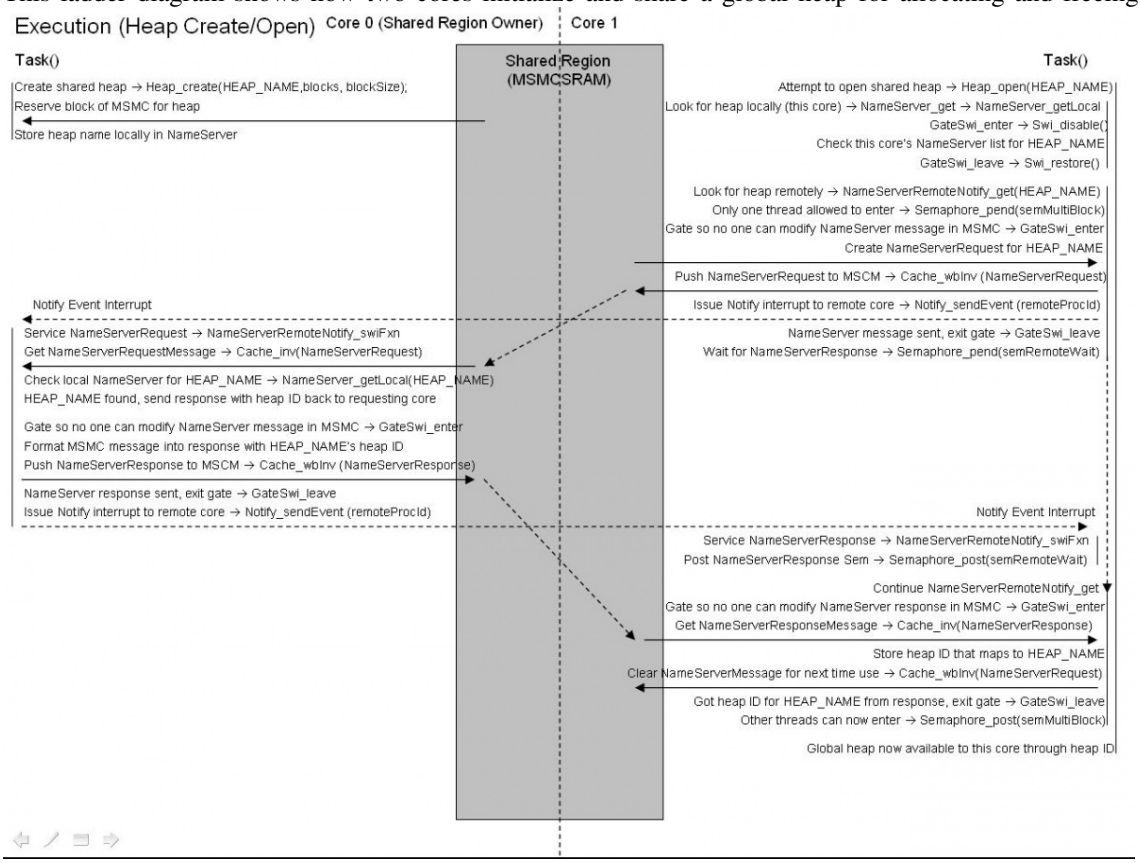
IPC Startup

This ladder diagram shows how two cores initialize and attach to one another via IPC:



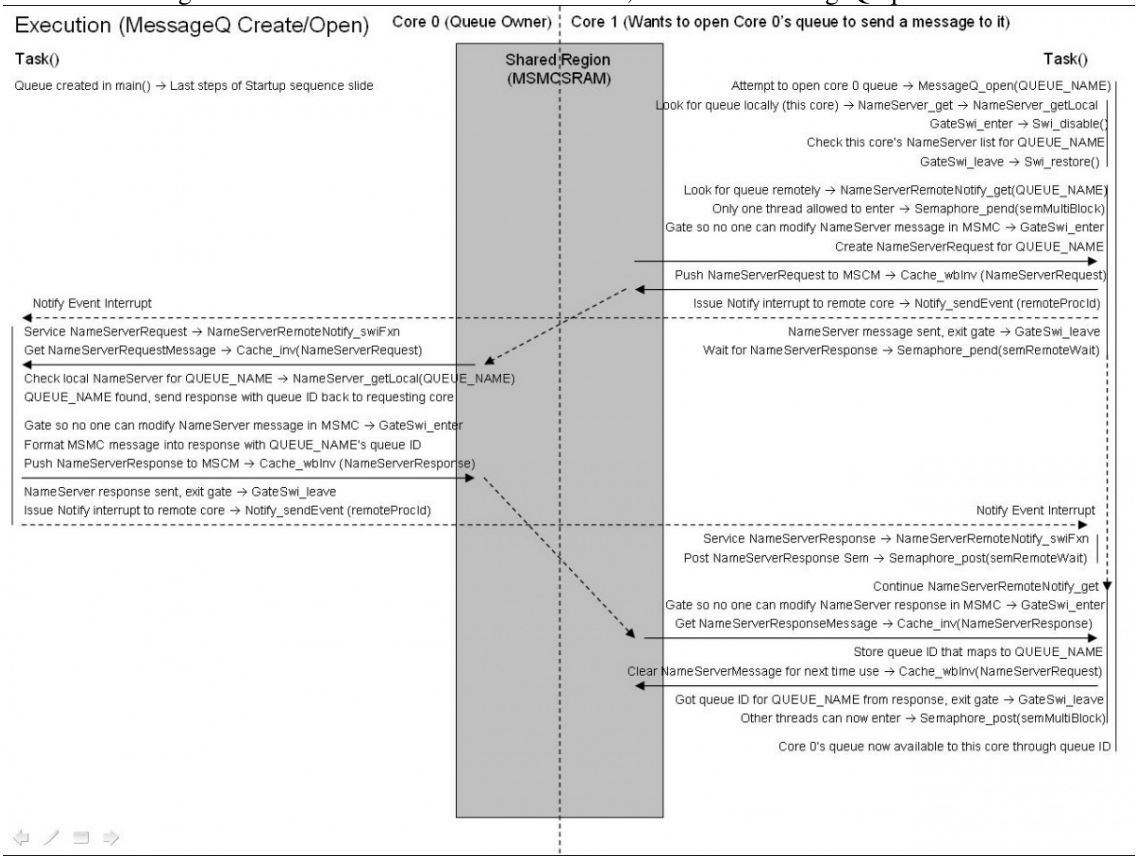
IPC Heap Sharing

This ladder diagram shows how two cores initialize and share a global heap for allocating and freeing messages:



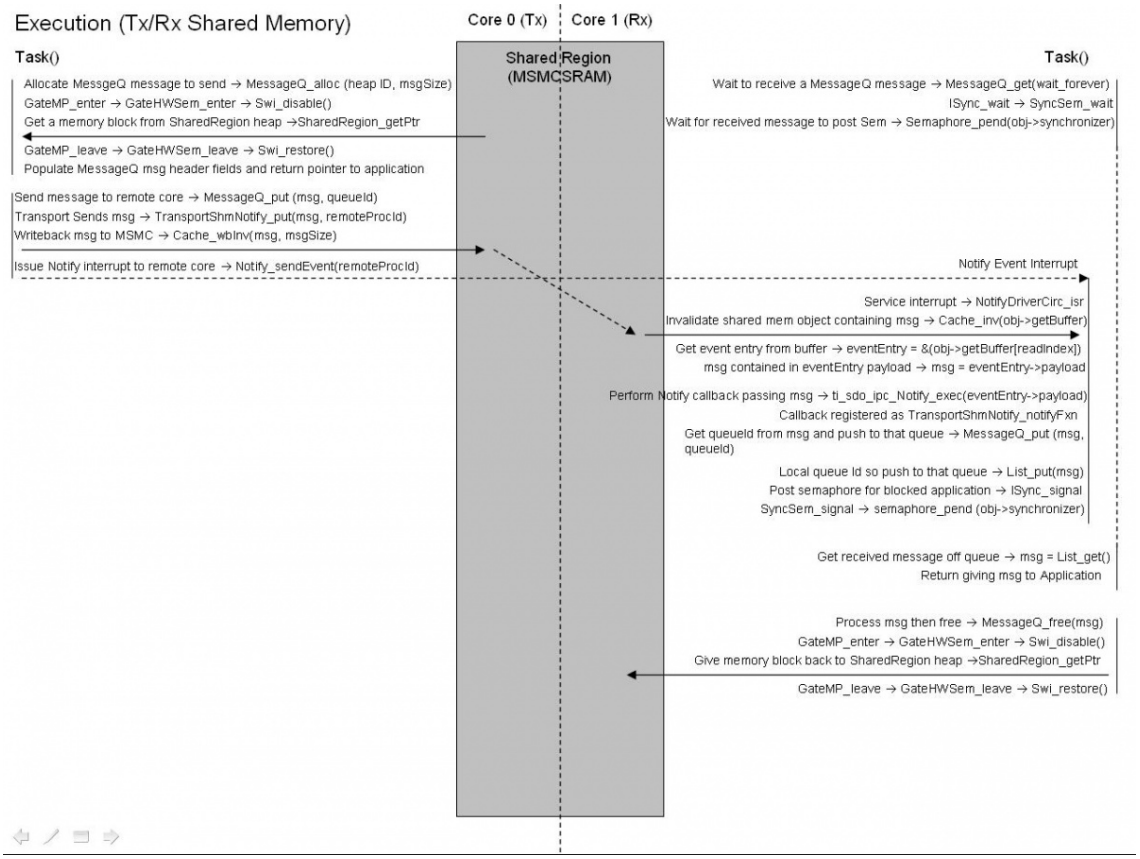
IPC MessageQ Queue Sharing

This ladder diagram shows how two cores search for, and find MessageQ queues located on remote cores:



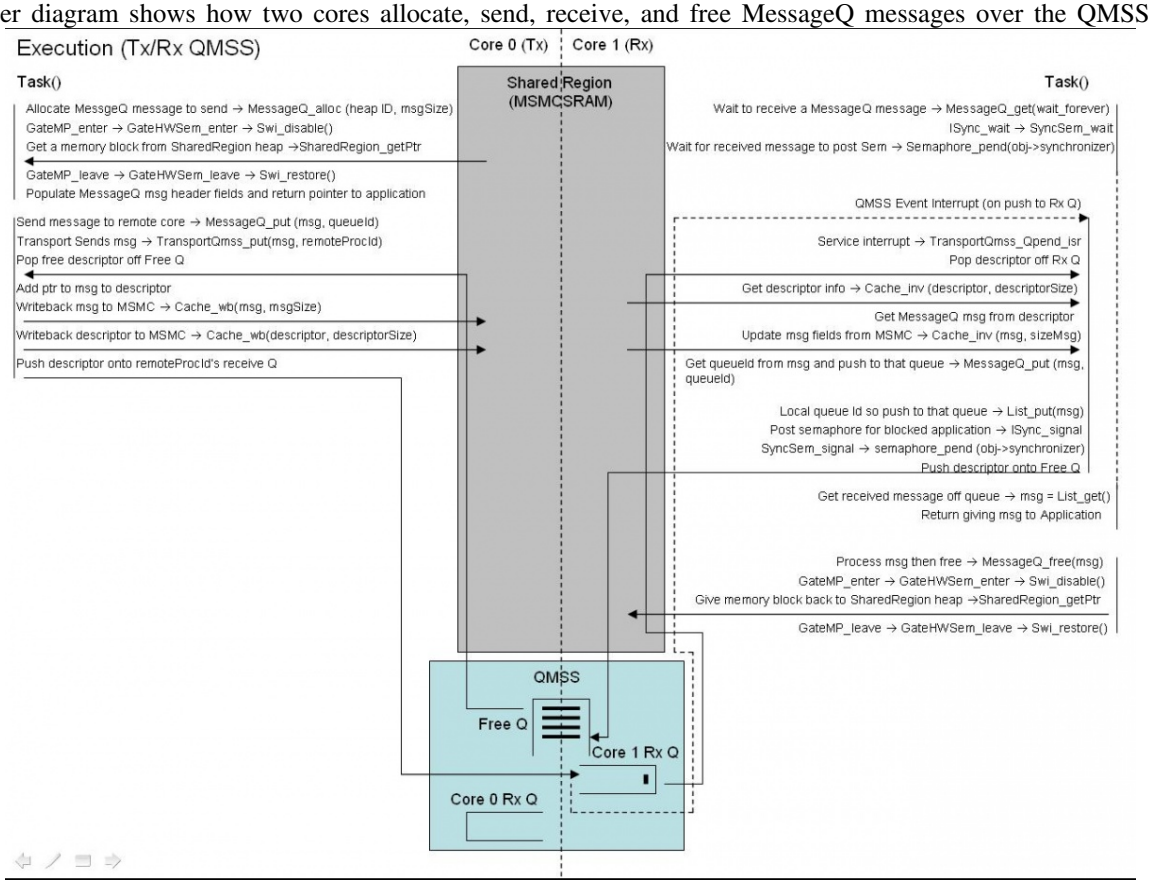
IPC Shared Memory Transport Message Passing

This ladder diagram shows how two cores allocate, send, receive, and free MessageQ messages over the Shared Memory transport:



IPC QMSS Transport Message Passing

This ladder diagram shows how two cores allocate, send, receive, and free MessageQ messages over the QMSS transport:



IPC Module Usage for Different Transports

When different IPC transports are used by an application some IPC modules may cease to function due to the system architecture. The system architecture dictates the IPC transport used. For example, chip to chip data transfer over MessageQ would be handled by the SRIO transport since SRIO established a transport path between two chips. This is something the Shared Memory and QMSS/Navigator transports are incapable of. The following describes which modules delivered in the IPC component are functional for each IPC transport.

Shared Memory IPC Transport

The Shared Memory transport is delivered with the IPC component package. The Shared Memory transport is the default IPC transport. As such, all modules delivered in IPC are functional and useable with the Shared Memory transport within the context of a single chip. The Shared Memory transport is delivered with IPC and used by default since it fits the generality module of IPC. It is the only transport that can be used when the architecture of the chip is not known.

Useable IPC Modules		
IPC Component	**Supported?**	**Comments**
IPC	**YES**	Required to start IPC regardless of transport
MessageQ	**YES**	Can use Shared Memory transport to send messages between threads on the same core and cores on the same chip
Heap*MP	**YES**	Messages allocated from shared memory on a source thread/core using a Heap*MP then sent over the Shared Memory transport can be freed on the destination thread/core
GateMP	**YES**	Can be used to synchronize threads/cores communicating over the Shared Memory transport
Notify	**YES**	Used to generate interrupt on destination core signalling there is a message available for it to receive on over the Shared Memory transport
SharedRegion	**YES**	Specifies the IPC Shared Region from which Heaps, MessageQ queues, and Shared Memory transport FIFOs should be allocated
MultiProc	**YES**	Specifies the cores within the system that the Shared Memory transport can transport messages between
NameServer	**YES**	Used to service MessageQ, Heap, and Gate _open requests between between cores which intend to communicate over the Shared Memory transport

QMSS/Navigator IPC Transport

The QMSS/Navigator transport is delivered with the PDK component packages. The QMSS/Navigator transport is a platform specific IPC transport that uses QMSS resources on the PDK platform. The QMSS/Navigator transport allows communication between threads on the same core and cores on the same chip. This is similar to the Shared Memory transport except the Navigator QMSS queues are used to move the message instead of shared memory. As such, all modules delivered in IPC are functional and useable with the QMSS/Navigator transport within the context of a single chip.

Useable IPC Modules		
IPC Component	**Supported?**	**Comments**
IPC	**YES**	Required to start IPC regardless of transport
MessageQ	**YES**	Can use QMSS/Navigator transport to send messages between threads on the same core and cores on the same chip
Heap*MP	**YES**	Messages allocated from shared memory on a source thread/core using a Heap*MP then sent over the QMSS/Navigator transport can be freed on the destination thread/core
GateMP	**YES**	Can be used to synchronize threads/cores communicating over the QMSS/Navigator transport
Notify	**YES but...**	Is not directly used by the QMSS/Navigator transport which generates an interrupt on the destination core via QMSS queue interrupt mechanisms. However, since the QMSS/Navigator transport works within the context of a single chip the Notify module can still be used to generate interrupts, out-of-band from the QMSS/Navigator transport, to different cores on the chip
SharedRegion	**YES**	Specifies the IPC Shared Region from which Heaps, and MessageQ queues should be allocated
MultiProc	**YES**	Specifies the cores within the system that the QMSS/Navigator transport can transport messages between
NameServer	**YES**	Used to service MessageQ, Heap, and Gate _open requests between between cores which intend to communicate over the QMSS/Navigator transport

SRIO IPC Transport

The SRIO transport is delivered with the PDK component packages. The SRIO transport is a platform specific IPC transport that uses SRIO and QMSS resources on the PDK platform. The SRIO transport allows communication between threads on the same core, cores on the same chip, and cores on different chips. When the SRIO transport is used to transport messages between entities within the same chip all IPC modules are useable, similar to the Shared Memory and QMSS/Navigator transports. However, when the SRIO transport is used to transport messages between entities on two separate chips only a subset of the IPC modules are useable. This is due to the assumption that there are no shared resources, such as hardware semaphores or shared memory, between two chips. The only thing connecting the chips are the SRIO lanes.

Useable IPC Modules When Communicating Between Cores on Different Chips		
IPC Component	**Supported?**	**Comments**
IPC	**YES**	Required to start IPC regardless of transport
MessageQ	**YES**	Can use SRIO transport to send messages between cores on different chips
Heap*MP	**NO**	Any heaps opened would only be useable for cores on the chip which the Heap*MP was opened. There is no sense of a Heap*MP instance that would be shared between cores on different chips. IPC assumes there is no shared memory between chips
GateMP	**NO**	Any gates used would only be synchronize cores on the chip which the Gate was opened. There is no sense of a Gate instance that would be shared between cores on different chips. IPC assumes there are no shared hardware semaphores between chips
Notify	**NO**	IPC assumes there is no hardware or software interrupt mechanism between cores on different chips
SharedRegion	**NO**	Any SharedRegion created would only be useable by cores on the chip which the SharedRegion was defined. There is no sense of a SharedRegion between cores on different chips. IPC assumes there is no shared memory between chips for the SharedRegion to exist
MultiProc	**YES**	Specifies the cores within the system, all chips, that the SRIO transport can transport messages between
NameServer	**YES**	Used to service MessageQ_open requests between between cores on different chips which intend to communicate over the SRIO transport. The SRIO transport itself is used to pass the NameServer request/response messages between the cores

IPC Benchmarks

IPC performance is measured in terms of the time (in cycles) to send a message from one core to another core and includes all cache coherency operations to ensure the message is ready for use by the receiving core. The one way latency is measured for shared memory, QMSS/Navigator, and SRIO transports.

Latency Benchmark Setup

To measure the 1-way latency a message is ping-ponged between two cores. Core 0 starts the test by sending a message to Core 1. Core 1 relays the message back to Core 0 who then sends it back to Core 1. The message ping-pongs between the two cores for a configured amount of iterations. Each time Core 0 receives the message it stores the round-trip time, in cycles, representing the total time for the message to go from Core 0 to Core 1 then back to Core 0. This measured time is divided by two to get the one-way latency. The one-way latency measurements are then averaged over all iterations to yield the average 1-way latency.

The QMSS/Navigator transport results are presented for both QPEND and Accumulator options. See IPC Transports for more information on the QPEND and Accumulator implementations.

For the SRIO transport a four 1x port and 3.125 Gbps link rate was used. Loopback mode was disabled so all packets were transferred over the SRIO lanes and not looped back in the SRIO hardware.

Benchmark Results

	Shared Memory Transport	**QMSS Transport (QPEND)**	**QMSS Transport (Accumulator - 1 Descriptor per Interrupt)**	**QMSS Transport (Accumulator - 10 Descriptors per Interrupt)**	**SRIO Transport (Type 11 - 1 packet per Interrupt)**	**SRIO Transport (Type 11 - 10 packets per Interrupt)**
Avg 1-way Latency (Cycles)	2,402	1,673	4,522	4,606	9,056	9,104

Notes:

- -o3 compiler option
- All debug and assert options disabled

Benchmark Comments:

- The Shared Memory transport is the default IPC transport offering good out-of-the box performance.
- Applications which require the very best in latency performance should use the QPEND implementation of the QMSS/Navigator transport. These queues, when pushed descriptors, interrupt the DSP directly through the INTC module. The QMSS/Navigator transport is delivered as part of PDK. For information on how to configure the QMSS/Navigator transport to use QPEND queues please see Using and Configuring the Navigator/QMSS Transport.
- The QMSS/Navigator transport should be configured to use the Accumulator implementation if interrupt pacing is desired. The Accumulator configuration has a higher latency than its transport counterpart but offers the ability to interrupt the DSP after a number of descriptors have been pushed to an accumulator queue or after a certain amount of time has passed. For information on how to configure the QMSS/Navigator transport to use Accumulator queues, as well as configure the pacing and timeout values, please see Using and Configuring the Navigator/QMSS Transport.
- The SRIO transport, despite a high latency, offers the ability to transfer messages between cores on different chips. This is something that is not possible with the Shared Memory or QMSS/Navigator transports. The SRIO transport is delivered as part of PDK. For information on how to configure the SRIO transport please see Using

and Configuring the SRIO Transport.

The benchmark applications used to find the latency measurements are included in PDK under $(TI_PDK_C667x_INSTALL_DIR)\packages\ti\transport\ipc\examples. READMEs describing how to build and run the benchmarks are contained within the individual benchmark directories. See Explicit Programming Module Using IPC for guidance on modifying transport configuration options when rerunning the benchmark applications.

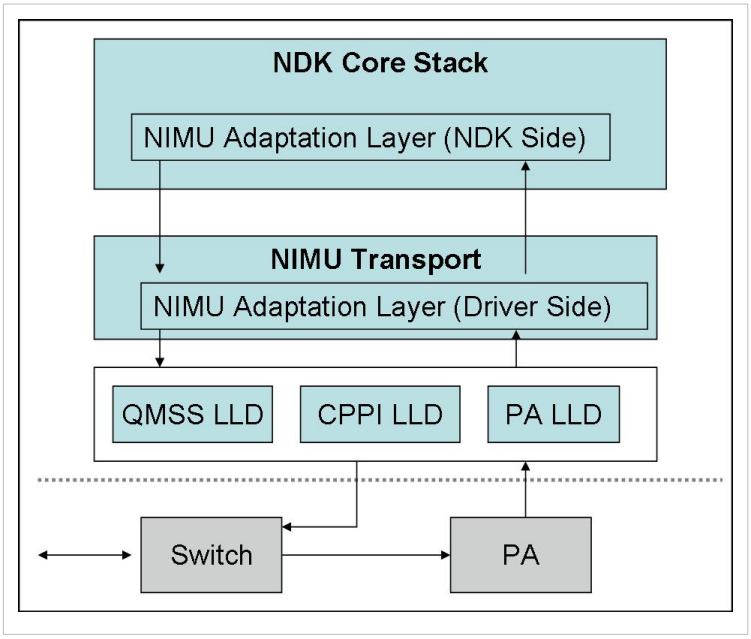
Network Development Kit (NDK)

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 architecure for NDK is shown below.



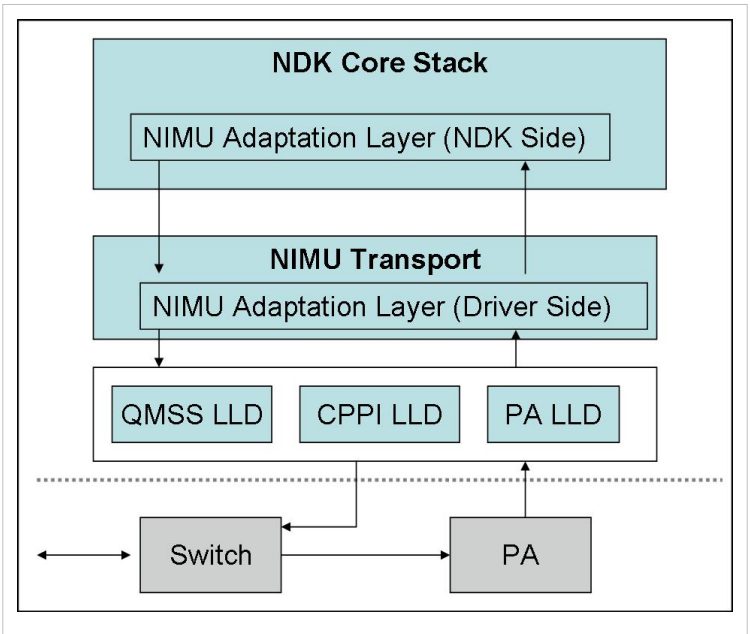
Network Development Kit Summary	
Component Type	Libraries
Install Package	NDK
Install Directory	ndk_<version>\
Project Type	Eclipse RTSC [40]
Endian Support	Little and Big
Library Name	binsrc.lib or binsrce.lib and cgi.lib or cgie.lib and console.lib or consolee.lib and hdlc.lib or hdlce.lib and miniPrintf.lib or miniPrintfe.lib and netctrl.lib or netctrle.lib and nettool.lib or nettoole.lib and os.lib or ose.lib and servers.lib or serverse.lib and stack.lib or stacke.lib
Linker Path	$(NDK_INSTALL_DIR)\packages\ti\ndk\lib\<arch>
Linker Sections	.far:NDK_OBJMEM, .far:NDK_PACKETMEM
Section Preference	L2 Cache
Include Paths	NDK_INSTALL_DIR is set automatically by CCS based on the version of NDK you have checked to build with. ${NDK_INSTALL_DIR}\packages\ti\ndk\inc ${NDK_INSTALL_DIR}\packages\ti\ndk\inc\tools
Reference Guides	See docs under Install Directory
Support	Technical Support
Additional Resources	The NDK unit test examples are available in $(TI_MCSDK_INSTALL_DIR)\packages\ti\platform\nimu\test\evm####
Extended Support	Eclipse RTSC Home [40] NDK User's Guide [61] NDK Programmer's Reference Guide [62] NDK Support Package Ethernet Driver Design Guide [63] NDK_FAQ [64] Rebuilding NDK Core [65]
Downloads	NDK Downloads [66]
License	BSD [42]

Network Interface Management Unit (NIMU) Driver

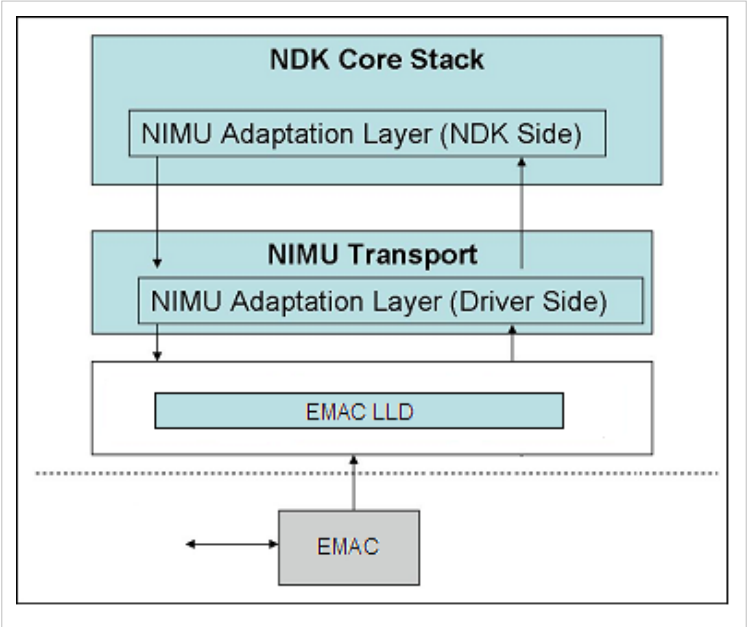
NIMU sits between NDK common software and the C6678 SoC and provides a common interface for NDK communication. This package contains NDK unit test examples for all supported platforms.

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 the C6678 platform as an example) is shown below. A similar architecture is also applicable for the C6670 platform.



Note: The below model is applicable for C6657 platform.



NIMU Summary	
Component Type	Library
Install Package	PDK_C6678_INSTALL_DIR
Install Directory	mcsdk_<version>\packages\ti\transport\ndk\nimu
Project Type	Eclipse RTSC [40]
Endian Support	Little
Library Name	ti.transport.ndk.nimu.ae66 (little)
Linker Path	$(TI_PDK_C6678_INSTALL_DIR)\packages\ti\transport\ndk\nimu\lib\debug for debug version $(TI_PDK_C6678_INSTALL_DIR)\packages\ti\transport\ndk\nimu\lib\release for release version
Linker Sections	nimu_eth_ll2
Section Preference	L2SRAM
Include Paths	$(TI_PDK_C6678_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_MCSDK_INSTALL_DIR)\examples\ndk\evm####
Downloads	[60]
License	BSD [42]

OpenMP Run-Time Library (OMP)

OMP is an implementation of an openMP run-time library for SYS/BIOS supporting KeyStone multicore DSP devices. The library implements support for thread management, shared memory, and synchronization as required for openMP.

Combined with the TI compiler (version 7.4 or greater) a user can create OpenMP programs for TI's multicore DSPs.

OMP Library Summary	
Component Type	Library
Install Package	OMP
Install Directory	omp_<version>\
Project Type	Eclipse RTSC
Endian Support	Little
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	$(OMP_INSTALL_DIR)\packages
Reference Guides	See docs under Install Directory
Support	Technical Support

Additional Resources	PDK
Downloads	BIOS MCSDK
License	BSD [42] and GPL-3.0-with-GCC-exception [67]

Algorithm Libraries

TI provides several algorithm libraries, each specific to a particular arena. Each library provides a collection of C-callable low-level functions (kernels), each tailored for optimal performance on a specific TI processing device (or devices). The libraries are typically used in computationally intensive real-time applications where execution speed is a critical factor. Their use generally accelerates execution speeds well beyond that achieved by equivalent code written in standard ANSI C. Additionally, use of these libraries can significantly reduce application development time. Source code is provided in all cases to facilitate kernel modification when needed.

See c6x Software Library mediawiki [38] for a comprehensive overview of the various software libraries available for TI's c6x family of processors.

DSP Library (DSPLIB)

DSPLIB is an optimized DSP Function Library and includes many C-callable, optimized, general-purpose signal-processing routines including:

- Adaptive Filtering
- Correlation
- Fast Fourier Transform
- Filtering and convolution
- Matrix

DSPLIB Summary	
Component Type	Library
Install Package	DSPLIB
Install Directory	dsplib_c66x_<version>\
Project Type	CCS [18]
Endian Support	Big and Little
Library Name	dsplib.a66 (COFF, little-endian) dsplib.a66e (COFF, big-endian) dsplib.ae66 (ELF, little-endian) dsplib.ae66e (ELF, big-endian)
Linker Path	<root_install_dir>\lib\
Linker Sections	N/A
Section Preference	N/A
Include Paths	<root_install_dir>\inc\ <root_install_dir>\packages\
Reference Guides	See docs under Install Directory
Support	BIOS E2e Forum [68]
Additional Resources	c6x Software Library mediawiki [38]

Downloads	DSPLIB Downloads [69]
License	BSD [42]

Image Processing Library (IMGLIB)

IMGLIB is an optimized image/video processing library with kernels in the following functional categories:

- Compression & Decompression
- Image Analysis
- Image Filtering and Conversion

IMGLIB Summary	
Component Type	Library
Install Package	IMGLIB
Install Directory	imglib_c66x_<version>\
Project Type	CCS [18]
Endian Support	Little
Library Name	imglib.ae66 (ELF, little-endian)
Linker Path	<root_install_dir>\lib\
Linker Sections	N/A
Section Preference	N/A
Include Paths	<root_install_dir>\inc\ <root_install_dir>\packages\
Reference Guides	See docs under Install Directory
Support	BIOS E2e Forum [68]
Additional Resources	c6x Software Library mediawiki [38]
Downloads	IMGLIB Downloads [70]
License	BSD [42]

Floating Point Math Library (MATHLIB)

MATHLIB contains optimized versions of most commonly used floating point math routines contained in the RTS library. Kernels are offered in two variations:

- Double-precision floating point
- Single-precision floating point

MATHLIB Summary	
Component Type	Library
Install Package	MATHLIB
Install Directory	mathlib_c66x_<version>\
Project Type	CCS [18]
Endian Support	Big and Little
Library Name	mathlib.a66 (COFF, little-endian) mathlib.a66e (COFF, big-endian) mathlib.ae66 (ELF, little-endian) mathlib.ae66e (ELF, big-endian)
Linker Path	<root_install_dir>\lib\
Linker Sections	N/A
Section Preference	N/A
Include Paths	<root_install_dir>\inc\ <root_install_dir>\packages\
Reference Guides	See docs under Install Directory
Support	BIOS E2e Forum [68]
Additional Resources	c6x Software Library mediawiki [38]
Downloads	MATHLIB Downloads [71]
License	BSD [42]

Demonstration Software

The MCSDK consist of demonstration software to illustrate device and software capabilities, benchmarks, and usage.

High-Performance DSP Utility Application (HUA)

HUA is the MCSDK out-of-box demonstration/utility application which includes a web server and has pages to query information about the platform and software versions, network statistics, network throughput benchmark, board diagnostics, flash read and write, and EEPROM read and write functions. This is a basic utility application which demonstrates basic platform functionality and how to integrate some of the basic software infrastructure (e.g., SYS/BIOS, NDK, Platform Library). The Utility is accessed from a web browser by browsing the platforms IP address (which can be assigned either as a static IP or through DHCP.) Pages available in the utility are Information, Statistics, Benchmarks, Flash, Diagnostics, and EEPROM.

See the HUA Demonstration Guide for more information.

Image Processing Demonstration

The Image Processing Demonstration illustrates the integration of key components in the MCSDK. The purpose of the demonstration is to provide a multicore software development framework on an evaluation module (EVM).

1. Demonstrates the transfer of image data from/to DDR and internal memory. Typically, images are large and need to be stored in external memory.
2. Operates on different segments of the same image in different DSP cores.
3. Operates across multiple cores executing different algorithms on the same image data.
4. Transfers input/output image to external systems (e.g., a PC).

See the Image Processing Demo Guide for more information.

Multicore Video Infrastructure Demonstration

The multicore video infrastructure demonstration includes a set of demonstration applications targeted to demonstrate the use of MCSDK for real-time multicore video processing applications. The applications include Ethernet packet-to-packet processing of video streams (transcoding, encoding, decoding) for a various common video standards, resolutions, and use cases. There are two demonstrations included:

1. Multichannel high-density operation with low resolution
2. Multicore processing of high resolution video codecs

See the MCSDK Video Demonstration Guide for more information.

Note: The multicore video infrastructure demo is not provided as part of the MCSDK, but is provided as a separate package available here [39].

Bootloader and Boot Utilities

The platform package includes POST (Power On Self Test), bootloader software and utilities to write images to the EEPROM, NOR and NAND Flash.

Boot Utilities

Boot Utilities include a set of tools to configure and boot the board. These include:

- **Intermediate Boot Loader (IBL):** Resides on EEPROM that supports customizing configuration for boot modes. See IBL user guide [72] for details.
- Examples of booting/loading images for NAND, NOR, and Ethernet
- Write utilities for NAND, NOR, and EEPROM

The boot utilities are discussed further in the section on Booting and Flash.

Multicore Application Deployment (MAD) Utilities

The Multicore Application Deployment (MAD) is a collection of tools allows you to create a bootable image that can support multiple images and multiple cores. The premise behind MAD is to allow you to:

- Deploy multiple applications on multiple cores.
- Conserve memory by sharing common code.
- Deploy an application dynamically on a core, if needed.

See MAD Utils User Guide [73] for more details.

An example of an MCSDK application that uses MAD is the Image Processing Demo Guide.

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 of the MCSA include the following:

- Real-time event monitoring
- Multicore event correlation
- Correlation of software events, hardware events and CPU trace
- Real-time profiling and benchmarking
- Real-time debugging

The MCSA includes two key components:

- 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/uia_<version>, ccsv5/eclipse, ccsv5/ccs_base_5.0.0.*/dvt\
Project Type	Eclipse RTSC [40]
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 [74]
Downloads	Installed as a part of BIOS MCSDK installation
UIA License	BSD [42]
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 [40]
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 [40] Users Guide and Reference Manual [75]
Downloads	N/A
License	See XDC Manifest in the install directory

Third Party Software and Tools

Prism from Criticalblue

Prism is a multicore analysis tool provided by Critical Blue [76] as an Eclipse plug-in which is intended to allow developers to evaluate parallelization strategies of existing sequential code upfront without implementing any code changes. This is accomplished by the developer taking their existing serial code and running it through a standard simulator (in this case our TI C66x simulator). The results from the simulator are fed into Prism, which displays the execution profile and, more importantly, uses the simulation data to allow the developers to perform what-if analysis without changing a line of code. This includes looking at data parallelization strategies and task parallelization strategies. The obvious motivation is to allow a developer to investigate various parallelization strategies without having to go through the entire process of implementation and debug which can be time consuming and requires significant effort. This enables a kind of ROI assessment for multicore prior to investment in implementation. In the end, this represents at least a good start for someone beginning to look at migration of existing applications to a multicore device.

Please see http://www.criticalblue.com/prism/ti/ [77] for more detailed information on Prism and to get started.

Please see Image Processing Demo Analysis with Prism [78] for notes on running Prism with the image processing demo application.

Poly-Platform from PolyCore Software

Poly-Platform by PolyCore Software [79], is a development framework, consisting of tools and runtime software, providing a programming model for the application to scale from one to many cores in homogenous and heterogeneous multicore environments. The tools are Eclipse plug-ins and are integrated with CCSv5 for a seamless development environment and, provide rapid development for MCAPI programming and topology configuration. Poly-Messenger, the runtime engine which is integrated with DSP BIOS, transparently handles the communications between cores and between processors across multiple transports. Applications readily move from one core to multicore to many cores using the same source code base.

Please see http://www.polycoresoftware.com/products.php [80] for more information.

Build and Example Guide

The Build and Example Guide talks about setting up your build environment for 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 complete the following:

- Install Code Composer Studio
- Install the MCSDK software
- Create a Target Configuration File that allows communication with the EVM over JTAG

The Getting Started Guide [81] talks about how to do this.

Once CCS and MCSDK are installed, they provide both Debug and Release versions of the demonstrations, examples and components. In addition, many of the components provide pre-built big endian versions as well. To rebuild 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.

Building the Software

Build in Place vs. Build in Workspace

The MCSDK uses a "Build in Place" philosophy. This means projects should not be import 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

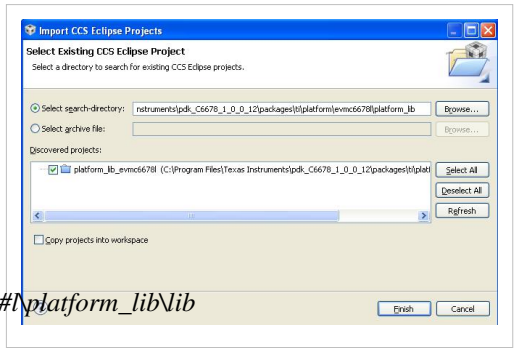
- If you want to modify and rebuild a library, it is best not to copy it into your workspace. We suggest building it "in place". When you build in place, you do not need to change build macros and so forth. You also not have to edit the example projects as they already have the correct paths to the library.
- 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 it is a change you need to add, then you can rebuild 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.

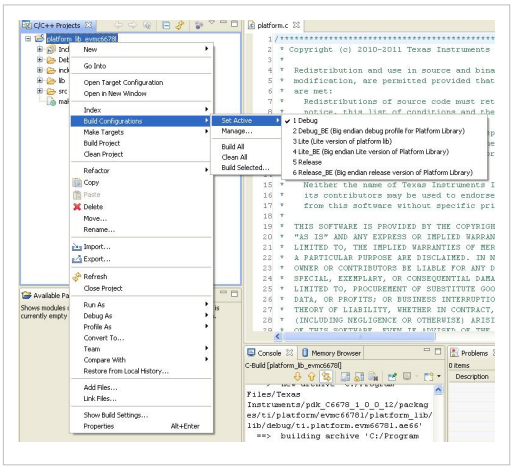
Platform Library

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 MCSDK is installed in *C:\Program Files\Texas Instruments*.

- Open CCS (preferably with a new workspace)
- Goto *Project->Import Existing CCS/CCE Eclipse project*
- In the *Select search-directory:* enter *C:\Program Files\Texas Instruments\pdk_C667##_#_#_#_##\packages\ti\platform\evmc667#l\platform_lib* and hit *Browse*. See Import Project Settings. This will import platform_lib_evmc667## into the workspace.
- Make sure the *Copy projects into workspace* is not checked. Then hit *Finish*.
- Import the platform library project under interest to CCS. For example, for building C6678 platform library import the project *platform_lib_evmc6678l* into the CCS.
- Now *Project->Rebuild All* should rebuild the project and library is created in *C:\Program Files\Texas Instruments\pdk_C66##_#_#_#_##\packages\ti\platform\evmc667#l\platform_lib\lib* for a selected profile. Setting Profile for Project Settings. This will set the desired profile for platform_lib_evmc667## into the workspace.





Profile	Little endian Library name	Big Endian Library Name	Comment
Debug	/lib/debug/ti.platform.evm6678l.ae66	/lib/debug/ti.platform.evm6678l.ae66e	Full Symbol Debug Platform library
Release	/lib/release/ti.platform.evm6678l.ae66	/lib/release/ti.platform.evm6678l.ae66e	Optimized Full Platform library
Lite	/lib/debug/ti.platform.evm6678l.lite.lib	lib/debug/ti.platform.evm6678l.lite.libe	Platform library intended only for Power On Self Test (POST) executable

See *platform_library_user_guide* located under *C:\Program Files\Texas Instruments\pdk_C667#_#_#_#_##\packages\ti\platform\docs\platform* for more information on platform APIs.

Note: The library name provided above is provided as an example for the C6678 platform. Similar naming conventions for the library can be applied for the C6657 and C6670 platforms.

Building CSL and the Low Level Device Drivers

Follow the instructions below to build CSL and LLDs.

- Open a command window inside of the $(TI_PDK_C66##_INSTALL_DIR)\packages directory.
- Set the environment by running the batch file and follow the instructions as per the batch file output.

```
.\ti\drv\pdksetupenv.bat
```

- After configuring the environment successfully, the following message appears.

```
... ... ... 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. Follow the steps below to build the CCS projects for the example projects.

- Check Prerequisites

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. The steps are as follows:

- 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.

The 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_C####_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 the test.

Compiling Big Endian MCSDK Demos and Examples

The pre-compiled platform libraries, NIMU drivers, NDK examples, and HUA demos 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.

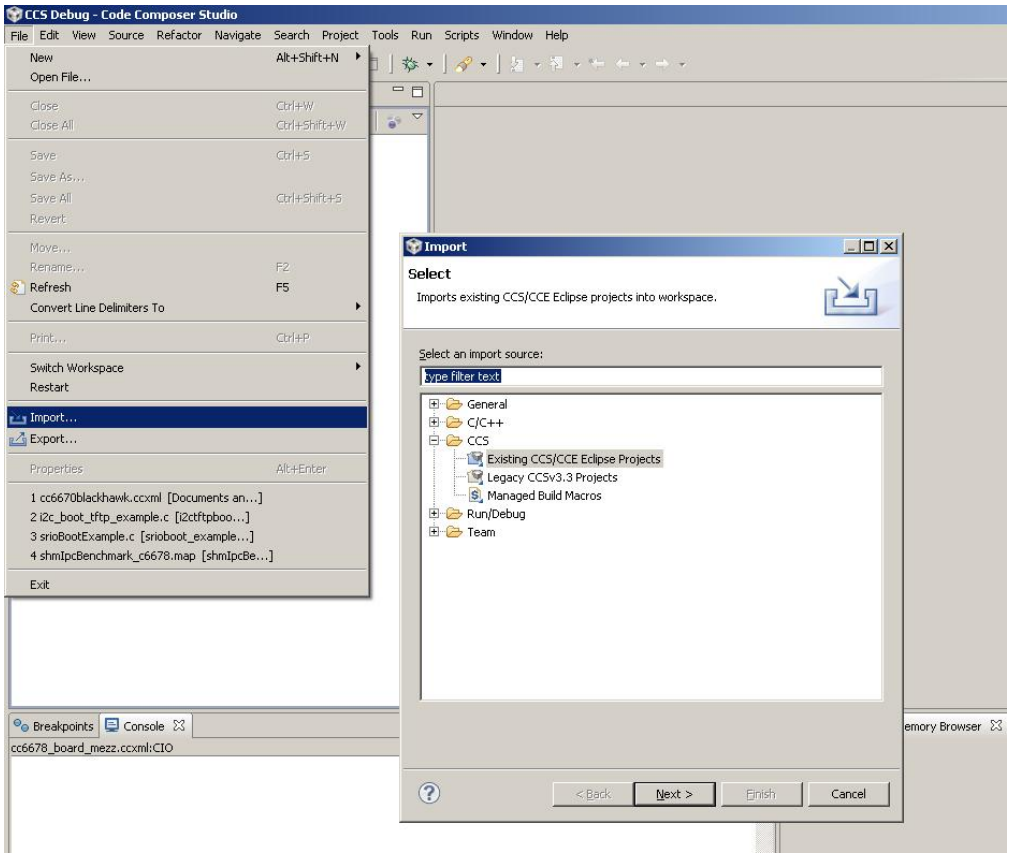
Note: The following images describing the steps to build the Big Endian libraries portray c6678l projects. The same instructions can be used for c6670l projects.

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\evmc66xxl\gel\evmc66xxl.gel.

Recompile Big Endian NDK NIMU Driver

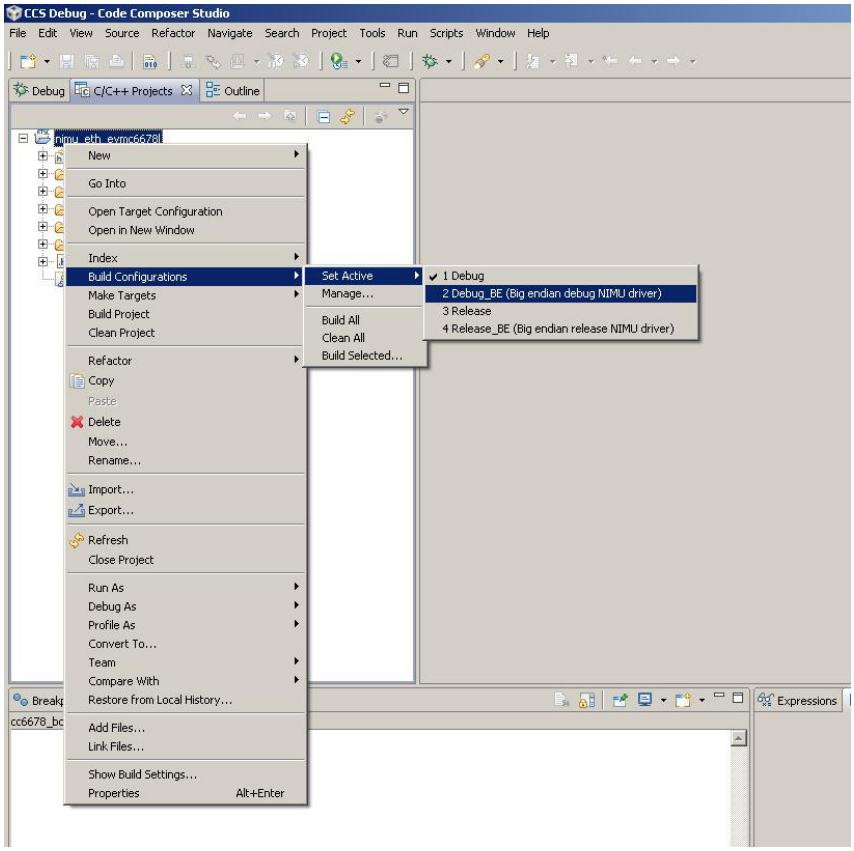
- The NIMU driver is required for all NDK examples and the HUA demo. This must be recompiled in Big Endian prior to recompiling any example or demo in Big Endian.

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_evmc66xxl 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_evmc66xxl 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 and the HUA demo.

Recompile Big Endian Platform Library

5. **Import the Platform library project:** Repeat steps 1. and 2. from above to import the platform_lib_evmc66xxl project. This project should be located within the PDK installation directory, under ti\platform\evmc66xxl\platform_lib.

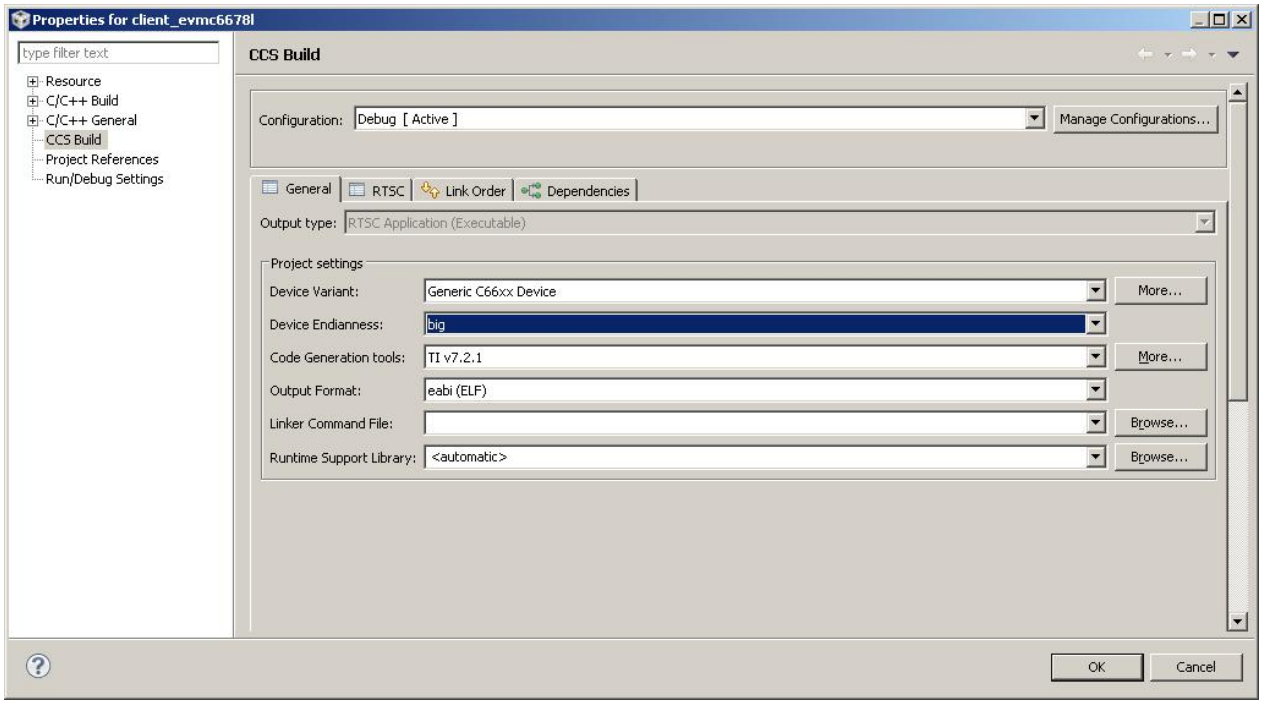
6. **Change the Platform project active build configuration to Big Endian (Debug or Release):** Repeat step 3. from above to set the big endian build configuration.

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_evmc66xxl project. This project should be located within the MCSDK installation directory, under examples\ndk\client\evmc66xxl.

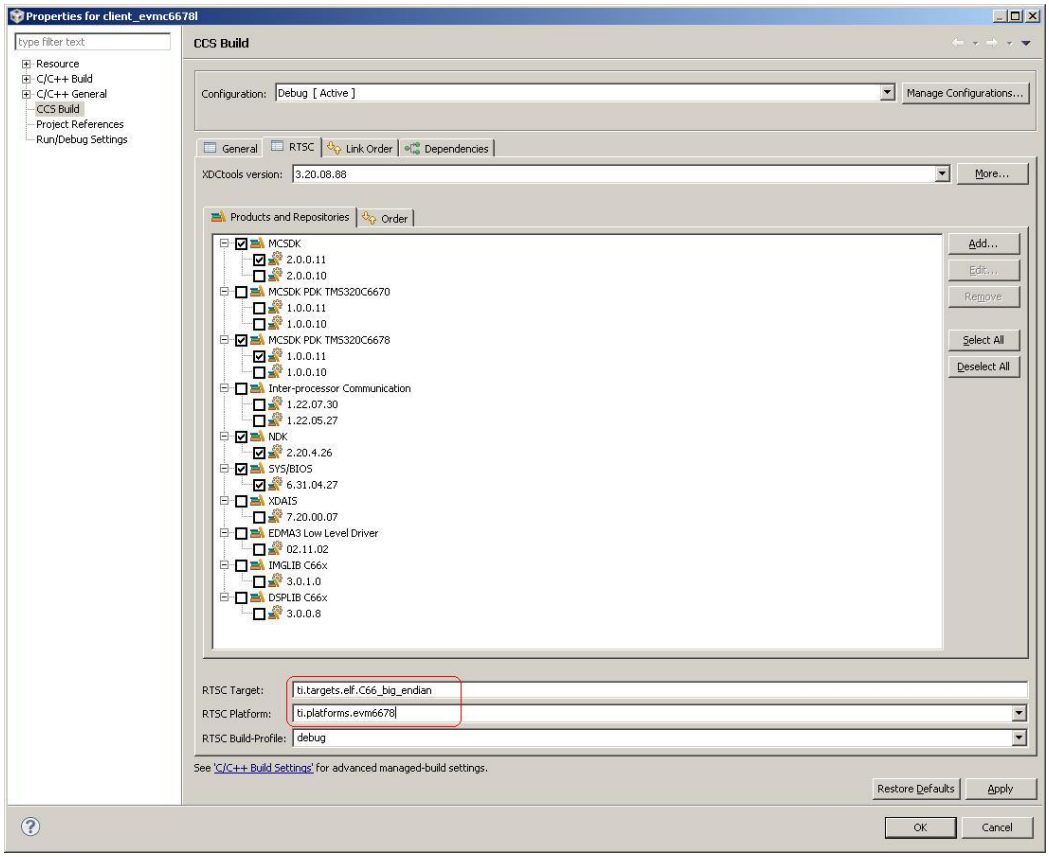
9. **Reconfigure the Client example for Big Endian:** With the client_evmc66xxl project selected, click on Project -> Properties and then select the "CCS Build" pane. In the "General" tab set "Device Endianness" to "big". Click "Apply".



In addition, click on the "RTSC" tab and configure the following and click "Apply" when finished:

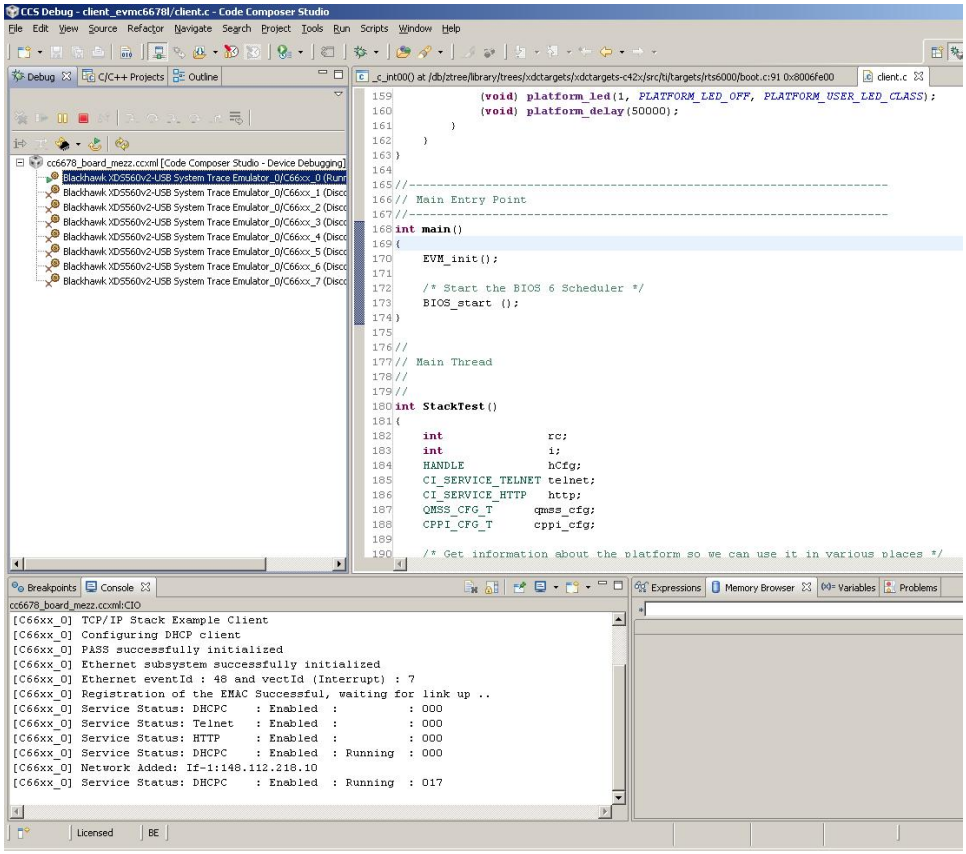
RTSC Target: ti.targets.elf.C66_big_endian

RTSC Platform: ti.platforms.evm66xx



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 and HUA Demo

11. **Reconfigure NDK HelloWorld Example and HUA Demo as Big Endian and rebuild:** Follow step 8. through 10. to rebuild the NDK HelloWorld Example and HUA Demo 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_5.x.x.xxxxxx\simulation_csp_ny\bin\configurations with name tisim_c####_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_sgmii0_tx_data_gen_opin,

System.C66XX_S.SHARED_SYSTEM.SWITCHSS.switchss_sgmii1_rx_data_gen_ipin;
CONNECT12
System.C66XX_S.SHARED_SYSTEM.SWITCHSS.switchss_sgmii0_rx_data_gen_ipin,

System.C66XX_S.SHARED_SYSTEM.SWITCHSS.switchss_sgmii1_tx_data_gen_opin;
```

as follows:

```
//CONNECT11
System.C66XX_S.SHARED_SYSTEM.SWITCHSS.switchss_sgmii0_tx_data_gen_opin,

System.C66XX_S.SHARED_SYSTEM.SWITCHSS.switchss_sgmii1_rx_data_gen_ipin;
//CONNECT12
System.C66XX_S.SHARED_SYSTEM.SWITCHSS.switchss_sgmii0_rx_data_gen_ipin,

System.C66XX_S.SHARED_SYSTEM.SWITCHSS.switchss_sgmii1_tx_data_gen_opin;
```

This disables loopback at EMAC adapter level (PHY simulation) in the simulator.

- Finally, configure the switch MAC configured in the example, i.e., 0x10-0x11-0x12-0x13-0x14-0x15 on the EMAC adaptor so that the simulator can pass all packets matching the switch MAC up to the application.

example:

```
 INPUT5  MAC_ADDRESS_PORT0, 10-11-12-13-14-15;   // configure the Port0 MAC to be the switch MAC
INPUT6  MAC_ADDRESS_PORT1, 00-01-02-03-04-05;
```

Note: Note: For details see C:\Program Files\Texas Instruments\ccsv5\ccs_base_5.x.x.xxxxxx\simulation_csp_ny\docs\pdf\TCI6616-C6670-TCI6608-C6678_Device_Simulator_EMAC

Re-compile NIMU library with simulator support

- Start CCS and import project from C:\Program Files\Texas Instruments\pdk_C66##_#_#_#_##\packages\ti\transport\ndk\nimu directory
- Open Project->Properties->C/C++ Build->Settings->Predefined symbols, add variable *SIMULATOR_SUPPORT*, OK to close the project
- Re-compile the project Project->Clean, Project->Compile

Update NDK client example and run it on simulator

Note: The PC running simulator needs to be set with static IP address *192.168.2.101* for this example program, see figure for Static IP Setup

- Import project from C:\Program Files\Texas Instruments\mcsdk_#_##_##_##\examples\ndk\client\evmC####
- Open client.cfg file in CCS text editor from the project client_evm####l and change the line

from `var PlatformLib = xdc.loadPackage('ti.platform.evmc####l');` to `var PlatformLib = xdc.loadPackage('ti.platform.simc####');`

- Open file client.c, then change clientMACAddress string to match your PC mac address, make sure the format needs to be as follows

```
Uint8 clientMACAddress [6] = {0x00,0x18,0x8B,0x10,0x17,0xBF};
```

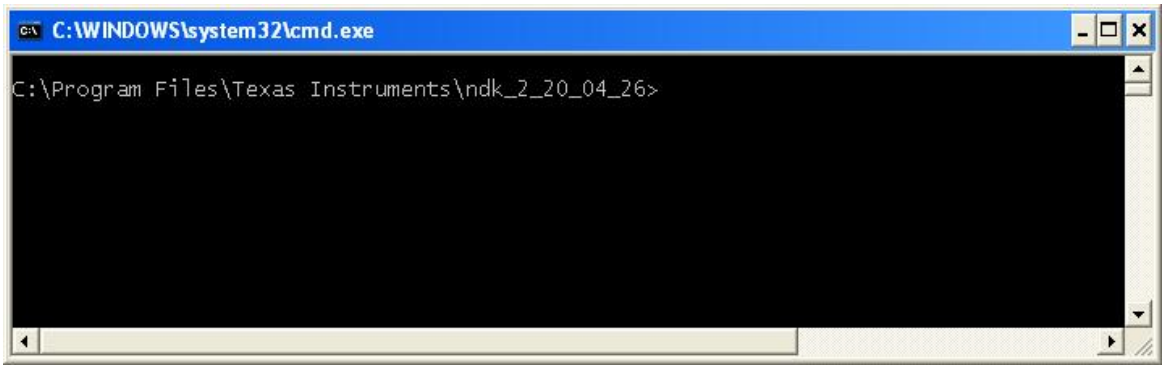
- Re-compile the project Project->Clean, Project->Build
- Load functional simulator target on CCS
- Load the client image created above on the simulator and hit run to run the application

Building NDK

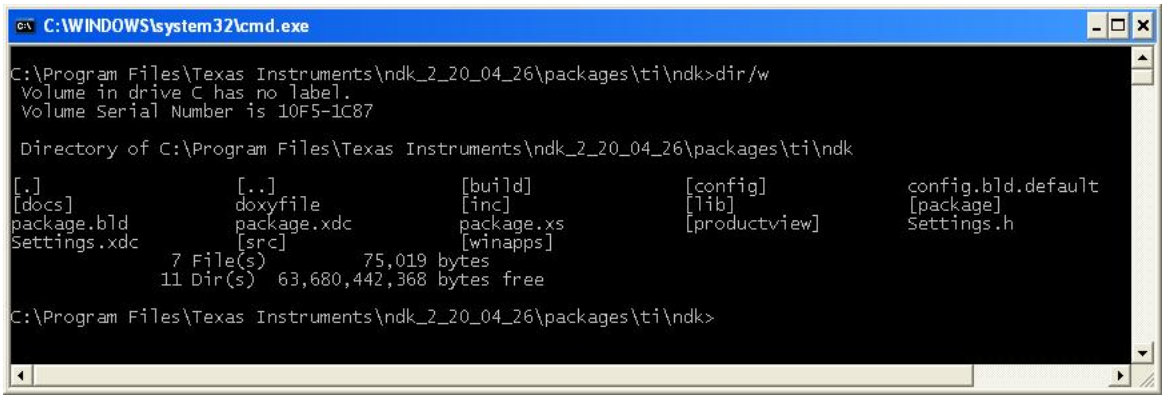
The following instructions how how to re-build the NDK libraries and enable debug versions if you need them.

Note: The NDK build re-builds everything in the library and its quite large so re-building may take some time on slower machines.

- Before you start building its a good idea to make a backup copy of the library.
- Open a Windows cmd window (dos box) in your NDK install directory. You can do this by selecting the NDK top directory and then right clicking and selecting run cmd here (in windows XP).



- Change directory to 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 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"

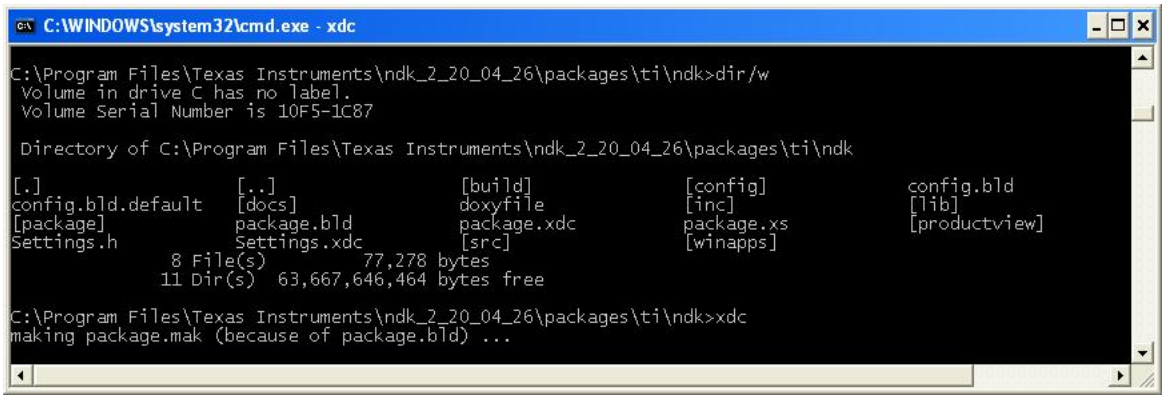
Remove tragets you do not need built. You should see our C66 targets. The others for ARM or C64 can removed if you do not need to build for them. Build.targets = [

```
    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.



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 MCSDK are introduced. It is highly recommended that you do them.

Note: The following examples assume you installed 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. As shown below, the #### refers to processor type (6678 for TMS320C6678 OR TMS320TCI6608; 6670 for TMS320C6670 OR TMS320TCI6618; 6657 for TMS320C6657) and the xx refers to a version number.

For example, a typical path might be:

```
"C:\Program Files\Texas Instruments\pdk_C####_1_0_0_xx\packages"
```

To specify that for the 6670 on the 2.0.0.11 release you would do:

```
"C:\Program Files\Texas Instruments\pdk_C6670_1_0_0_11\packages"
```

Example 1 - Building and running a simple single core application

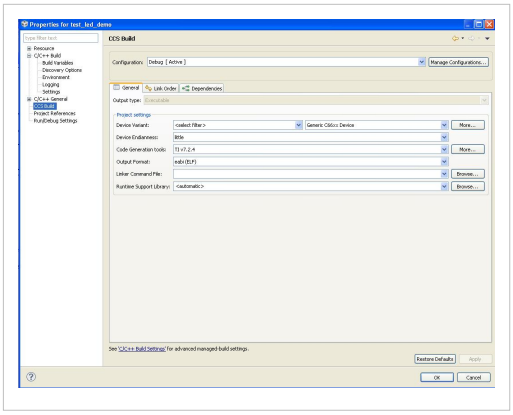
This is the first example program. It's purpose is to get you used to creating projects in CCS, building an executable and then running it on your EVM. The application executes out of shared memory on the EVM and does not use the external DDR.

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 Templets* 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 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
    .sysmem      >      MSMCSRAM
    .far         >      MSMCSRAM
    .testMem     >      MSMCSRAM
    .fardata     >      MSMCSRAM
    platform_lib >       MSMCSRAM
```

}

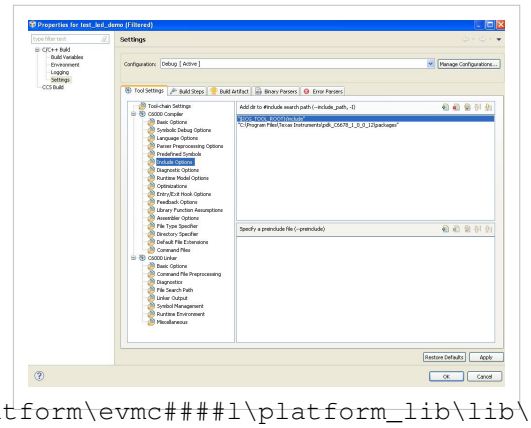
4. Were almost done. We have some code to execute and a memory map. Now we need to build the executable we will load and run. Before we build though, we will need to define a few include paths and specify the library for the Platform Library.

- 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_C####_1_0_0_xx\packages"
```

See Include Path

- Select *File Search Path* from *C6000 Linker* section. Add following items in *Include library...* section



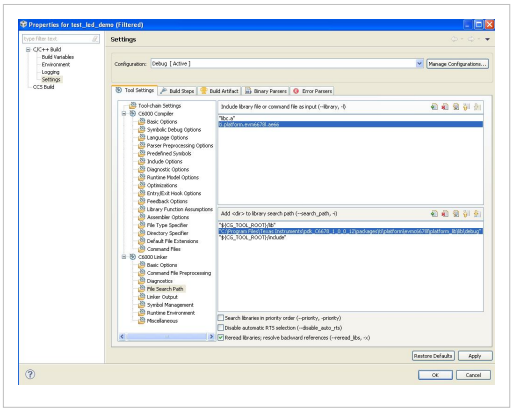
`ti.platform.evm####l.ae66` **Note:** Please note that the above library is the little endian debug version library of the platform library. This is needed for the application built for Little Endian. Please refer to the above table for including the appropriate library for the particular platform library application.

And add following items in *Add <dir> to library...* section

```
"C:\Program Files\Texas Instruments\pdk_C####_1_0_0_xx\packages\ti\platform\evmc####l\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 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. In the *RTSC Target* field

enter *ti.targets.elf.C66*. Before you're done with this screen you need to select the *RTSC Platform* you are using. Select the *ti.platforms.evm66##* from the list box (note it will be empty, but just click on it and values will be filled in to select from).

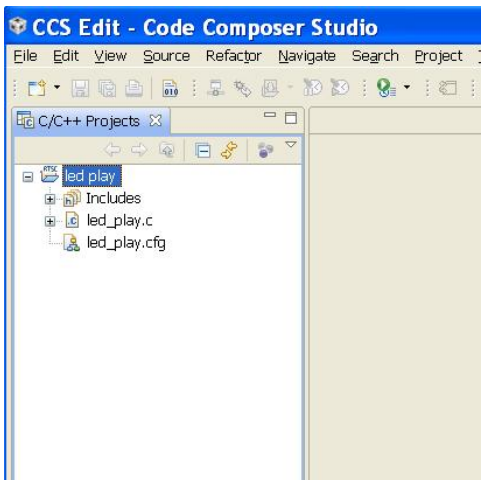
- Hit *Finish*

Note: The eclipse plugin discovery tool registers the project templates from the individual components with CCSv5. After the discovery tool registers XDCtools 3.22.01 version provided with BIOS MCSDK 2.0.1 release, the option *Empty RTSC Project* does not appear in the *Project Templates* screen because XDCtools 3.22.01 does not have the *Empty RTSC Project* template. Please follow this link to work around this problem.

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.



LedRtscProject.JPG

3. Lets add the code we need to the *led_play.c* file:

1. include <cerrno>
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 *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;
```

}

```
/*********************************************************************
* 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.ddr = 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.firstFxns.$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";

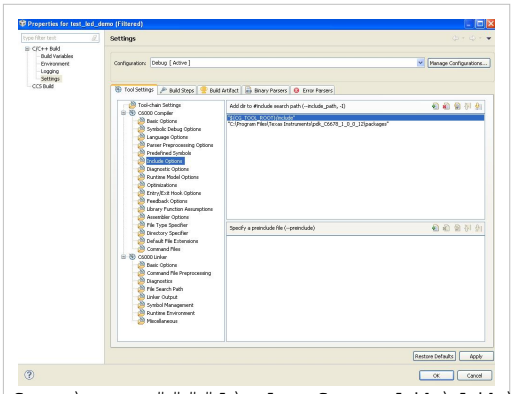
5. Now we need to configure a few project settings for the Platform Library (just like we did in the previous example).

- 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_C####_1_0_0_xx\packages"`

See Include Path

- Select *File Search Path* from *C6000 Linker* section. Add following items in *Include library...* section

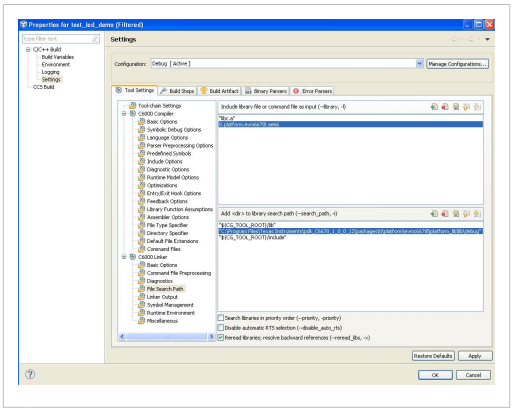


`ti.platform.evm####l.ae66` **Note:** Please note that the above library is the little endian debug version library of the platform library. This is needed for the application built for Little Endian. Please refer to the above table for including the appropriate library for the particular platform library application.

And add following items in *Add <dir> to library...* section `"C:\Program Files\Texas Instruments\pdk_C####_1_0_0_xx\packages\ti\platform\evmc####l\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 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.

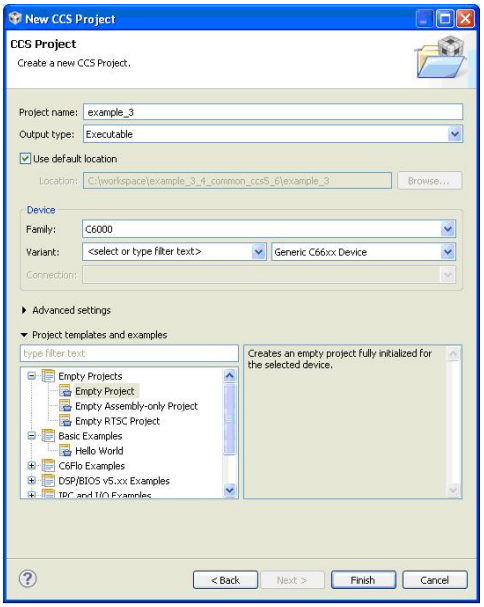
6. Build the project.

7. Connect to your EVM with your Target Configuration file, then load and run the program!

Example 3 - Running from external memory (DDR)

This example essentially re-does the second example and takes the LED example code and puts it into DDR3 external memory. This example is created using CCS version 5.1.1. Please note that steps used to create the LED example using CCS version 5.0 and version 5.1 are very similar.

1. The first step is to create an Eclipse RTSC project as follows:



- Open CCS (preferably with a new workspace).
- Open File->New->CCS Project and in the project name field enter led_play_ddr3.
- Select device family as C6000
- Leave Device Variant as “select or type filter text” and select Generic C66xx Device on the next drop down list.
- In the Project Templates screen (see image to the right), select Empty Project then hit Finish

2. The second step is to create RTSC configuration file as follows:

- Right click on led_play_ddr3 project->New->Other->RTSC->RTSC configuration File, then hit Next
- Enter RTSC configuration file name as led_play_ddr3.cfg and hit Finish.

3. Now we have an Eclipse RTSC project and its configuration file. Our next step is to overwrite .cfg file and the source file with test code and configuration that 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.) It allows you to configure default settings within them.

4. Lets add the code we need to the led_play_ddr3 main.c file.

/*

=main.c

- /

1. include <xdc/std.h>
2. include <xdc/runtime/Error.h>
3. include <xdc/runtime/System.h>
4. include <ti/sysbios/BIOS.h>
5. include <ti/sysbios/knl/Task.h>
6. include <cerrno>
7. include <stdio.h>
8. include <stdlib.h>
9. include <string.h>
10. include <ti/sysbios/BIOS.h>
11. include <ti/sysbios/hal/Hwi.h>
12. include <ti/bios/include/swi.h>
13. include "ti\platform\platform.h"
14. 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;
```

}

/**

```
* 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.ddr = 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;
```

}

/*

=taskFxn

- /

Void taskFxn(UArg a0, UArg a1) { 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; }

}

/*

=main

- /

Void main() {

```
   Task_Handle task;
   Error_Block eb;
```

```
   System_printf("enter main()\n");
```

```
   Error_init(&eb);
   task = Task_create(taskFxn, NULL, &eb);
   if (task == NULL) {
       System_printf("Task_create() failed!\n");
       BIOS_exit(0);
   }
```

```
BIOS_start();     /* enable interrupts and start SYS/BIOS */
```

}

4. Add the code to the cfg file led_play_ddr3.cfg by opening .cfg file with XDCscript editor. Right click on configuration file->open with->XDCscript editor. Copy and paste the following code to .cfg file.

/*

```
* led_play_ddr3.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 Task = xdc.useModule('ti.sysbios.knl.Task'); var BIOS = xdc.useModule('ti.sysbios.BIOS');

/* Configure the Modules */ BIOS.taskEnabled = true;

/* Create our memory map - i.e. this is equivalent to linker.cmd */ Program.sectMap[".const"] = "DDR3"; Program.sectMap[".text"] = "DDR3"; Program.sectMap[".code"] = "DDR3"; Program.sectMap[".data"] = "DDR3"; Program.sectMap[".sysmem"] = "DDR3"; Program.sectMap["platform_lib"] = "DDR3";

/* 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.firstFxns.$add('&EVM_init');

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_ddr3 project, select Build->C6000 linker->File Search Path from the left pane.
- On File Search Path window, add library file name ti.platform.evm6678l.ae66 and add dir to library file search path "c:\Program Files\Texas Instruments\ pdk_C####_1_0_0_xx

\packages\ti\platform\evmc6678l\platform_lib\lib\debug"

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. You may be wondering why we do not need include/library paths or library names for BIOS? Any RTSC enabled component in the 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. Now select appropriate RSTC components by right click on project name->properties->Resource->General->RTSC (select PDK and BIOS versions and etc.), and then select appropriate target platform.

7. Build the project.

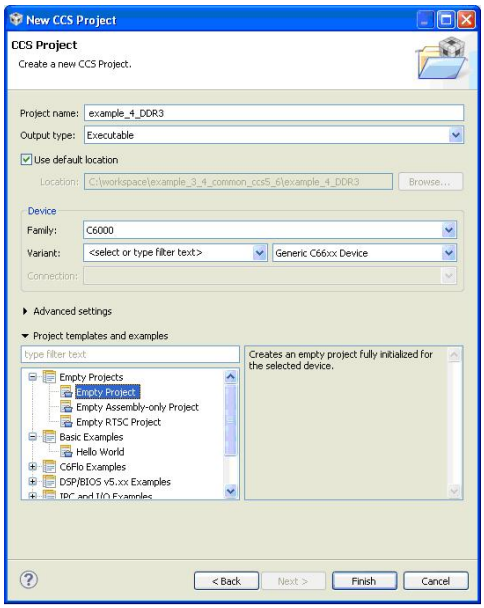
8. Connect to your EVM with your Target Configuration file, then load and run the program. You should now see all LEDs blinking.

Example 4 - Let's make it multi-core

This example enhances the LED example code to run on multicore and puts it into DDR3 external memory. Similar to example 3, this example uses CCS version 5.1.1 to create its project. Please note that steps used to create this LED example with CCS version 5.0 and version 5.1 are very similar.

1. The first step is to create an Eclipse RTSC project as follows:

- Open CCS (preferably with a new workspace).
- Open File->New->CCS Project and in the project name field enter led_play_ddr3.
- Select device family as C6000
- Leave Device Variant as “select or type filter text” and select Generic C66xx Device on the next drop down list.
- In the Project Templates screen, select Empty Project then hit Finish



2. The second step is to create RTSC configuration file as follows:

- Right click on led_play_ddr3 project->New->Other->RTSC->RTSC configuration File, then hit Next
- Enter RTSC configuration file name as led_play_ddr3.cfg and hit Finish.

3. Now we have an Eclipse RTSC project and its configuration file. Our next step is to overwrite .cfg file and the source file with test code and configuration that 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 ponents you wish to use and 3.) It allows you to configure default settings within them.

4. Lets add the code we need to the led_play_ddr3.c file:

/*

```
* led_play.c
*
```

```
*  Created on: Feb 6, 2012
*
*/
```

1. include <cerrno>
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"

1. pragma DATA_SECTION(next, ".sharedVar")
2. pragma DATA_ALIGN (next, 128)

typedef union { uint32_t core; uint8_t padding[128]; }n;

n next;

uint32_t maxFlashes = 50;

/* 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;
}
/*********************************************************************
*
* Function: Converts a core local L2 address to a global L2 address
*   Input addr:  L2 address to be converted to global.
*   return:  uint32_t   Global L2 address
*
*********************************************************************/
uint32_t convert_CoreLocal2GlobalAddr (uint32_t addr) {
 uint32_t coreNum;
 /* Get the core number. */
 coreNum = CSL_chipReadReg(CSL_CHIP_DNUM);
/* Compute the global address. */
 return ((1 << 28) | (coreNum << 24) | (addr & 0x00ffffff));
}
/*********************************************************************
* 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.ddr = 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
* each core (0-3) sequentially twinkles its LEDs. This task is configured to start in led_play.cfg
* configuration file and it is called from BIOS.
*
**************************************************************************/
```

void ledPlayTask (void) {

```
platform_info p_info;
uint32_t led_no = 0;
uint32_t coreId, i;
```

```
/* determine the core number. */
coreId = CSL_chipReadReg (CSL_CHIP_DNUM);
```

```
/* Get information about the platform */
platform_get_info(&p_info);
```

```
/* determine which core to twinkle LED      */
if(coreId != 0){
      while(1){
            /* lets delay a bit before reading shared variable       */
            platform_delay(30000);
            CACHE_invL1d (&next, 4, CACHE_FENCE_WAIT);
            if(next.core == coreId)
                  break;
      }
}
```

```
/* lets delay a bit before twinkling the next LED      */
platform_delay(30000);
i = 0;
led_no = coreId;
platform_write("core = %d starts twinkling its LED\n", coreId);
```

```
/* twinkle the LED based on core id and LED id, respectively      */
while(1) {
    platform_led(led_no, PLATFORM_LED_ON, PLATFORM_USER_LED_CLASS);
```

```
    platform_delay(300000);
    platform_led(led_no, PLATFORM_LED_OFF, PLATFORM_USER_LED_CLASS);
```

```
    platform_delay(300000);
```

```
    i++;
    if ( i == maxFlashes){
          break;
    }
}
```

```
/* let next core twinkles its LED      */
next.core = coreId + 1;
CACHE_wbL1d ((void *) &next, 4, CACHE_WAIT);
platform_write("core %d is done.\n", coreId);
```

}

5. Add the code to the cfg file led_play_ddr3.cfg by opening it with XDCscript editor by right click on configuration file->open with->XDCscript editor

```
var Startup = xdc.useModule('xdc.runtime.Startup');
```

var Defaults = xdc.useModule('xdc.runtime.Defaults'); var Diags = xdc.useModule('xdc.runtime.Diags'); var Error = xdc.useModule('xdc.runtime.Error'); var Log = xdc.useModule('xdc.runtime.Log'); var LoggerBuf = xdc.useModule('xdc.runtime.LoggerBuf'); var Main = xdc.useModule('xdc.runtime.Main'); var Memory = xdc.useModule('xdc.runtime.Memory') var SysMin = xdc.useModule('xdc.runtime.SysMin'); var System = xdc.useModule('xdc.runtime.System'); var Text = xdc.useModule('xdc.runtime.Text');

var Csl = xdc.loadPackage('ti.csl');

var BIOS = xdc.useModule('ti.sysbios.BIOS'); var Clock = xdc.useModule('ti.sysbios.knl.Clock'); var Swi = xdc.useModule('ti.sysbios.knl.Swi'); var Task = xdc.useModule('ti.sysbios.knl.Task'); var Semaphore = xdc.useModule('ti.sysbios.knl.Semaphore'); var Hwi = xdc.useModule('ti.sysbios.hal.Hwi');

/*

```
* Program.argSize sets the size of the .args section.
* The examples don't use command line args so argSize is set to 0.
*/
```

Program.argSize = 0x0;

/*

```
* Uncomment this line to globally disable Asserts.
* All modules inherit the default from the 'Defaults' module.  You
* can override these defaults on a per-module basis using Module.common$.
* Disabling Asserts will save code space and improve runtime performance.
```

Defaults.common$.diags_ASSERT = Diags.ALWAYS_OFF;

```
*/
```

/*

```
* Uncomment this line to keep module names from being loaded on the target.
* The module name strings are placed in the .const section. Setting this
* parameter to false will save space in the .const section.  Error and
* Assert messages will contain an "unknown module" prefix instead
* of the actual module name.
```

Defaults.common$.namedModule = false;

```
*/
```

/*

```
* Minimize exit handler array in System.  The System module includes
* an array of functions that are registered with System_atexit() to be
* called by System_exit().
*/
```

System.maxAtexitHandlers = 4;

/*

```
* Uncomment this line to disable the Error print function.
* We lose error information when this is disabled since the errors are
* not printed.  Disabling the raiseHook will save some code space if
* your app is not using System_printf() since the Error_print() function
```

```
* calls System_printf().
```

Error.raiseHook = null;

```
*/
```

/*

```
* Uncomment this line to keep Error, Assert, and Log strings from being
* loaded on the target.  These strings are placed in the .const section.
* Setting this parameter to false will save space in the .const section.
* Error, Assert and Log message will print raw ids and args instead of
* a formatted message.
```

Text.isLoaded = false;

```
*/
```

/*

```
* Uncomment this line to disable the output of characters by SysMin
* when the program exits.  SysMin writes characters to a circular buffer.
* This buffer can be viewed using the SysMin Output view in ROV.
```

SysMin.flushAtExit = false;

```
*/
```

/*

```
* The BIOS module will create the default heap for the system.
* Specify the size of this default heap.
*/
```

BIOS.heapSize = 0x1000;

/* System stack size (used by ISRs and Swis) */ Program.stack = 0x2000;

/* Circular buffer size for System_printf() */ SysMin.bufSize = 0x200;

/*

```
* Create and install logger for the whole system
*/
```

var loggerBufParams = new LoggerBuf.Params(); loggerBufParams.numEntries = 16; var logger0 = LoggerBuf.create(loggerBufParams); Defaults.common$.logger = logger0; Main.common$.diags_INFO = Diags.ALWAYS_ON;

System.SupportProxy = SysMin;

/* Example 3 Create our memory map - i.e. this is equivalent to linker.cmd */ Program.sectMap[".const"] = "DDR3"; Program.sectMap[".text"] = "DDR3"; Program.sectMap[".code"] = "DDR3"; Program.sectMap[".data"] = "DDR3"; Program.sectMap[".sysmem"] = "DDR3"; Program.sectMap[".sharedVar"] = "DDR3"; Program.sectMap["platform_lib"] = "DDR3";

/* 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.firstFxns.$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";

6. 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_ddr3 project, select Build->C6000 linker->File Search Path from the left pane.
- On File Search Path window, add library file name ti.platform.evm6678l.ae66 and add dir to library file search path "c:\Program Files\Texas Instruments\ pdk_C####_1_0_0_xx \packages\ti\platform\evmc6678l\platform_lib\lib\debug"

```
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.
```

You may be wondering why we do not need include/library paths or library names for BIOS? Any RTSC enabled component in the 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.

7. Now select appropriate RSTC components by right click on project name->properties->Resource->General->RTSC (select PDK and BIOS versions and etc.), and then select appropriate target platform.

8. Build the project.

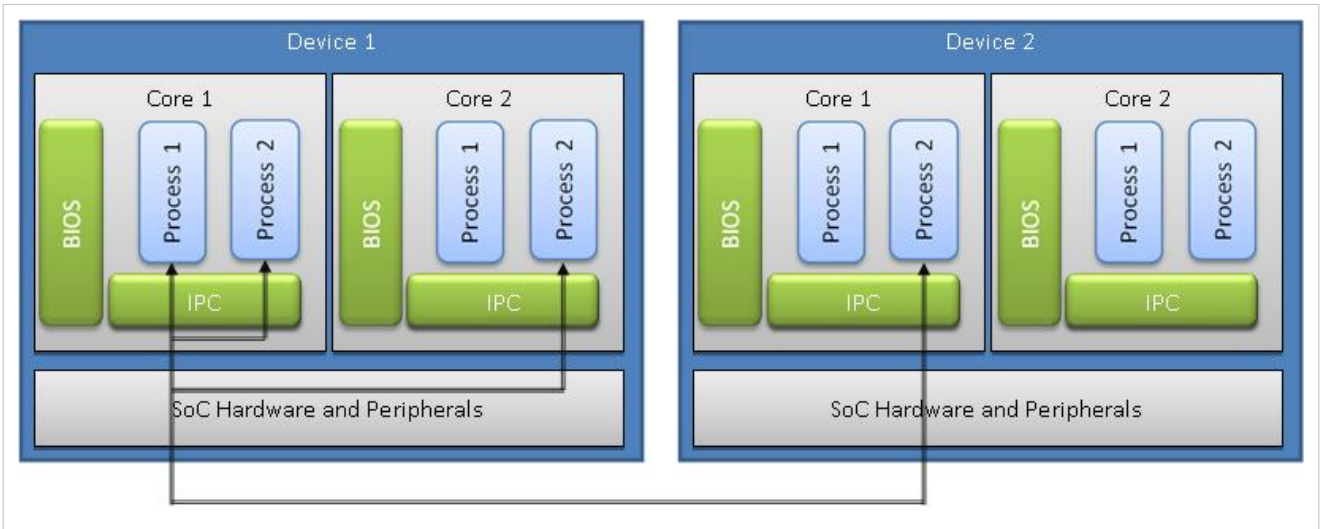
9. Connect to your EVM with your Target Configuration file, then load and run the program on first 4 cores. You should now see LEDs (0-3) blinking one after another.

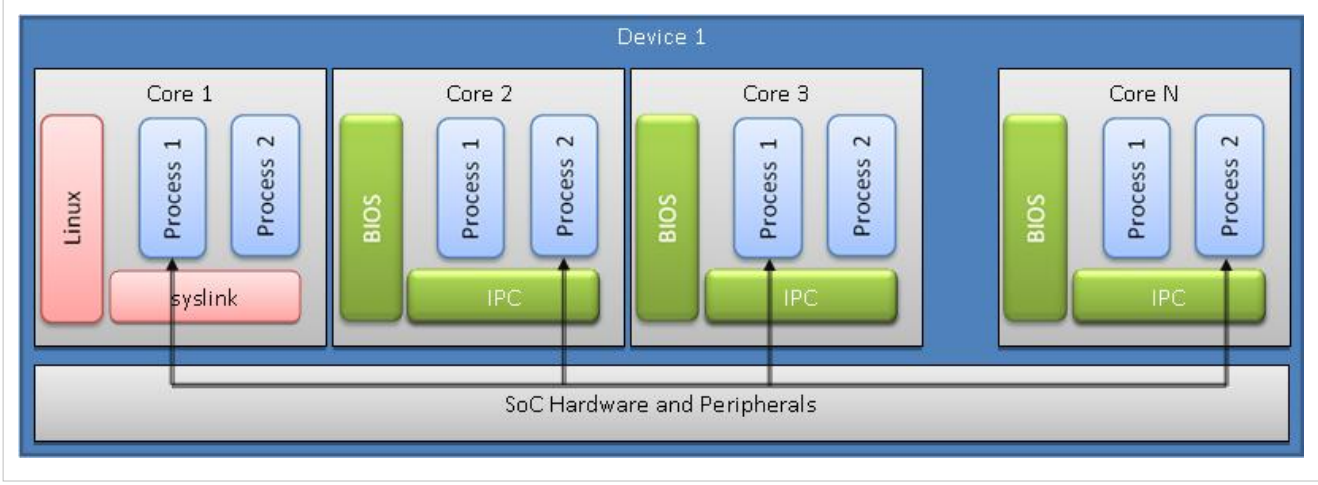
Multi-core Programming Models

Explicit Programming Model using IPC

The MCSDK provides the foundations to support an explicit programming model based on Inter-Processor Communication (IPC). An explicit programming model is one in which the developer analyzes their application and manually partition tasks and processing elements across the cores and devices. In this model the developer is responsible for creating and managing processing tasks, communication between tasks, and data management.

The figures below illustrate the concept in different scenarios including both Linux and BIOS Operating systems.





The IPC provides a processor agnostic API which can be used for communication between processes on the same processing core (inter-process), processes on different cores (inter-core), and processes on different devices (interdevice). For inter-core communication, the transport can be shared memory or leverage the hardware queuing in the KeyStone architecture. And across devices multiple transports can be supported (e.g., SRIO). For all cases, the API is maintained so as to ease the task of migrating tasks and processes across cores and processors as part of designing and tuning an implementation.

IPC Transport Types	Thread 2 Thread	Core 2 Core	Device 2 Device	Description	Examples
Shared Memory	x	x		IPC transport which utilizes the IPC's shared memory region, configured to be in MSMC or DDR3, to pass messages and data between threads and cores on the same chip. Delivered as part of the IPC package: IPC_INSTALL_DIR\packages\ti\sdo\ipc\transports\	Shared Memory transport core to core benchmark example app: PDK_INSTALL_DIR\packages\ti\transport\ipc\examples\shmIpcBenchmark MCSDK Image Processing Demo[1] thread to thread and core to core demo app: MCSDK_INSTALL_DIR\demos\image_processing\
Navigator/QMSS	x	x		IPC transport which utilizes the QMSS IP block to pass pointers to messages and data between threads and cores on the same chip. The messages and data the pointers refer are resident within the global, MSMC and/or DDR3 memory. Delivered as part of the PDK package: PDK_INSTALL_DIR\packages\ti\transport\ipc\qmss\	QMSS transport core to core benchmark example app: PDK_INSTALL_DIR\packages\ti\transport\ipc\examples\qmssIpcBenchmark\
sRIO	x	x	x	IPC transport which utilizes the SRIO IP block to pass messages and data between threads, cores, and devices. Delivered as part of the PDK package: PDK_INSTALL_DIR\packages\ti\transport\ipc\srio\	SRIO transport core to core benchmark example app: PDK_INSTALL_DIR\packages\ti\transport\ipc\examples\srioIpcBenchmark\ SRIO transport device to device example app: PDK_INSTALL_DIR\packages\ti\transport\ipc\examples\srioIpcChipToChipExample\

[1] Image Processing Demo Guide.

Using and Configuring the Navigator/QMSS Transport

The QMSS Transport can be used in place of the shared memory transports delivered as part of the IPC module. This section will describe how to enable the use of and configure the QMSS transport.

Following, snippets from the qmssIpcBenchmark example project's RTSC configuration file, bench_qmss.cfg, included as part of MCSDK will be used to show how an application can utilize and configure the QMSS transport for use in IPC. The qmssIpcBenchmark example is found in pdk_C667#_w_x_y_z\packages\ti\transport\ipc\examples\qmssIpcBenchmark.

Configure IPC to Use the QMSS Transport

```
/* Load and use the CPPI and QMSS packages */ var Cppi = xdc.loadPackage('ti.drv.cppi'); var Qmss = xdc.loadPackage('ti.drv.qmss');
```

Program.sectMap[".qmss"] = new Program.SectionSpec(); Program.sectMap[".qmss"] = "MSMCSRAM";

Program.sectMap[".cppi"] = new Program.SectionSpec(); Program.sectMap[".cppi"] = "MSMCSRAM";

var MessageQ = xdc.useModule('ti.sdo.ipc.MessageQ'); var TransportQmssSetup = xdc.useModule('ti.transport.ipc.qmss.transports.TransportQmssSetup');

MessageQ.SetupTransportProxy = xdc.useModule(Settings.getMessageQSetupDelegate()); MessageQ.SetupTransportProxy = TransportQmssSetup;

The code includes the CPPI and QMSS modules, allocates their global objects in MSMC, and then assigns the use of the QMSS Transport module (TransportQMSS) at the transport layer. Interrupts are tied to queue push actions at the transport layer so the Notify later is not required.

Changing the GEM Interrupt Used by the QMSS Transport Module & Other TransportQmssSetup Parameters

```
TransportQmssSetup.dspIntVectId = 8 /* Desired GEM interrupt */
```

Adding the latter line to the .cfg file after creating the TransportQmssSetup variable allows the application developer to specify which GEM interrupt is used by the QMSS Transport module.

```
TransportQmssSetup.descMemRegion = 0;
```

Adding the latter line to the .cfg file after creating the TransportQmssSetup variable allows the application developer to specify the memory region in which the descriptors were allocated.

TransportQmss Configuration Options

```
var TransportQmss = xdc.useModule('ti.transport.ipc.qmss.transports.TransportQmss');
```

The latter defines a TransportQmss variable in order to access and change the QMSS transport configurations. Use of the TransportQmssSetup module automatically includes the use of the TransportQmss module but this variable must be created in order to access all the TransportQmss transport configuration options.

```
TransportQmss.numDescriptors = 1024;
```

The latter option defines the total number of descriptors to be used by **all** cores. This value should match the number of descriptors inserted in the memory region by the application.

```
TransportQmss.descriptorIsInSharedMem = true;
```

The latter option defines whether the descriptors are placed into shared memory, such as MSMCSRAM or DDR3, or into local L2 memory. If the descriptors are in L2 memory task-to-task communication, within the same core, will only work.

```
TransportQmss.descriptorSize = 128;
```

The latter option defines the descriptor size in bytes. It is recommended this value be equivalent to the cache line size of 128 bytes.

```
TransportQmss.useAccumulatorLogic = false;
```

The latter option defines whether the QMSS transport uses the Accumulator or QPEND queues. If this value is set to true the QPEND queues will be used. If false, the Accumulator queues and logic will be used. As of now, the QPEND queue configuration offers higher throughput and lower latency.

```
TransportQmss.pacingEnabled = false;
```

The latter option defines whether the accumulator accumulation logic is enabled. If this value is set to true the accumulator will interrupt the DSP as soon as intThreshold (next parameter discussed) number of descriptors have been received. Enabling pacing will increase end-to-end delay.

This option is only valid when useAccumulatorLogic is true

```
TransportQmss.intThreshold = 100;
```

The latter option defines the number of descriptors that should be received by the accumulator prior to interrupting the DSP when accumulator pacing is enabled. If pacing is disabled this value should be left at its default of 1.

This option is only valid when useAccumulatorLogic is true

```
TransportQmss.timerLoadCount = 0; // timer ticks. This value only has effect when the pacingEnabled is true.
```

The latter option defines the time the accumulator should wait prior to interrupting the DSP. If the accumulator has not received a number of descriptors equal to intThreshold within the timeout period the accumulator will interrupt the DSP.

This option is only valid when useAccumulatorLogic is true

```
TransportQmss.accuHiPriListSize = 204; // this number should be >= (2*intThreshold)+2
```

The latter option defines the accumulator list size. The list is a ping pong buffer so the accumulator list should be sized as greater than or equal to twice the intThreshold+2. The +2 is for the words included at the start of the ping and pong buffers storing the number of entries in each buffer.

This option is only valid when useAccumulatorLogic is true

TransportQmss Queue Allocation Notes

The QMSS Transport does not hard code which high priority accumulator, or QPEND, queues it uses. The transport initialization code queries the QMSS LLD for the next available high priority, or QPEND, queue. When the queue number is returned by the QMSS LLD the DSP GEM Event to be tied to the specified GEM Interrupt is chosen based on the interrupt map tables in the SPRUGR9 - Keystone Architecture Multicore Navigator document. The tables of interest are in Section 5.3-Interrupt Maps. For the accumulator queues, Table 5-3 is for C6670 devices and Table 5-4 is for C6678 devices. For the QPEND queues, Table 5-6 is for C6670 devices and Table 5-7 is for C6678 devices.

Using and Configuring the sRIO Transport

The sRIO Transport can be used in place of the shared memory transports delivered as part of the IPC module. This section will describe how to enable the use of and configure the sRIO transport.

Following, snippets from the srioIpcBenchmark example project's RTSC configuration file, bench_srio.cfg, included as part of MCSDK will be used to show how an application can utilize and configure the sRIO transport for use in IPC. The srioIpcBenchmark example is found in pdk_C667#_w_x_y_z\packages\ti\transport\ipc\examples\srioIpcBenchmark.

Configure IPC to Use the sRIO Transport

```
/* Load and use the CPPI, QMSS, and SRIO packages */ var Cppi = xdc.loadPackage('ti.drv.cppi'); var Qmss = xdc.loadPackage('ti.drv.qmss'); var Srio = xdc.loadPackage('ti.drv.srio');
```

Program.sectMap[".qmss"] = new Program.SectionSpec(); Program.sectMap[".qmss"] = "MSMCSRAM";

Program.sectMap[".cppi"] = new Program.SectionSpec(); Program.sectMap[".cppi"] = "MSMCSRAM";

Program.sectMap[".srioSharedMem"] = new Program.SectionSpec(); Program.sectMap[".srioSharedMem"] = "MSMCSRAM";

var MessageQ = xdc.module('ti.sdo.ipc.MessageQ'); MessageQ.SetupTransportProxy = xdc.useModule(Settings.getMessageQSetupDelegate());

var TransportSrioSetup = xdc.useModule('ti.transport.ipc.srio.transports.TransportSrioSetup'); MessageQ.SetupTransportProxy = TransportSrioSetup;

The latter code includes the CPPI, QMSS, and sRIO modules, allocates their global objects in MSMCSRAM, and then assigns the use of the sRIO Transport module (TransportSrio) at the transport layer. The Notify layer is not required since sRIO can interrupt the remote core directly via QMSS queue interrupt.

Changing the GEM Interrupt Used by the sRIO Transport Module & Other TransportSrioSetup Parameters

```
TransportSrioSetup.dspIntVectId = 8 /* Desired GEM interrupt */
```

Adding the latter line to the .cfg file after creating the TransportSrioSetup variable allows the application developer to specify which GEM interrupt is used by the sRIO Transport module.

```
TransportSrioSetup.descMemRegion = 0;
```

Adding the latter line to the .cfg file after creating the TransportSrioSetup variable allows the application developer to specify the memory region in which the descriptors were allocated.

```
TransportSrioSetup.numRxDescBuffs = 256;
```

Adding the latter line to the .cfg file after creating the TransportSrioSetup variable allows the application developer to specify the number of descriptor buffers that can be tied to sRIO receive-side descriptors. The number of receive buffers must be at least the number of receive descriptors (TransportSrio.srioNumRxDescriptors) times the number of cores on the local chip. There should be enough buffers such that buffers are still available for tying to receive descriptors while other buffers are being processed by the application.

```
TransportSrioSetup.messageQHeapId = 0;
```

Adding the latter line to the .cfg file after creating the TransportSrioSetup variable allows the application developer to specify the heap ID of the heap from which MessageQ is to allocate the receive-side buffers.

This head ID should not be used by any other module within the system. It is meant solely for the receive-side descriptor buffers.

TransportSrio Configuration Options

```
var TransportSrio = xdc.useModule('ti.transport.ipc.srio.transports.TransportSrio');
```

The latter defines a TransportSrio variable in order to access and change the sRIO transport configurations. Use of the TransportSrioSetup module automatically includes the use of the TransportSrio module but this variable must be created in order to access all the sRIO transport configuration options.

```
TransportSrio.srioNumTxDescriptors = 4; TransportSrio.srioNumRxDescriptors = 4;
```

The latter options define the number of transmit and receive descriptors to be used by each core. For example, if there are two cores in the system each core would be assigned 4 transmit and 4 receive descriptors. The number of descriptors inserted in the memory region by the application should be greater than or equal to ((srioNumTxDescriptors + srioNumRxDescriptors) * number of cores used on chip).

```
TransportSrio.descriptorSize = 128;
```

The latter option defines the descriptor size in bytes. It is recommended this value be equivalent to the cache line size of 128 bytes.

```
TransportSrio.pacingEnabled = true;
```

The latter option defines whether the accumulator accumulation logic is enabled. If this value is set to true the accumulator will interrupt the DSP as soon as intThreshold (next parameter discussed) number of descriptors have been received. Enabling pacing will increase end-to-end delay.

```
TransportSrio.intThreshold = 100;
```

The latter option defines the number of descriptors that should be received by the accumulator prior to interrupting the DSP when accumulator pacing is enabled. If pacing is disabled this value should be left at its default of 1.

```
TransportSrio.timerLoadCount = 0; // timer ticks. This value only has effect when the pacingEnabled is true.
```

The latter option defines the time the accumulator should wait prior to interrupting the DSP. If the accumulator has not received a number of descriptors equal to intThreshold within the timeout period the accumulator will interrupt the DSP.

```
TransportSrio.accuHiPriListSize = 204; // this number should be >= (2*intThreshold)+2
```

The latter option defines the accumulator list size. The list is a ping pong buffer so the accumulator list should be sized as greater than or equal to twice the intThreshold+2. The +2 is for the words included at the start of the ping and pong buffers storing the number of entries in each buffer.

```
TransportSrio.srioMaxMtuSizeBytes = 256;
```

The latter option defines the maximum transmissible unit by sRIO in bytes. The maximum value that can be specified is 256 bytes.

```
TransportSrio.numTxDescToCleanUp = 1;
```

The latter option defines the number of descriptors to cleanup each time TransportSrio_put is called. After sRIO sends out data associated with a descriptor provided by the TransportSrio_put function, sRIO will put the descriptor into a transmit completion queue. The next time TransportSrio_put is invoked it will check the transmit completion queue for descriptors, and their associated buffers, to clean up. If the number of descriptors in the transmit completion queue equals this setting it will cleanup the the defined number of descriptors and buffers.

```
TransportSrio.srioGarbageQ = "defined SRIO garbage queue value";
```

The latter option defines the sRIO garbage queue for which the SRIO transport should check for descriptors to cleanup. The sRIO hardware can be assigned up to six separate QMSS queues which are used as repositories for descriptors which failed to send because of different errors. The sRIO transport has the ability to check one of these queues for descriptors, and their associated buffers, to clean up. The application can specify six separate queues for each sRIO failure type or can tie one or more failure types to a single garbage queue. This allows the SRIO transport to clean up anywhere from one to all six failure types. The cleanup process occurs every TransportSrio_put operation.

TransportSrio Core Map Configuration and IPC Cluster Parameters

The sRIO transport is a multi-chip transport, allowing communication between two or more cores on separate chips. This attribute means that each chip running the sRIO transport must contain a copy of the core address array configurations. This copy must be exactly the same across all chips. The multi-chip capabilities of the sRIO transport are facilitated by the IPC cluster support. The IPC cluster support allows the core map to remain the same across all chips. Based on the IPC cluster base defined for each chip the sRIO core map is indexed in the transport to find the proper address for the destination core.

TransportSrio Single Device Core Map and IPC Cluster Configuration

This section covers the sRIO transport core map and IPC cluster configuration for a system that contains two cores within the same device communicating with one another. This scenario is illustrated by the srioIpcBenchmark example project and the code covered below is taken directly from the bench_srio.cfg file.

```
Program.global.Srio8BitDeviceId1 = 0xAB;
```

The latter operation defines the only valid device ID for data routed through the sRIO IP block. This value or any other device IDs must match with any device IDs used to set the sRIO TLM Base Routing Pattern Match information. In the srioIpcBenchmark example the pattern match information is set in the SrioDevice_init function in device_srio.c.

```
TransportSrio.srioMaxNumSystemCores = 2;
```

The latter option defines the total number of cores across all chips contained in the system. For this case, there is only one chip with only two cores on the chip being utilized.

```
TransportSrio.srioCoreTT.length = TransportSrio.srioMaxNumSystemCores;
TransportSrio.srioCoreTT[0] = 0; TransportSrio.srioCoreTT[1] = 0;
```

The srioCoreTT array specifies whether each core's socket uses 16 or 8-bit identifiers (deviceIDs as named in this example). The srioCoreTT array should have as many entries as there are cores in the system. The srioCoreTT array is sized to the maximum number of system cores prior to assigning a value to each entry in the array. For information on valid srioCoreTT settings please refer to pdk_C667#_w_x_y_z\packages\ti\transport\ipc\srio\transports\TransportSrio.xdc.

```
TransportSrio.srioCoreDeviceId.length = TransportSrio.srioMaxNumSystemCores;
TransportSrio.srioCoreDeviceId[0] = Program.global.Srio8BitDeviceId1;
TransportSrio.srioCoreDeviceId[1] = Program.global.Srio8BitDeviceId1;
```

The srioCoreDeviceId array specifies the deviceID assigned to each core's sRIO socket. The srioCoreDeviceId array should have as many entries as there are cores in the system. The srioCoreDeviceId array is sized to the maximum number of system cores prior to assigning a value to each entry in the array. For information on valid srioCoreDeviceId settings please refer to pdk_C667#_w_x_y_z\packages\ti\transport\ipc\srio\transports\TransportSrio.xdc.

```
TransportSrio.srioCoreMailbox.length = TransportSrio.srioMaxNumSystemCores;
TransportSrio.srioCoreMailbox[0] = 0; TransportSrio.srioCoreMailbox[1] = 0;
```

The srioCoreMailbox array specifies the mailbox number assigned to each core's sRIO socket. The srioCoreMailbox array should have as many entries as there are cores in the system. The srioCoreMailbox array is sized to the maximum number of system cores prior to assigning a value to each entry in the array. For information on valid srioCoreMailbox settings please refer to pdk_C667#_w_x_y_z\packages\ti\transport\ipc\srio\transports\TransportSrio.xdc.

```
TransportSrio.srioCoreLetter.length = TransportSrio.srioMaxNumSystemCores;
TransportSrio.srioCoreLetter[0] = 0; TransportSrio.srioCoreLetter[1] = 1;
```

The srioCoreLetter array specifies the letter number assigned to each core's sRIO socket. The srioCoreLetter array should have as many entries as there are cores in the system. The srioCoreLetter array is sized to the maximum number of system cores prior to assigning a value to each entry in the array. For information on valid srioCoreLetter settings please refer to pdk_C667#_w_x_y_z\packages\ti\transport\ipc\srio\transports\TransportSrio.xdc.

```
TransportSrio.srioCoreSegMap.length = TransportSrio.srioMaxNumSystemCores;
TransportSrio.srioCoreSegMap[0] = 0; TransportSrio.srioCoreSegMap[1] = 0;
```

The srioCoreSegMap array specifies the segmentation mapping for core's sRIO socket. The srioCoreSegMap array should have as many entries as there are cores in the system. The srioCoreSegMap array is sized to the maximum number of system cores prior to assigning a value to each entry in the array. For information on valid srioCoreSegMap settings please refer to pdk_C667#_w_x_y_z\packages\ti\transport\ipc\srio\transports\TransportSrio.xdc.

```
var procName = null;
```

This option can be used to define the MultiProc ID for cores prior to runtime. Typically, this option is set to null and the MultiProc ID for each core is set at runtime.

```
var procNameList = []; procNameList = ["CORE0", "CORE1"];
```

This option defines the number of cores on this chip that will be used.

```
var MultiProc = xdc.useModule('ti.sdo.utils.MultiProc');
```

The latter option defines a MultiProc variable for use in setting the cluster configurations.

```
MultiProc.numProcessors = TransportSrio.srioMaxNumSystemCores;
```

The latter option sets the number of processors in the entire system, across all chips. For this case the number of cores is 2, or srioMaxNumSystemCores.

baseIdOfCluster and numProcessors must be set BEFORE setConfig is run

```
MultiProc.baseIdOfCluster = 0;
```

The latter option sets the base cluster ID for this chip. In this case, there is only one chip with two cores. The base ID is zero.

baseIdOfCluster and numProcessors must be set BEFORE setConfig is run

```
MultiProc.setConfig(procName, procNameList);
```

The latter function sets up the MultiProc module using the specified processor and cluster information.

TransportSrio Multi-Device Core Map and IPC Cluster Configuration

This section covers the sRIO transport core map and IPC cluster configuration for a system that contains two devices with two cores each, for a total four cores, communicating with one another. This scenario is illustrated by the srioIpcChipToChipExample project.

Device One (Producer) Configuration

This section covers the core map and IPC cluster configuration settings for the first, producer device within the system. As previously noted, each device .cfg file must map every core within the system. This scenario is illustrated by the SrioIpcChipToChipExample\producer example project and the code covered below is taken directly from the producer_srio.cfg file.

```
Program.global.Srio8BitDeviceId1 = 0xAB Program.global.Srio8BitDeviceId2 = 0xCD
```

The latter operations define the only valid device IDs for data routed through the sRIO IP block. These values or any other device IDs must match with any device IDs used to set the sRIO TLM Base Routing Pattern Match information. In the srioIpcBenchmark example the pattern match information is set in the SrioDevice_init function in device_srio.c.

```
TransportSrio.srioMaxNumSystemCores = 4;
```

The latter option defines the total number of cores across all chips contained in the system. There are two cores on device one and two cores on device two, for a total of four cores being utilized.

```
TransportSrio.srioCoreTT.length = TransportSrio.srioMaxNumSystemCores;
TransportSrio.srioCoreTT[0] = 0; TransportSrio.srioCoreTT[1] = 0;
TransportSrio.srioCoreTT[2] = 0; TransportSrio.srioCoreTT[3] = 0;
```

The srioCoreTT array specifies whether each core's socket uses 16 or 8-bit identifiers (deviceIDs, as named in this example). The srioCoreTT array should have as many entries as there are cores in the system. The srioCoreTT array is sized to the maximum number of system cores prior to assigning a value to each entry in the array. For information on valid srioCoreTT settings please refer to pdk_C667#_w_x_y_z\packages\ti\transport\ipc\srio\transports\TransportSrio.xdc.

```
TransportSrio.srioCoreDeviceId.length = TransportSrio.srioMaxNumSystemCores;
TransportSrio.srioCoreDeviceId[0] = Program.global.Srio8BitDeviceId1;
TransportSrio.srioCoreDeviceId[1] = Program.global.Srio8BitDeviceId1;
TransportSrio.srioCoreDeviceId[2] = Program.global.Srio8BitDeviceId2;
TransportSrio.srioCoreDeviceId[3] = Program.global.Srio8BitDeviceId2;
```

The srioCoreDeviceId array specifies the deviceID assigned to each core's sRIO socket. The srioCoreDeviceId array should have as many entries as there are cores in the system. The srioCoreDeviceId array is sized to the maximum number of system cores prior to assigning a value to each entry in the array. For information on valid srioCoreDeviceId settings please refer to pdk_C667#_w_x_y_z\packages\ti\transport\ipc\srio\transports\TransportSrio.xdc.

```
TransportSrio.srioCoreMailbox.length = TransportSrio.srioMaxNumSystemCores;
TransportSrio.srioCoreMailbox[0] = 0; TransportSrio.srioCoreMailbox[1] = 0;
TransportSrio.srioCoreMailbox[2] = 0; TransportSrio.srioCoreMailbox[3] = 0;
```

The srioCoreMailbox array specifies the mailbox number assigned to each core's sRIO socket. The srioCoreMailbox array should have as many entries as there are cores in the system. The srioCoreMailbox array is sized to the maximum number of system cores prior to assigning a value to each entry in the array. For information on valid srioCoreMailbox settings please refer to pdk_C667#_w_x_y_z\packages\ti\transport\ipc\srio\transports\TransportSrio.xdc.

```
TransportSrio.srioCoreLetter.length = TransportSrio.srioMaxNumSystemCores;
TransportSrio.srioCoreLetter[0] = 0; TransportSrio.srioCoreLetter[1] = 1;
TransportSrio.srioCoreLetter[2] = 0; TransportSrio.srioCoreLetter[3] = 1;
```

The srioCoreLetter array specifies the letter number assigned to each core's sRIO socket. The srioCoreLetter array should have as many entries as there are cores in the system. The srioCoreLetter array is sized to the maximum number of system cores prior to assigning a value to each entry in the array. For information on valid srioCoreLetter settings please refer to pdk_C667#_w_x_y_z\packages\ti\transport\ipc\srio\transports\TransportSrio.xdc.

```
TransportSrio.srioCoreSegMap.length = TransportSrio.srioMaxNumSystemCores;
TransportSrio.srioCoreSegMap[0] = 0; TransportSrio.srioCoreSegMap[1] = 0;
TransportSrio.srioCoreSegMap[2] = 0; TransportSrio.srioCoreSegMap[3] = 0;
```

The srioCoreSegMap array specifies the segmentation mapping for core's sRIO socket. The srioCoreSegMap array should have as many entries as there are cores in the system. The srioCoreSegMap array is sized to the maximum number of system cores prior to assigning a value to each entry in the array. For information on valid srioCoreSegMap settings please refer to pdk_C667#_w_x_y_z\packages\ti\transport\ipc\srio\transports\TransportSrio.xdc.

```
var procName = null;
```

This option can be used to define the MultiProc ID for cores prior to runtime. Typically, this option is set to null and the MultiProc ID for each core is set at runtime.

```
var procNameList = []; procNameList = ["CORE0", "CORE1"];
```

This option defines the number of cores on this chip that will be used.

```
var MultiProc = xdc.useModule('ti.sdo.utils.MultiProc');
```

The latter option defines a MultiProc variable for use in setting the cluster configurations.

```
MultiProc.numProcessors = TransportSrio.srioMaxNumSystemCores;
```

The latter option sets the number of processors in the entire system, across all chips. For this case the number of cores is 2, or srioMaxNumSystemCores.

baseIdOfCluster and numProcessors must be set BEFORE setConfig is run

```
MultiProc.baseIdOfCluster = 0;
```

The latter option sets the base cluster ID for this chip. In this case, the Producer chip contains the first two cores in the system. Therefore, the cluster base ID for this chip is 0.

baseIdOfCluster and numProcessors must be set BEFORE setConfig is run

```
MultiProc.setConfig(procName, procNameList);
```

The latter function sets up the MultiProc module using the specified processor and cluster information.

Device Two (Consumer) Configuration

This section covers the core map and IPC cluster configuration settings for the second, consumer device within the system. As previously noted, each device .cfg file must map every core within the system. This scenario is illustrated by the SrioIpcChipToChipExample\consumer example project and the code covered below is taken directly from the consumer_srio.cfg file.

```
Program.global.Srio8BitDeviceId1 = 0xAB Program.global.Srio8BitDeviceId2 = 0xCD
```

TransportSrio.srioMaxNumSystemCores = 4;

TransportSrio.srioCoreTT.length = TransportSrio.srioMaxNumSystemCores; TransportSrio.srioCoreTT[0] = 0; TransportSrio.srioCoreTT[1] = 0; TransportSrio.srioCoreTT[2] = 0; TransportSrio.srioCoreTT[3] = 0;

TransportSrio.srioCoreDeviceId.length = TransportSrio.srioMaxNumSystemCores; TransportSrio.srioCoreDeviceId[0] = Program.global.Srio8BitDeviceId1; TransportSrio.srioCoreDeviceId[1] = Program.global.Srio8BitDeviceId1; TransportSrio.srioCoreDeviceId[2] = Program.global.Srio8BitDeviceId2; TransportSrio.srioCoreDeviceId[3] = Program.global.Srio8BitDeviceId2;

TransportSrio.srioCoreMailbox.length = TransportSrio.srioMaxNumSystemCores; TransportSrio.srioCoreMailbox[0] = 0; TransportSrio.srioCoreMailbox[1] = 0; TransportSrio.srioCoreMailbox[2] = 0; TransportSrio.srioCoreMailbox[3] = 0;

TransportSrio.srioCoreLetter.length = TransportSrio.srioMaxNumSystemCores; TransportSrio.srioCoreLetter[0] = 0; TransportSrio.srioCoreLetter[1] = 1; TransportSrio.srioCoreLetter[2] = 0; TransportSrio.srioCoreLetter[3] = 1;

TransportSrio.srioCoreSegMap.length = TransportSrio.srioMaxNumSystemCores; TransportSrio.srioCoreSegMap[0] = 0; TransportSrio.srioCoreSegMap[1] = 0; TransportSrio.srioCoreSegMap[2] = 0; TransportSrio.srioCoreSegMap[3] = 0;

All the latter commands match exactly with what was defined for the producer device. For the sRIO transport to work all device's must have the same knowledge of the global core map. As a result, all the latter information must not change between device .cfg files.

```
var procName = null; var procNameList = [];
```

procNameList = ["CORE0", "CORE1"];

var MultiProc = xdc.useModule('ti.sdo.utils.MultiProc'); MultiProc.numProcessors = TransportSrio.srioMaxNumSystemCores; MultiProc.baseIdOfCluster = 2; MultiProc.setConfig(procName, procNameList);

The latter options configure MultiProc for the Consumer chip. For this case, the Consumer chip contains the last two cores in the system. Therefore, the cluster base ID for this chip is 2.

baseIdOfCluster and numProcessors must be set BEFORE setConfig is run

TransportSrio Queue Allocation Notes

The sRIO transport does not hardcode which general purpose, sRIO, or high priority accumulator queues it uses. The transport initialization code queries the QMSS LLD for the next available queues. When the queue number is returned for the high priority accumulator queues by the QMSS LLD the DSP GEM Event to be tied to the specified GEM Interrupt is chosen based on the interrupt map tables in the SPRUGR9 - Keystone Architecture Multicore Navigator document. The tables of interest are in Section 5.3-Interrupt Maps. Table 5-3 is for C6670 devices and Table 5-4 is for C6678 devices.

TransportSrio Application Configuration Requirements

In order to use the sRIO IPC transport to communicate with a core off-chip a couple rules must be followed when settings up the transport in the application.

1. A core to be used to communicate with an off-chip core must attach to at least one local core prior to communicating off-chip. The first invocation of Ipc_attach for a core will result in the sRIO transport starting up and configuring itself for send/receive. The IPC cluster mechanism does not enable attaching to core's off-chip. Those connection are setup manually. Therefore, at least one local IPC attach is required in order to setup and configure the sRIO transport. This local attach must be done in the context of main prior to BIOS_start enabling interrupts.

2. A core's connections to off-chip cores must be registered manually. Manual registration must be done after the local Ipc_attach is performed and before BIOS_start runs, enabling interrupts. A connection must be registered for each off-chip core that is to be communicated with. The following code gives an example of how to manually register an off-chip core connection:

```
/* NameServerMessageQ and SRIO Transport handles are global so they can be deleted
* in task context when execution completes. */
NameServerMessageQ_Handle nsHandle = NULL; TransportSrio_Handle srioHandle = NULL;
Int main(Int argc, Char* argv[]) {
 Error_Block eb;
 ...
 Attach_to_local_cores();
 ...
 /* Create messageQ to remote proc .  This will use srioTransport to send/receive nameserver
  * messages to/from remote chip.  A MessageQ heap must be registered prior to calling
  * NameServerMessageQ_create()*/
 Error_init(&eb);
 nsHandle = NameServerMessageQ_create(off_chip_core_multiProc_id, NULL, &eb);
 if (nsHandle == NULL)
 {
   System_abort("NameServerMessageQ_create() failed");
 }
 /* Register a transport for messages received from off-chip cores */
 Error_init(&eb);
 srioHandle = TransportSrio_create(off_chip_core_multiProc_id, NULL, &eb);
 ...
 /* Start BIOS and all defined tasks.  Function will not return since it acts as the scheduler. */
 BIOS_start();
/* should not reach here */
 return (0);
}
```

3. Attempts to open a MessageQ located on an off-core chip must be done after the manual connection to the off-chip core has been created and after BIOS_start() has enabled interrupts. When a core attempts to open an MessageQ located on an off-chip core, the NameServerMesssageQ uses the sRIO transport to send a NameServer request message to the off-chip core. In order to service the request and send a response back the remote off-chip core must have the sRIO transport up and running and have a manual connection to the requesting core created. If a requesting core tries to make a NameServer request to an off-chip core that is not ready yet, the NameServerMessageQ request functionality will timeout. At that point the application can wait then try to open the MessageQ at a later time. The timeout period to wait for a NameServer response can be configured in the .cfg file with the following commands. The resolution of the timeout value is microseconds.

```
var NameServerMessageQ = xdc.useModule('ti.sdo.ipc.nsremote.NameServerMessageQ');
NameServerMessageQ.timeoutInMicroSecs = 1000000; /* 1 sec */
```

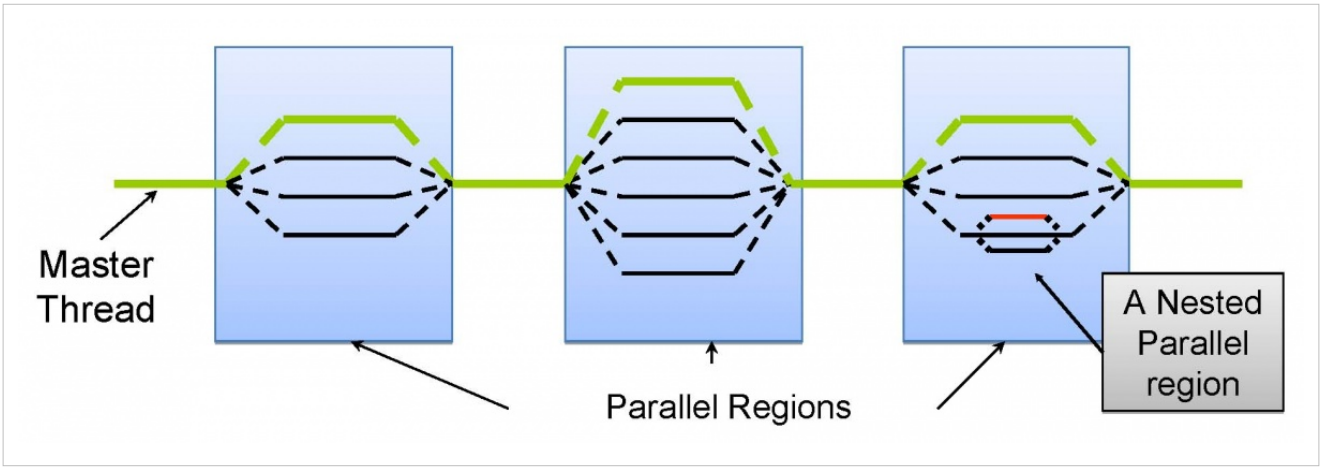
For a working example of how to use the multi-chip IPC and the sRIO transport for device to device communication please examine the producer and consumer RTSC projects in the directory pdk_C667#_w_x_y_z\packages\ti\transport\ipc\examples\srioIpcChipToChipExample. The project .cfg and .c files have been highlighted in the latter sections but contain more in-line comments regarding the use of the sRIO transport.

Programming Model using OpenMP

OpenMP is the industry standard for shared memory parallel programming in C, C++, or Fortran. It provides portable high-level programming constructs that enable users to easily expose a program's task and loop level parallelism in an incremental fashion. With OpenMP, users specify the parallelization strategy for a program at a high level by annotating the program code with compiler directives that specify how a region of code is executed by a team of threads. The compiler works out the detailed mapping of the computation to the machine. The OpenMP programming API enables the programmer to perform the following:

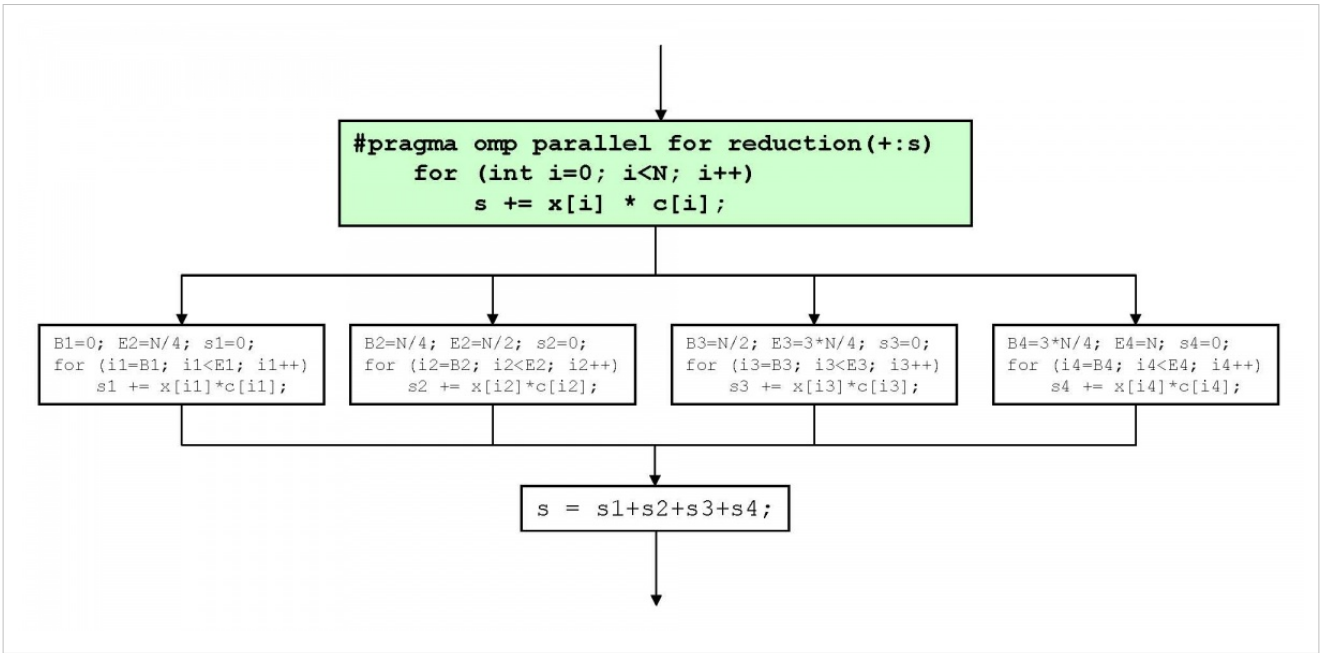
- Create and manage threads
- Assign and distribute work (tasks) to threads
- Specify which data is shared among threads and which data is private
- Coordinate thread access to shared data

As shown in the following figure, OpenMP is a thread-based programming language. The master thread executes the sequential parts of a program. When the master thread encounters a parallel region, it forks a team of worker threads that along with the master thread execute in parallel.



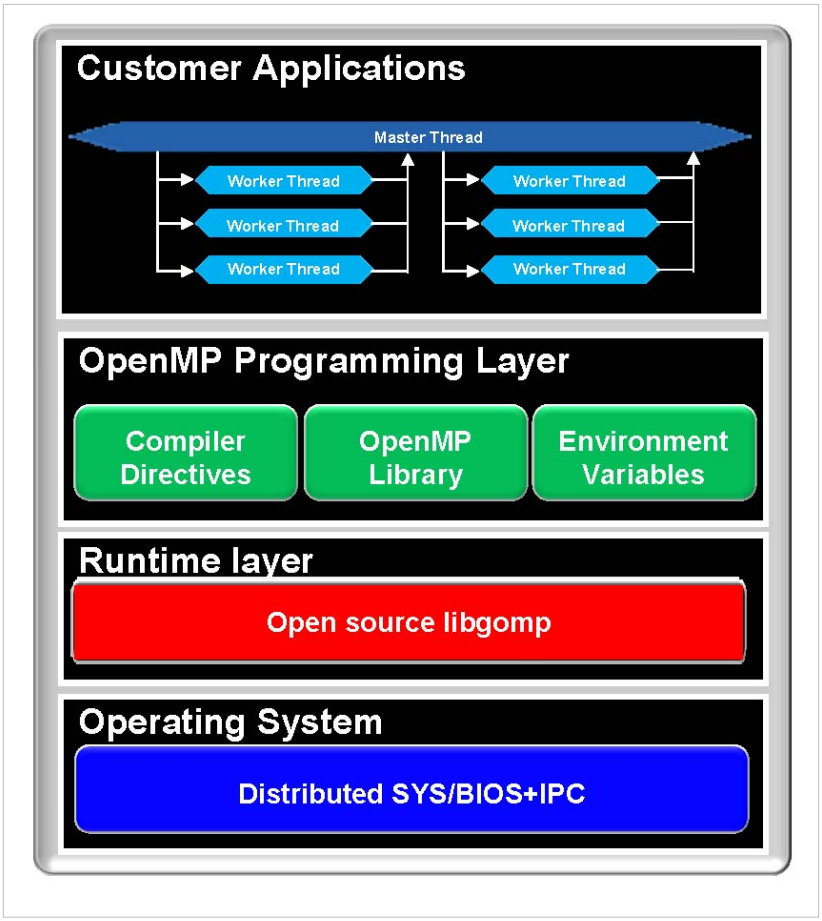
There is a fairly easy migration for existing code base - C/C++ based directives (#pragma) - used to express parallelism. OpenMP directives specify that a well-structured region of code is executed by a collection of threads that share in the work. Worksharing directives are provided to effect a distribution of work among the participating threads. The programmer incrementally adds OpenMP pragmas to an existing sequential application allowing them to quickly port code to a multicore platform.

The following figure is an example of data-parallelism. A parallel-for loop where each thread executes a chunk of the loop and their intermediate results are reduced to a final result. A single copy of x[] and c[] is shared by all the threads.



The following figure shows the OpenMP solution stack. The OpenMP API is made up of directives(#pragmas), function calls, and environment variables. The compiler translates the OpenMP API into multi-threaded code with calls to a custom runtime library that implements support for thread management, shared memory and synchronization.

The OpenMP run-time for SYS/BIOS (OMP) library implements the bottom two layers of the OpenMP solution stack. Currently, OpenMP is supported on TI DSPs only for SYS/BIOS operating system. All OpenMP programs must be linked with the OMP run-time library.



See also:

- http://openmp.org/wp/www.openMP.org for more tutorials, references, online tutorials for OpenMP programming

Compiling OpenMP code with the TI compiler using Makefile

The TI compiler (version 7.4 or higher) includes support for OpenMP 3.0.

To enable support for OpenMP in the compiler you will need to use the --openmp command line option.

The number of threads available to an OpenMP program is determined by the configuration of the OMP run-time.

Hello World example OpenMP program:

```
/* omp-hello.c */ include <stdlib.h> include <stdio.h> include <ti/omp/omp.h>
include <ti/omp/libgomp_g.h>
```

int main (int argc, char *argv[]) {

```
int nthreads, tid;
```

```
/* Fork a team of threads giving them their own copies of variables */
```

1. pragma omp parallel private(nthreads, tid)

```
{
```

```
/* Obtain thread number */
tid = omp_get_thread_num();
printf("Hello World from thread = %d\n", tid);
```

```
/* Only master thread does this */
if (tid == 0)
{
nthreads = omp_get_num_threads();
printf("Number of threads = %d\n", nthreads);
}
```

```
} /* All threads join master thread and disband */
```

```
return 0;
```

}

You may generate prebuilt C libraries against which an OpenMP application can be compiled and linked. A typical build flow involves building the prebuilt library once for a given device (i.e. evm6678) and for a specific RTSC configuration.

The files needed to generate the prebuilt libraries can be found in [OMP_INSTALL_DIR]\preconfig directory.

1) Edit **ompdefault.cfg** as needed to match your desired RTSC configuration.

2) Edit the **makeomplibs** file as needed:

a. Point to your BIOS, IPC, PDK, XDCTools and OMP products

b. Change the build profile as needed

c. Change the build platform as needed

3) Build the prebuilt libraries

```
$ make -f makeomplibs omp-evm6678
```

4) Edit **Makefile** as follows:

a. Edit the path to the C6x OpenMP-aware codegen tools

b. Add application build goals to the Makefile using the example for 'omp_hello' provided as a guideline.

5) Build the application: `$ make omp_hello.xe66`

The above procedure would produce a hello.out core executable which needs to be loaded and run on CORE0 only.

Using OpenMP on TI devices

Memory Coherency

OpenMP has shared and private variables. Each thread has its own copy of a private variable that the other threads cannot access. OpenMP specifies a relaxed consistency shared memory model. Threads executing in parallel have a temporary view of shared memory until they reach memory synchronization or flush points in the execution flow.

- It is currently the programmers responsibility to maintain the consistency of shared variables that are allocated to cachable memory. Something like:

```
/* process elements of shared_array in parallel*/ pragma omp parallel for
```

```
for (i=0; i<N; i++)
shared_array[i] = do_stuff(shared_array[i]);
```

```
/* write-back invalidate each thread/core's cache */
```

1. pragma omp parallel

```
{
Cache_wbInvAll();
```

```
_mfence();
}
```

- All global and static variables are shared. All dynamically allocated memory is shared.
- Stacks must also be placed in shared memory since a stack variable can be shared.
- If a variable is smaller than a cache line it is possible for two cores to cache the line that contains the variable. In this case, the last core to write the cache line will over-write in shared memory the other core's version of the variable.

Threadprivate Memory

- The compiler allocates threadprivate variables into the .threadprivate section. The execution model assumes that the .threadprivate section is allocated by the linker into the L2 private memory.
- The above restriction will be removed once the compiler tools implement support for thread local storage.
- When using threadprivate, only one thread can be assigned to each core.

Known issues

- The collapse clause is not supported
- Error messages are sparse
- Goto in/out of a parallel region is not flagged as an error

Examples

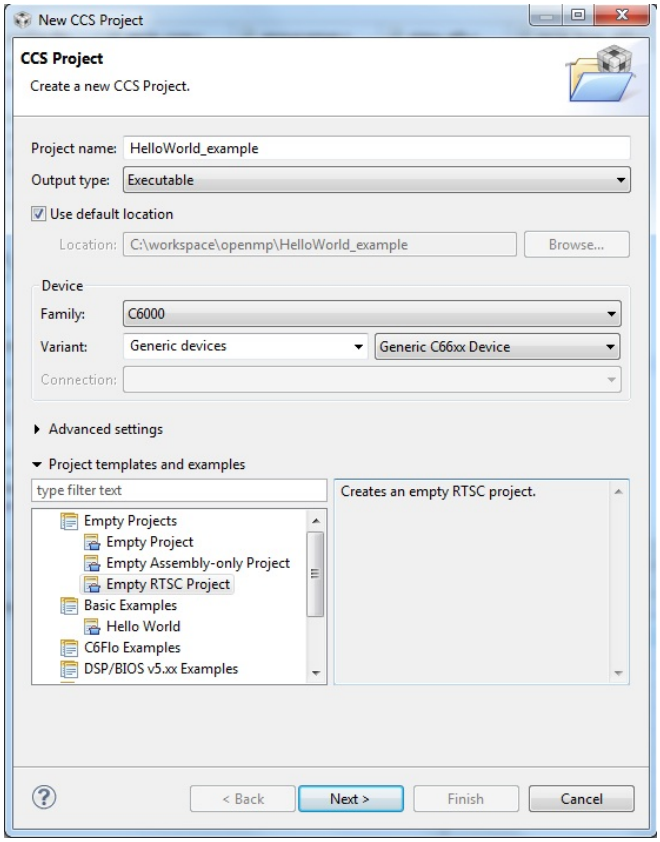
The example programs are designed to familiarize you with the various steps required to create, compile, and run and OpenMP program. Besides these examples are are additional examples included under the OMP (e.g.,*\OMP_xx_xx_xx\packages\examples*).

Multicore Hello World Example

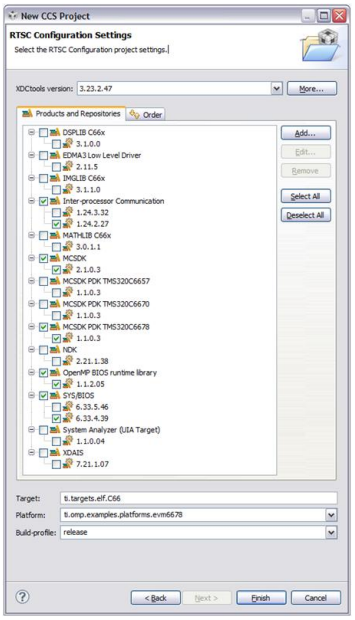
This is the first example OpenMP program. It's purpose is to get you used to creating projects in CCS, building an executable and then running it on your EVM.

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 *HelloWorld_example*.
- In the CCS project window, select *Project Type*: as *C6000*.
- In the *New CCS Project*, select *Device Variant*: as *Generic C66xx Device*.
- In the *Project Templates* window select *Empty RTSC Project* and hit *Next*. See figure below.
- Configure your RTSC settings. The packages that need to be selected, are as per the snapshot in instruction #2 below.
- It should open an empty project with name *HelloWorld_example*.



2. Configure your RTSC settings. The following packages needs to be selected as shown in the snapshot below: BIOS, IPC, OpenMP, PDK, and MCSDK:



3. Now that we have a project, we are going to create a source file for the project.

- Select *File->New->Source File*, enter *Source File* name as *helloworld.c*, then hit *Finish*.
- It should open *helloworld.c* empty file in the eclipse editor. Paste following source code in the editor

```
/******************************************************************************

* FILE: omp_hello.c
* DESCRIPTION:
* OpenMP Example - Hello World - C/C++ Version
```

```
* In this simple example, the master thread forks a parallel region.
* All threads in the team obtain their unique thread number and print it.
* The master thread only prints the total number of threads. Two OpenMP
* library routines are used to obtain the number of threads and each
* thread's number.
* AUTHOR: Blaise Barney 5/99
* LAST REVISED: 04/06/05
* UPDATED: For BIOS MCSDK
******************************************************************************/
include <ti/omp/omp.h>
```

1. include <string.h>
2. include <assert.h>
3. include <stdio.h>
4. include <time.h>
5. include "ti/platform/platform.h"
6. include "ti/platform/resource_mgr.h"

1. define NTHREADS 8

void main() {

```
int nthreads, tid;
```

```
nthreads = NTHREADS;
```

```
omp_set_num_threads(NTHREADS);
```

```
/* Fork a team of threads giving them their own copies of variables */
```

1. pragma omp parallel private(nthreads, tid)

```
{
```

```
/* Obtain thread number */
tid = omp_get_thread_num();
printf("Hello World from thread = %d\n", tid);
```

```
/* Only master thread does this */
if (tid == 0)
{
nthreads = omp_get_num_threads();
printf("Number of threads = %d\n", nthreads);
}
```

```
} /* All threads join master thread and disband */
```

}

4. Create a new .cfg File by right clicking your project and selecting *New* --> *File*. Name this file helloworld.cfg and copy the source code:

```
/*
* Copyright 2012 by Texas Instruments Incorporated.
*
```

```
*/
```

```
var OpenMP = xdc.useModule('ti.omp.utils.OpenMP'); var System = xdc.useModule("xdc.runtime.System"); var SysMin = xdc.useModule("xdc.runtime.SysMin"); System.SupportProxy = SysMin; SysMin.bufSize = 0x8000;
```

/* Increase local heap size */ var BIOS = xdc.useModule('ti.sysbios.BIOS'); BIOS.heapSize = 0x20000;

/* Use more efficient Notify driver */ var Notify = xdc.module('ti.sdo.ipc.Notify'); Notify.SetupProxy = xdc.module('ti.sdo.ipc.family.c647x.NotifyCircSetup');

/* Use more efficient MessageQ transport */ var MessageQ = xdc.module('ti.sdo.ipc.MessageQ'); MessageQ.SetupTransportProxy = xdc.useModule('ti.sdo.ipc.transports.TransportShmNotifySetup');

var System = xdc.useModule('xdc.runtime.System'); System.extendedFormats = "%f";

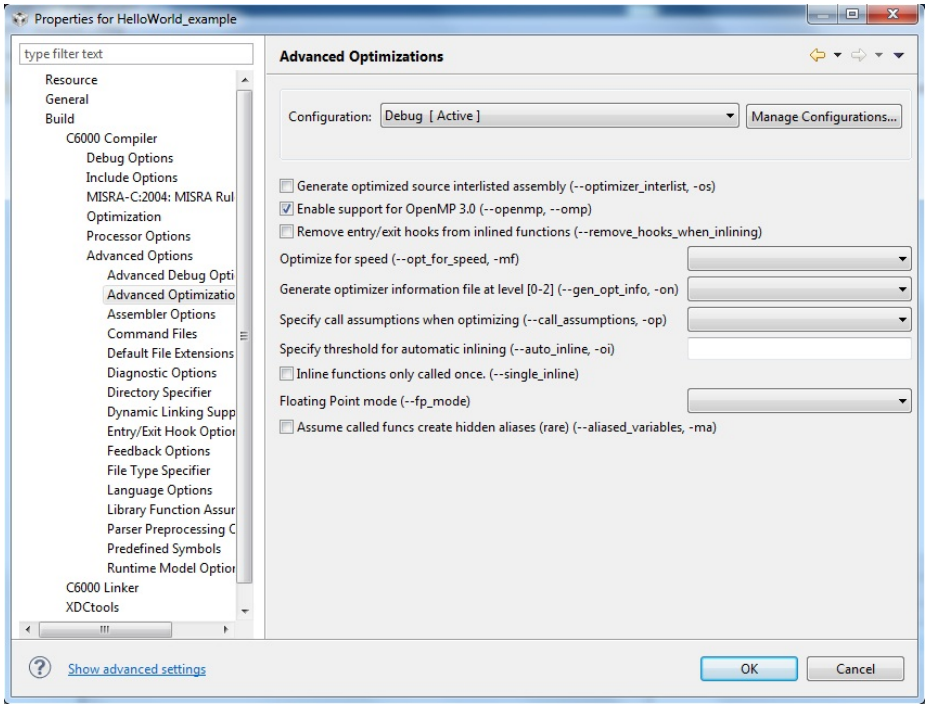
OpenMP.setNumProcessors(8);

/* Create HeapOMP for shared heap */ var SharedRegion = xdc.useModule('ti.sdo.ipc.SharedRegion'); var HeapOMP = xdc.useModule('ti.omp.utils.HeapOMP'); HeapOMP.sharedRegionId = 2; HeapOMP.localHeapSize = 0x20000; HeapOMP.sharedHeapSize = 0x1000000; // Specify the Shared Region SharedRegion.setEntryMeta(HeapOMP.sharedRegionId,

```
                            {   base: 0x90000000,
                                len:  HeapOMP.sharedHeapSize,
                                ownerProcId: 0,
                                createHeap: true,
                                isValid: true,
                                name: "HeapOMP",
                            }
                           );
```

var Cache = xdc.useModule('ti.sysbios.family.c66.Cache'); Cache.setMarMeta(0x90000000, 0x10000000, Cache.PC | Cache.WTE);

5. Enable OpenMP compile option by right clicking your project and selecting Properties. Navigate to: *Build --> C6000 Compiler --> Advanced Options --> Advanced Optimizations*. Tick the checkbox that says "Enable support for OpenMP 3.0 (--openmp, --omp)".

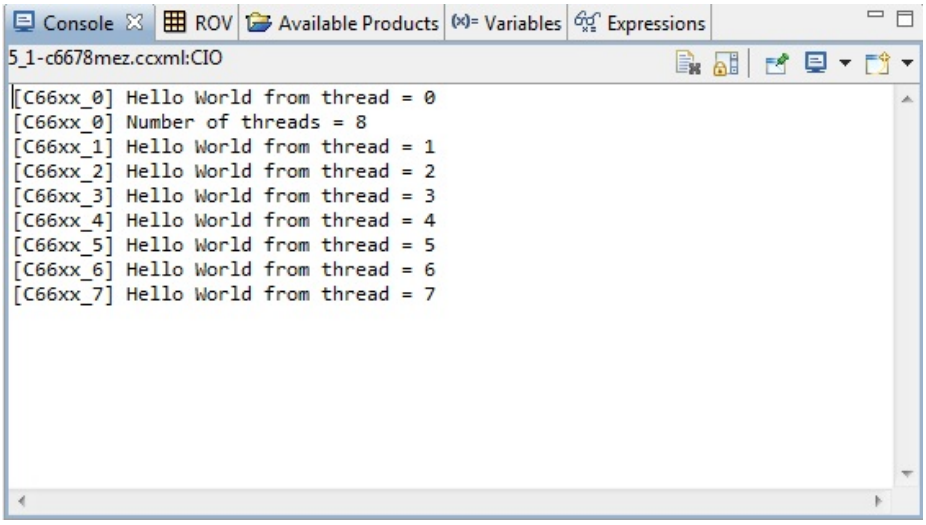


6. Build your project by right clicking your project and select *Build Project*.

7. Connect and power your device. Launch your configuration file and connect to core0. For more information on connecting your device with CCS, refer to the 2.0.x User Guide.

8. Load your helloworld program: select the core 0 and select *Run --> Load --> Load program*. Browse and select the .out program you compiled in step 6.

9. Press run (the green triangle). You should see the following output:



Notes:

- The number of cores available available to an OpenMP program is determined by the configuration of the OpenMP run-time using *OpenMP.setNumProcessors*. As an example, you can change *OpenMP.setNumProcessors* to a lower value and try running the Hello World again and see the number of print out change.

OMP Integration for Advanced Users

If you are already familiar with OpenMP and TI BIOS MCSDK software, then please see OpenMP Integration in existing applications for more information.

Multi-core Application Image Creation

The standard TI compiler and linker create 'single' *.out files which can be loaded independently and run synchronously on the various cores through CCS or bootloaders. This can be cumbersome when attempting to load a multicore application through CCS and requires additional support infrastructure to boot the complete application.

Packaged with the MCSDK is a collection of tools, called Multi-core Application Deployment (MAD) utilities, that allows a user to create a single loadable/bootable multicore application image from one or more standard *.out files generated by the compiler and linker. The generated multicore image can be loaded and run using CCS. In addition, the IBL provided as part of the MCSDK supports loading of MAD generated multicore application images hence provides a complete infrastructure for booting multicore applications.

MAD is a collection of utilities intended to support a broad range of multicore use cases. More details can be found here in the MAD Utils User Guide [73].

See also:

An example of an MCSDK application that uses MAD is the Image Processing Demo Guide.

Booting and Flash

Boot Overview

The MCSDK includes a Tools Package which provides POST, boot loader and boot utilities for use with the TI EVMs and are intended to serve as example/reference for customers.

The MCSDK tools package is located in the C:\Program Files\Texas Instruments\mcsdk_2_00_00_xx\tools directory and includes:

- **POST**: Power on Self Test application.
- **IBL**: 1st stage and 2nd stage Bootloader for booting an application from the NOR/NAND flash or Ethernet over I2C EEPROM.
- **MAD**: Multicore application deployment tool to support multicore booting.
- **Boot Examples**: Example projects demonstrating the booting of an user application using the boot loader.
- **Writer Utilities**: Utilities to program an application image to flash or EEPROM.
- **Other Utilities**: Utilities to do file format conversion that are required by the boot examples.

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. The POST application resides on the EEPROM of the EVM, therefore the size of the image has to be less than 64 KB.

POST will 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

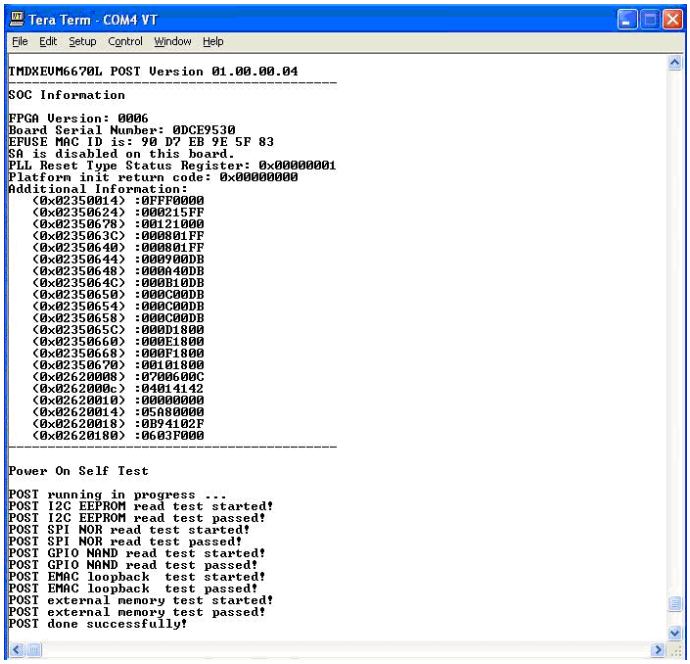
Note: POST is not intended to perform functional tests of the DSP.

At power on, the DSP starts execution with bootrom which transfers execution to the POST boot program from EEPROM using the I2C slave bus address as 0x50. 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, remain ON for approximately 10 sec, 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
EMAC loopback failed	on	blink	blink	blink
Other failures	blink	blink	blink	blink

Note: POST should only be programmed to EEPROM I2C bus address 0x50 (please refer to C:\Program Files\Texas Instruments\mcsdk_2_00_00_xx\tools\post\docs\README.txt on how to build POST and program POST to EEPROM), to execute the POST you must ensure the boot DIP switches for your platform are properly configured to boot from I2C master mode, bus address 0x50 (please refer to the C667x EVM technical reference manual and C667x device data sheet for the boot mode configurations). The POST will put board information and test result on the UART console.



Intermediate Boot Loader (IBL) and Examples

Below is the table showing the boot modes supported by the C66x EVMs:

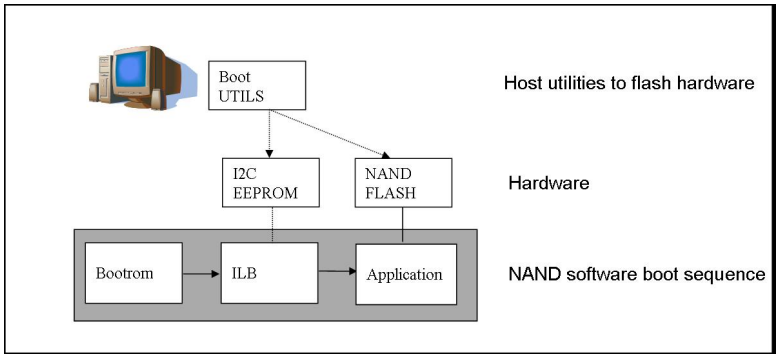
Boot Mode	TMDSEVM6678	TMDSEVM6670	TMDSEVM6618	TMDXEVM6657
NOR boot via IBL over I2C[1]	Yes	Yes	Yes	Yes
NAND boot via IBL over I2C[1]	Yes	Yes	Yes	Yes
TFTP boot via IBL over I2C[1]	Yes	Yes	Yes	Yes
I2C POST boot[2]	Yes	Yes	Yes	Yes
Ethernet boot	Yes	Yes	Yes	Yes
SRIO boot	Yes	Yes	Yes	Yes
PCIe boot	Yes	Yes	Yes	Yes

Note:

1. Support boot over I2C bus address 0x51
2. Support POST boot over I2C bus address 0x50
3. Only ELF and BBLOB images are supported for booting
4. IBL is using the first 128KB L2 local memory, any application booting from IBL should NOT use the first 128KB L2 memory, OR should only use the first 128KB L2 memory for uninitialized data section

NAND Boot

NAND boot is a multi-stage process which is designed to boot an application from NAND flash after reset. Figure below illustrates the elements of the NAND boot process.



Nandboot.jpg

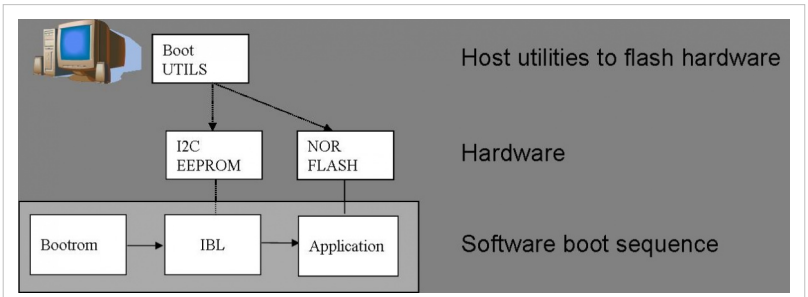
On reset the DSP starts execution with the bootrom which transfers execution to the secondary bootloader from EEPROM using the I2C slave bus address 0x51. The secondary bootloader loads the application program from NAND flash then transfers control to the application. To execute the NAND bootloader you must ensure the DIP switches for your platform are properly configured for I2C Master Boot and address 0x51, AND the boot parameter index dip switch should be set to 2 or 3.

NAND boot supports multiple images booting[1]. Depending on the boot parameter index dip switch, maximum 2 boot images can be supported. By default NAND boot only supports a BBLOB image format, if the customer wants to boot an ELF image, the IBL configuration table needs to be modified and re-programmed to EEPROM.

Please refer to C:\Program Files\Texas Instruments\mcsdk_2_00_00_xx\tools\boot_loader\examples\i2c\nand\docs\README.txt on how to build an Hello World example application and program it to NAND, and boot the Hello World image from the NAND flash.

NOR Boot

NOR boot is a multi-stage process which is designed to boot an application from NOR flash after reset. Figure below illustrates the elements of the NOR boot process.



Norboot.jpg

On reset the DSP starts execution with the bootrom which transfers execution to the secondary bootloader from EEPROM using the I2C slave address 0x51. The secondary bootloader loads the application program from NOR flash then transfers control to the application. To execute the NOR bootloader you must ensure the DIP switches for your platform are properly configured for I2C Master Boot and address 0x51, AND the boot parameter index switch should be set to 0 or 1.

NOR boot supports multiple images booting[1]. Depending on the boot parameter index dip switch, maximum 2 boot images can be supported.

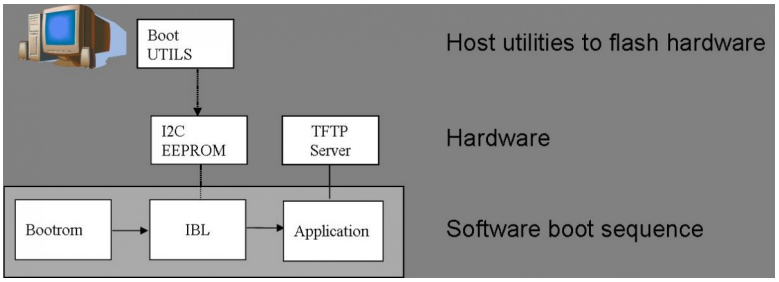
Please refer to C:\Program Files\Texas Instruments\mcsdk_2_00_00_xx\tools\boot_loader\examples\i2c\nor\docs\README.txt on how to build an Hello World example application and program it to NOR, and boot the Hello World image from the NOR flash.

Note:

1. Not supported in Beta-1 release

TFTP Boot

EMAC boot is a multi-stage process which is designed to boot an application from TFTP server after reset. Figure below illustrates the elements of the EMAC boot process.



Emacboot.jpg

On reset the DSP starts execution with the bootrom which transfers execution to the secondary bootloader from EEPROM using the I2C slave address 0x51. The secondary bootloader loads the application program from a remote TFTP server then transfers control to the application. To execute the EMAC bootloader you must ensure the DIP switches for your platform are properly configured for I2C Master Boot and address 0x51, AND the boot parameter index switch should be set to 4. By default EMAC boot only supports a BBLOB image format, if the customer wants to boot an ELF image, the IBL configuration table needs to be modified and re-programmed to EEPROM.

Please refer to C:\Program Files\Texas Instruments\mcsdk_2_00_00_xx\tools\boot_loader\examples\i2c\emac\docs\README.txt on how to build an Hello World example application and boot the Hello World image from a remote TFTP server.

Note:

Please refer to the boot mode dip switch settings for different boot mode on TMDSEVM6678L_EVM [82], TMDSEVM6670L_EVM [83], and TMDSEVM6657L_EVM [84] that IBL supports.

Note:

IBL is flashed into I2C EEPROM bus address 0x51. IBL provides a workaround for the PLL lockup issue (please refer to C6678 errata document, February 2011, advisory 8 for details on the PLL lockup issue). For ROM boot modes (EMAC,SRIO,PCIe,Hyperlink etc) and I2C boot mode with bus address 0x50, DSP will initially boot from I2C EEPROM bus address 0x51 which does the PLL reset workaround, updates the DEVSTAT for appropriate values based on the DIP switch settings (SW3 through SW6 settings) and then re enters the ROM to accomplish the desired boot mode. Please note that the re entry is done for all boot modes except for PCIe boot mode and I2C boot mode with bus address 0x51.

Below are the steps done in the IBL:

1. FPGA samples the bootmode pins
2. FPGA forces the DSP to boot via I2C bus address 0x51
3. PLL is initialized correctly by the IBL on the I2C.
4. IBL reads the sampled bootmode from an FPGA register.
5. IBL checks the bootmode, if it is not I2C boot or it is I2C boot but with bus address 0x50, IBL writes bootmode into the DEVSTAT register
6. IBL then checks if the bootmode is PCIE boot or not. If it is, it executes some PCIE workaround to configure the PCIE registers (mainly to accept spread spectrum clock) and stays inside IBL waiting for PCIe boot.
7. If it is not PCIE boot mode, IBL writes the Boot ROM entry address into the DSP Program Counter, DSP executes the desired internal ROM boot mode or boot from I2C bus address 0x50 as normal.

Updating the IBL Ethernet Configurations

As of MCSDK 2.0.5.17, there are two ways to update the IBL ethernet configurations for ethernet boot.

Using CCS

Please follow the steps as mentioned under section IBL [85]and follow steps 10 through 14. Please note that the i2cConfig.gel file can be modified via a text editor before loading and running the script in CCS. Please note that this gel file contains configuration settings for multiple devices and multiple boot modes.

Using iblConfig Utility Program

The second way to update the IBL ethernet configurations is to use iblConfig.out. This utility program is located under mcsdk_2_00_xx_xx\tools\boot_loader\ibl\src\util\iblConfig\build. In command line, use the "make" program with the given Makefile to generate iblConfig.out and input.txt. Please be sure to fill in the parameters for input.txt before running iblConfig.out; below is an example of input.txt:

```
file_name = ibl.bin
device = 6
offset = 0x500
ethBoot-doBootp = TRUE
ethBoot-bootFormat = ibl_BOOT_FORMAT_ELF
ethBoot-ipAddr = 192.168.1.3
ethBoot-serverIp = 192.168.1.2
ethBoot-gatewayIp = 192.168.1.1
ethBoot-netmask = 255.255.255.0
ethBoot-fileName =
```

The first 3 parameters must be filled in for iblConfig.out to work:

- file_name refers to the IBL binary file to update. This file must be in the same directory as iblConfig.out.
- device refers to the device being used. Please enter **6 for C6678, 7 for C6670, and 8 for C6657**.
- offset refers to an offset space in the IBL. The value is 0x500 for C6678, C6670, and C6657

The ethernet parameters (the entries beginning with ethBoot) refer to specific ethernet configurations. If they are not specified, they will be defaulted to the values in the mcsdk_2_00_xx_xx\tools\boot_loader\ibl\src\util\iblConfig\src\device.h file. In the example above, the ethernet boot file name will be defaulted to c6678-le.bin when iblConfig.out is run.

After running iblConfig.out and updating the IBL binary, you must flash the modified IBL binary to your EVM. You can do this as part of program_evm (refer to section Using Program Evm [86]) or you can flash it individually using eepromwriter (refer to section IBL [85]).

Note: If you updated the IBL with iblConfig and flashed it with eepromwriter, you should **NOT** use i2cparam_0x51_c667#_le_0x500.out and iblConfig.gel - this would overwrite the changes you made to the IBL.

Flash and Flash Utilities

The following boot utilities for loading code into the EEPROM, NOR and NAND are provided as part of the Tools Package with the MCSDK. All source code is provided along with documentation so that customers can port to other environments as necessary or to make modifications and enhancements.

- **romparse**: Utility which converts either the IBL or POST out files into an image format that can be writtent to the EEPROM using the EEPROM writer utility. This utility is specific to Microsoft Windows and generates an image format that MUST be loaded into CCS memory. Romparse utility is located under C:\Program Files\Texas Instruments\mcsdk_2_00_00_xx\tools\boot_loader\ibl\src\util\romparse directory.
- **i2cConfig**: Utility for writing the IBL boot parameter configuration tables to the I2C EEPROM. The configuration table configures the IBL to boot the image from NOR, NAND or EMAC based on the boot priority. This utility executes on the EVM using CCS and JTAG. i2cConfig utility is located under C:\Program Files\Texas Instruments\mcsdk_2_00_00_xx\tools\boot_loader\ibl\src\util\i2cConfig directory.
- **EEPROM Writer**: Utility for writing to the EEPROM. This utility executes on the EVM using CCS and JTAG and it is located under C:\Program Files\Texas Instruments\mcsdk_2_00_00_xx\tools\writer\eeprom\evmc6678l\bin directory.

- **NOR Writer**: Utility for writing to the NOR flash. This utility executes on the EVM using CCS and JTAG and it is located under C:\Program Files\Texas Instruments\mcsdk_2_00_00_xx\tools\writer\nor\evmc6678l\bin directory.
- **NAND Writer**: Utility for writing to the NAND flash. This utility executes on the EVM using CCS and JTAG and it is located under C:\Program Files\Texas Instruments\mcsdk_2_00_00_xx\tools\writer\nand\evmc6678l\bin directory.

Useful Tip

Starting in BIOS-MCSDK 2.1.1, the program_evm utility provides the ability to format the NAND (i.e., permanently erase the entire NAND device). Please refer to program_evm_userguide.pdf (located in the mcsdk_2_00_xx_xx\tools\program_evm\ directory) for more information.

Programming I2C EEPROM (address 0x51) with IBL and boot configuration table[1]

Please refer to C:\Program Files\Texas Instruments\mcsdk_2_00_00_xx\tools\boot_loader\ibl\doc\README.txt on how to build IBL and program IBL and boot parameter configuration table to EEPROM bus address 0x51.

Programming I2C EEPROM (address 0x50) with POST boot[1]

Please refer to C:\Program Files\Texas Instruments\mcsdk_2_00_00_xx\tools\post\docs\README.txt on how to build POST and program POST to EEPROM bus address 0x50.

Flashing NOR FLASH with a user application for NOR boot over I2C

Please refer to C:\Program Files\Texas Instruments\mcsdk_2_00_00_xx\tools\writer\nor\docs\README.txt on how to program a user application to NOR.

Flashing NAND FLASH with a user application for NAND boot over I2C

Please refer to C:\Program Files\Texas Instruments\mcsdk_2_00_00_xx\tools\writer\nand\docs\README.txt on how to program a user application to NAND.

Note:

1. If the customer wants to user their own EEPROM writer to write a raw binary file to the EEPROM, they can use the C:\Program Files\Texas Instruments\mcsdk_2_00_00_xx\tools\boot_loader\ibl\src\util\btoccs\ccs2bin utility to convert the .dat to .bin either with byte swapping or without swapping depending on the data format their EEPROM writer uses.

Technical Support and Product Updates

Technical Support and Forums

For technical discussions and issues, please visit

- **C66x Multicore forum**: http://e2e.ti.com/support/dsp/c6000_multi-core_dsps/f/639.aspx
- **BIOS Embedded Software forum**: http://e2e.ti.com/support/embedded/f/355.aspx
- **Code Composer Studio forum**: http://e2e.ti.com/support/development_tools/code_composer_studio/f/81/t/3131.aspx
- **TI C/C++ Compiler forum**: http://e2e.ti.com/support/development_tools/compiler/f/343/t/34317.aspx
- **Embedded Processors wiki**: http://processors.wiki.ti.com

For local support in China, please visit

- **China Support forum**: http://www.deyisupport.com

Note: When asking for help in the forum you should tag your posts in the Subject with “MCSDK”, the part number (e.g. “C6678”) and additionally the component (e.g. “NDK”).

Useful Tip

You can always get the most recent version of this document on the Texas Instruments Embedded Processors Wiki. See the page titled BIOS MCSDK 2.0 User Guide for the most up to date revision.

Product Updates

There are various ways to receive updates for MCSDK. They are oulined in the following sections.

MCSDK Product Folder

- Visit **Multicore Software Development Kits**: http://focus.ti.com/docs/toolsw/folders/print/bioslinuxmcsdk.html
- Use the **CCS/Eclipse Update Manager**

Note: The EVM comes with disks containing the MCSDK software and CCS. You can start with these or go to the MCSDK software download site listed above to check for the latest updates and version. The BIOS-MCSDK release download will also have pointers to applicable CCS and compiler release versions as well. Please review the release notes and software manifest before downloading and/or installing the software.

Eclipse Update Manager

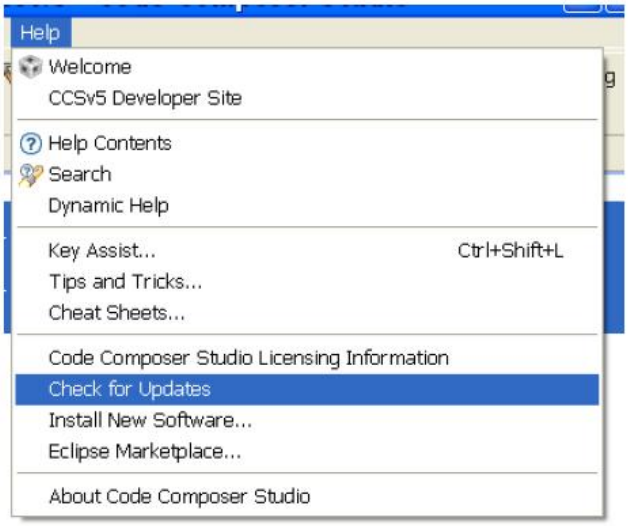
The BIOS MCSDK utilizes Eclipse Update Manager in CCS to detect, download, and install updates in an automated fashion. Eclipse provides various controls for this process -- from manually checking for updates to periodically checking for updates. In the event you can not update via Eclipse using the Eclipse Update Manager, please visit the Texas Instruments software download site for MCSDK: http://focus.ti.com/docs/toolsw/folders/print/bioslinuxmcsdk.html

Note: If you are running CCS on Linux, make sure you have write permissions to CCS folders before doing Eclipse updates. If you installed CCS with root permission, please launch CCS with root permission before updating. Incompatible write permissions will prevent CCS's update plugin to update your files correctly.

Eclipse Update (Automatic)

1. Please make sure the **MCSDK 2x** box is checked in the *available software sites* of CCS, before clicking *check for updates* using the CCS help menu.
2. After CCS re-starts it should recognize MCSDK and can check its update site using the Eclipse Update Manager
3. When the Update Manager connects you will have the option to download the updated release of BIOS MCSDK
4. After downloading, CCS will shut down and run the updated BIOS MCSDK installer
5. After installation, CCS will be re-started and updated BIOS MCSDK content will be installed

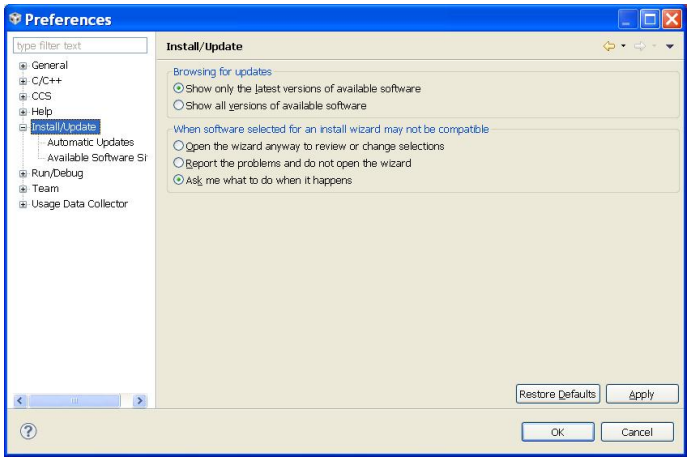
Note: For the Eclipse update to work you must have Eclipse Updates enabled. You may also have it set to check on a periodic basis. If so, you may need to run the Update Manager to get the update immediately from "Help/Check for Update" as shown in the picture below:



Eclipse Update (Manual)

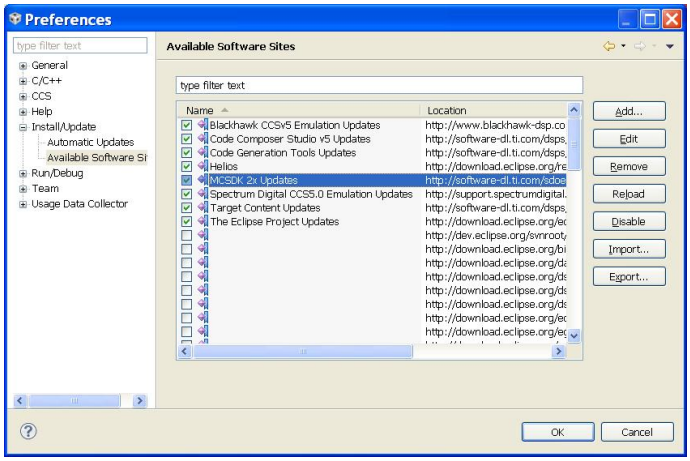
If automatic update does not work, or you wish to just search for an update to MCSDK, do the following, after installing MCSDK.

1. Start CCS, and select Window->Preferences
2. In the left pane select and expand Install/Update, then select Available Software Sites

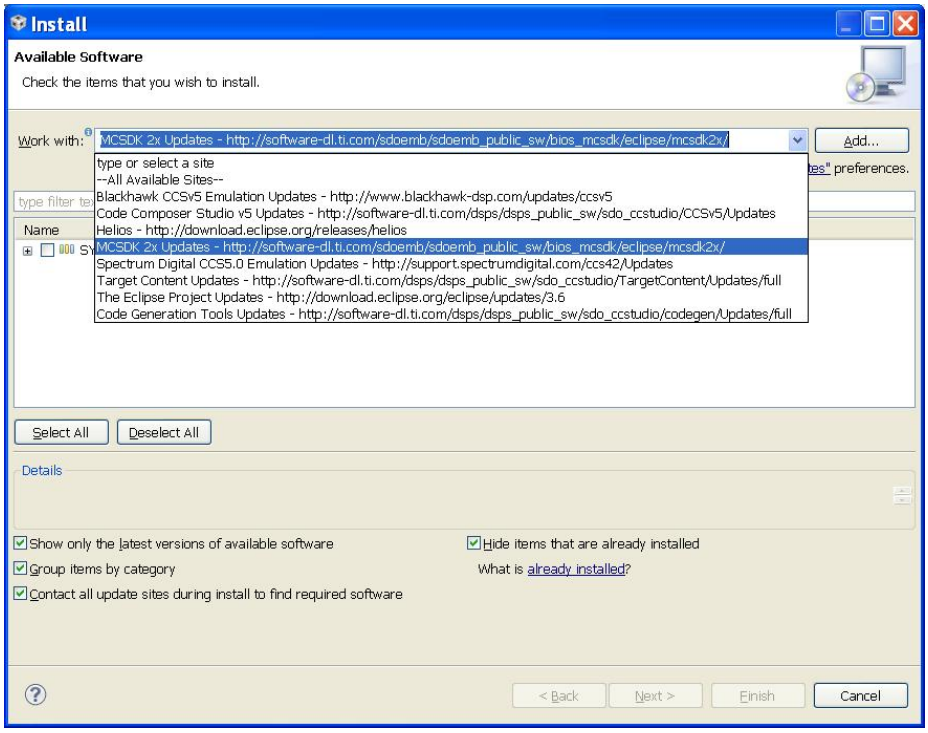


1. It will open a list of avilable software sites

2. In the list find and check URL http://software-dl.ti.com/sdoemb/sdoemb_public_sw/bios_mcsdk/eclipse/mcsdk2x/, the Enabled column should change to Enabled. You can also enter a name for the site but its not required.



1. Select OK to close the window
2. Then select Help->Install New Software… , In the Work with: select the above URL from the drop down menu



1. Check the URL in Name and select Finish
2. The CCS should discover new MCSDK release to install

Frequently Asked Questions

Q: How can I get the EVM back to factory default state?

To flash the EVM to its factory defaults, refer to the *program_evm.pdf* document located in the*factory_images*\ folder from the DVD that came with the EVM. If you have misplaced the DVD, this folder can be downloaded directly from the EVM manufacturer site: TMDSEVM6678 [87],TMDSEVM6670 [88], TMDXEVM6657 (TBD).

After successfully flashing, the EVM will be restored to its original NOR, NAND, and EEPROM binaries.

Q: I have just updated my BIOS MCSDK software, how do I load it to my EVM?

Setup the boot mode to *No Boot* mode by having the dip switches as the following for updating the images to EVM's flash area:

No Boot mode DIP SW Settings

Pin#	1 2 3 4	1 2 3 4	1 2 3 4	1 2 3 4
State	OFF-ON-ON-ON	ON-ON-ON-ON	ON-ON-ON-ON	ON-ON-ON-ON
Switch	SW3	SW4	SW5	SW6

Note: Pin 1 of SW3 is the endian switch - when set to **OFF** put the EVM into *Little Endian* Mode and **ON** puts the EVM into *Big Endian* Mode.

Using Program EVM

As of BIOS MCSDK 2.0.5, there exists a convenient script in the mcsdk_2_00_xx_xx\tools\ directory to update all the images automatically via command line. Follow the steps in program_evm_userguide.pdf (located in the mcsdk_2_00_xx_xx\tools\program_evm\ directory) to flash the new images. The images that are loaded are kept in the .\program_evm\binaries\evm66xxl\ directory; you can substitute any image here.

Useful Tip

To avoid updating CCS from the version that came with the EVM, you can use the program_evm tool found on the EVM DVD and substitute newer images in the binaries\evm66xxl\ directory.

Note: The NAND image for the Linux kernel is not provided with the BIOS MCSDK release.

Note: The DSS script under program_evm directory is using the default ccxml files that are created under CCS 5.0.3; So, for CCS 5.1 please provide the customized ccxml file that user created; Example steps below are for a Windows PC for C6670 EVM XDS560V2 mezzanine card. Please follow similar steps for using the dss script for CCS 5.1 under Linux.

1. program_evm>set PROGRAM_EVM_TARGET_CONFIG_FILE=C:\Documents and Settings\user\CCSTargetConfigurations\evmc6670_CCS51_mezzanine.ccxml (note that there are no double quotes to be given in this path)

2. program_evm>set DSS_SCRIPT_DIR="C:\ti\ccsv5\ccs_base\scripting\bin" (please observe the double quotes in the path here)
3. program_evm>%DSS_SCRIPT_DIR%\dss.bat program_evm.js TMDSEVM6670Le-Le

Q: Can I update the new images individually instead of using Program EVM?

Yes. Setup your EVM to *No Boot* mode as described in the previous question. Then follow the instructions for the EEPROM/NOR/NAND images:

Updating EEPROM Images

The EEPROM images are IBL (intermediate boot loader) and Power On Self Test (POST). The IBL/POST is often updated with MCSDK releases. Follow these instructions to update the EVM to the newer IBL and POST images: IBL is flashed at EEPROM 0x51 address and POST is flashed at EEPROM 0x50 address.

Note: For MCSDK version 2.0.3 and prior, .dat files are provided instead of .bin files. If you are using MCSDK version 2.0.3 or prior, please follow the instructions provided here by replacing .bin with .dat

IBL

1. Copy *i2crom_0x51_c667#_le.bin* from mcsdk_2_00_xx_xx\tools\boot_loader\ibl\src\make\bin to mcsdk_2_00_xx_xx\tools\writer\eeprom\evmc667#l\bin. Rename this copied file to *app.bin*.
2. Open eepromwriter_input.txt in mcsdk_2_00_xx_xx\tools\writer\eeprom\evmc667#l\bin. Set file_name equal to app.bin and bus_addr equal to 0x51. Make sure start_addr and swap_data are set to 0. Save and close eepromwriter_input.txt.
3. Turn on and connect your EVM. Open CCSv5, load the appropriate Target Configuration, connect to Core 0, and load the corresponding GEL file.
4. Load the EEPROM writer program by going to *Run -> Load Program* and browse for the eeprom writer DSP executable. For e.g, *eepromwriter_evm667#l.out* in the same folder as *app.bin* for C667# EVM.
5. View the memory browser (go to *View -> Memory Browser*). Browse to address 0x0C000000.
   Note: For BIOS-MCSDK 2.0.8 and prior, please use address 0x80000000 instead of 0x0C000000.
6. Right click on the memory window and select Load Memory. Select app.bin (By default, the browse menu only displays .dat files. You will have to change the option TI Data Format (*.dat) to Raw Data Format (*.bin) to find your binary file.)
   Note: If you are loading a .dat file, check the box for the option to "Use the file header information to set the start address and size of the memory block to be loaded." This option will not be available for .bin files.
7. Click "Next".
8. Change the Start Address to 0x0C000000 if it is not already. Leave the swap checkbox unchecked. Click "Finish". Please select 32-bits for Type-Size option in CCS.
   Note: For BIOS-MCSDK 2.0.8 and prior, please use address 0x80000000 instead of 0x0C000000.
9. Run the program. This will program the EEPROM.
   A sample successful eeprom writer output would like as below.

```
[C66xx_0] EEPROM Writer Utility Version 01.00.00.05[C66xx_0] [C66xx_0]
Writing 52264 bytes from DSP memory address 0x0c000000 to EEPROM bus
address 0x0051 starting from device address 0x0000 ... [C66xx_0]
Reading 52264 bytes from EEPROM bus address 0x0051 to DSP memory
address 0x0c010000 starting from device address 0x0000 ... [C66xx_0]
Verifying data read ... [C66xx_0] EEPROM programming completed
successfully
```

10. IBL Configuration needs to be programmed after successfully completing step 9. Go to *Run -> Load Program* and select *i2cparam_0x51_c667#_le_0x500.out* located in the *mcsdk_2_00_xx_xx\tools\boot_loader\ibl\src\make\bin* folder).
11. Load the i2cConfig.gel GEL file, located in the *mcsdk_2_00_xx_xx\tools\boot_loader\ibl\src\make\bin* folder.
12. Run the program. The following message will be printed on the CCS console

```
Run the GEL for the device to be configured, press return to program the I2C.
```

Note: DO NOT PRESS ENTER UNTIL STEP 14.

1. Run the GEL script*"EVM c6678 IBL" -> setConfig_c6678_main*.
2. Now press "Enter" in the CCS console window, and the program will write the boot parameter table to the EEPROM. On success the message "I2c table write complete" will be printed on the CCS console.

POST

1. Copy *post_i2crom.bin* from mcsdk_2_00_xx_xx\tools\post\evmc667#l\bin to mcsdk_2_00_xx_xx\tools\writer\eeprom\evmc667#l\bin.
2. Open eepromwriter_input.txt in mcsdk_2_00_xx_xx\tools\writer\eeprom\evmc667#l\bin. Set file_name equal to post_i2crom.bin and bus_addr equal to 0x50. Make sure start_addr and swap_data are set to 0. Save and close eepromwriter_input.txt.
3. Turn on and connect your EVM. Open CCSv5, load the appropriate Target Configuration, connect to Core 0, and load the corresponding GEL file.
4. Load the EEPROM writer program by going to *Run -> Load Program* and browse for the eeprom writer DSP executable. For e.g, *eepromwriter_evm667#l.out* in the same folder as *post_i2crom.bin* for C667# EVM.
5. View the memory browser (go to *View -> Memory Browser*). Browse to address 0x80000000.
6. Right click on the memory window and select Load Memory. Select post_i2crom.bin (By default, the browse menu only displays .dat files. You will have to change the option TI Data Format (*.dat) to Raw Data Format (*.bin) to find your binary file.) **Note: If you are loading a .dat file, check the box for the option to "Use the file header information to set the start address and size of the memory block to be loaded." This option will not be available for .bin files.**
7. Click "Next".
8. Change the Start Address to 0x80000000 if it is not already. Leave the swap checkbox unchecked. Click "Finish".
9. Run the program. This will program the EEPROM.
   A sample successful eeprom writer output would like as below.

```
[C66xx_0] EEPROM Writer Utility Version 01.00.00.04

[C66xx_0]

[C66xx_0] Writing 49752 bytes from DSP memory address 0x80000000 to EEPROM bus address 0x0051 starting from device address 0x0000 ...

[C66xx_0] Reading 49752 bytes from EEPROM bus address 0x0051 to DSP memory address 0x80010000 starting from device address 0x0000 ...

[C66xx_0] Verifying data read ...

[C66xx_0] EEPROM programming completed successfully
```

Updating NOR/NAND Images

The NOR/NAND writers support reading a binary image directly. Please rename the DSP executable *xxx.out* to app.bin and use the writers to directly write a binary image file to the NAND or NOR. Please refer to writer\nand\docs\README.txt or writer\nor\docs\README.txt for details.

Useful Tip

If booting from NOR Flash on a 6670 EVM is failing the DDR3 test with Bios MCSDK 2.0.2 or earlier, an update to the Intermediate Bootloader is available which will fix it. If you have a more recent version of the BIOS MCSDK, this fix is included in your installation. See the instructions for applying the update here. Once you have updated the files, come back to this page and follow the instructions for updating the IBL EEPROM image

Q: How do I use JTAG with 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)

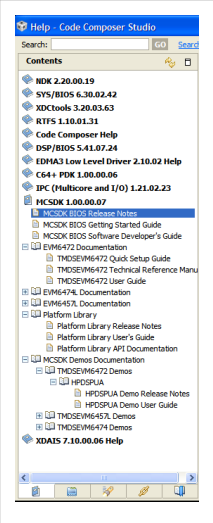
```
[C66xx_0] Verify_Init: Expected 16 entry count for gTxFreeQHnd queue 736, found 62 entries
[C66xx_0] Verify_Init: Expected 0 entry count for gRxQHnd= 704, found 22 entries
[C66xx_0] Verify_Init: Expected 0 entry count for Queue number = 0, found 1 entries
[C66xx_0] Verify_Init: Expected 0 entry count 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 interfearing 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.

Q: Is there a simple way to access documents provided in the release?

Once BIOS-MCSDK is installed in the system, many of the documents can be accessed from *CCS->Help->Help Contents*.



Q: How do I uninstall the BIOS-MCSDK?

The BIOS MCSDK installer installs the un-installer in mcsdk_##_##_##_## directory. The name of the un-installer is *uninstall-bios_mcsdk_2.##.##.##.exe*. It also adds links of the un-installer in **Programs->Texas Instruments->BIOS Multicore SDK** program menu and in Windows *Add and Remove Programs* menu with name **TI BIOS Multicore SDK**. Selecting any one of the links will start the un-installer and remove the BIOS-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*.

Q: Are there example code for 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 [89]
2. The GPIO implementation is provided in file pdk_C66##_1_0_0_##\packages\ti\platform\evmc66##l\platform_lib\src\evmc66x_gpio.c
3. The FPGA implementation is provided in file pdk_C66##_1_0_0_##\packages\ti\platform\evmc66##l\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##l\platform_lib\src\evmc66x_fpga.c

Timer

1. The link SYSBIOS_Training:Timers and Clocks [90] provides detail presentation on configuring timer to get peoridic interrupt
2. An older document on SYSBIOS timer implementation is in DSP/BIOS Timers and Benchmarking Tips [91]

DDR3

1. The DDR3 controller users guide is in DDR3 Memory Controller for KeyStone Devices User's Guide [92]
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\evmc66##l\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 [93]
2. The sample code is in pdk_C66##_1_0_0_##\packages\ti\platform\evmc66##l\platform_lib\src\evmc66x_uart.c

Q: How do I speed up downloading the BIOS-MCSDK installer?

The size of the BIOS-MCSDK installer is large since we want to provide one bundle for all the components. The bad side of this is that if you are manually downloading the BIOS-MCSDK (or CCS) installer, you may run into issues such as download stall or slow download. One simple solution is to run a download manager/accelerator. One open source solution is http://www.freedownloadmanager.org/.

Q: Can I use CCS 5.1 with BIOS MCSDK 2.0?

Starting with BIOS-MCSDK 2.0.5, we support both CCS 5.0.3 and CCS 5.1.0. We are planning on maintaining CCS 5.0.3 support through all the BIOS-MCSDK 2.0.x releases; it will be dropped in the next major release, v2.1. However, the recommended version of CCS is v5.1.0 to benefit from the latest updates of features and bug fixes.

Two notes:

1. Starting from CCS 5.1.0, the MCSA component, which is installed in the CCS directory, is bundled with CCS and installing the version from the BIOS-MCSDK installer into CCS 5.1.0 results in the BIOS-MCSDK installer to crash. The BIOS-MCSDK 2.0.5 installer has MCSA unselected, but previous versions need to be manually unchecked.
2. CCS 5.1 may include a different version of CGT than the version validated with BIOS MCSDK. See the respective release notes to find the actual versions. If there is a mismatch, it is recommended that you use the version that BIOS MCSDK lists as a dependency, and ensure that CCS projects are configured for the appropriate version when building projects.

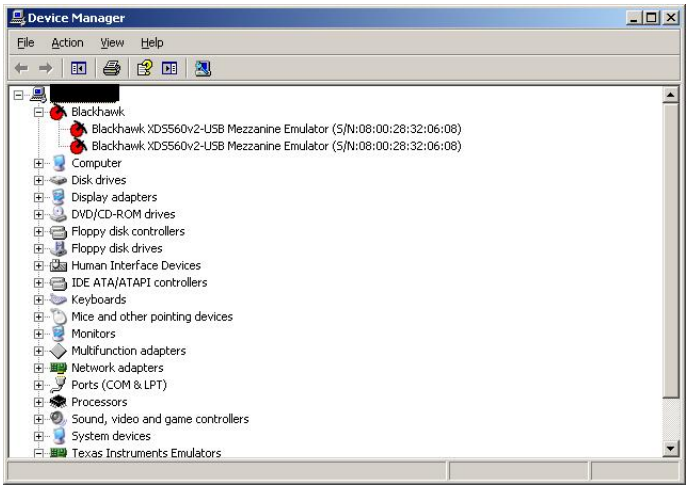
Q: How can I connect and use two emulators of the same type in the same CCS instance?

For the development of some applications involving board to board communications such as SRIO or Hyperlink it may be desirable to simultaneously connect to two boards while running a single instance of Code Composer Studio. The following steps document how to create and use a Target Configuration that allows connect, program load, and debug capabilities on two boards simultaneously. To document these steps the following hardware and software was used.

- 2x c6678 boards with attached Blackhawk XDS560v2-USB Mezzanine Emulator
- Code Composer Studio v5.0.3.00028

Steps to connect to two boards with the same target configuration:

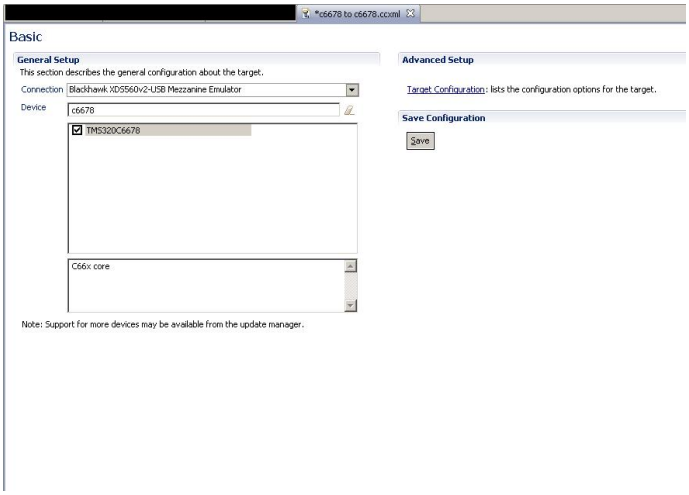
1. Make sure the boards and emulators are powered up and ready to be launched for a debug session. The device manager should show two Blackhawk XDS560v2-USB Mezzanine Emulators under the BlackHawk tab.



2. Start CCS and open the Target Configurations tab, View -> Target Configurations.

3. Right-click within the Target Configurations tab and select "New Target Configuration". Give the target configuration a name and click "Finish".

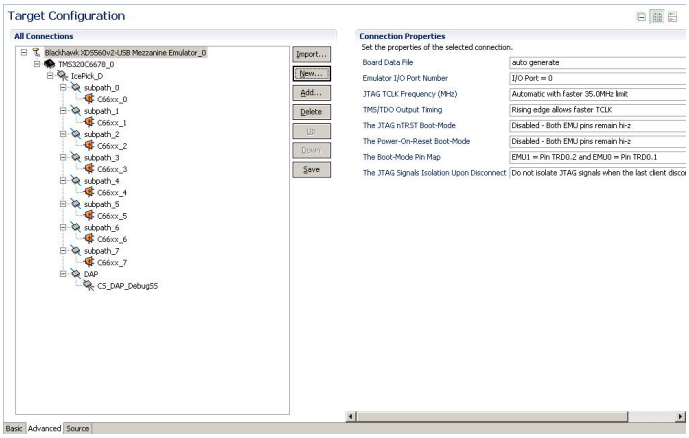
4. In this, and the following step, we'll set up the configuration for the first target. The second target will be added later. In the "Connection" drop down menu select 'Blackhawk XDS560v2-USB Mezzanine Emulator' or the emulator type you're using.



5. In the Device selection window check the TMS320C6678 box, or the box of the processor you're using, and click "Save".

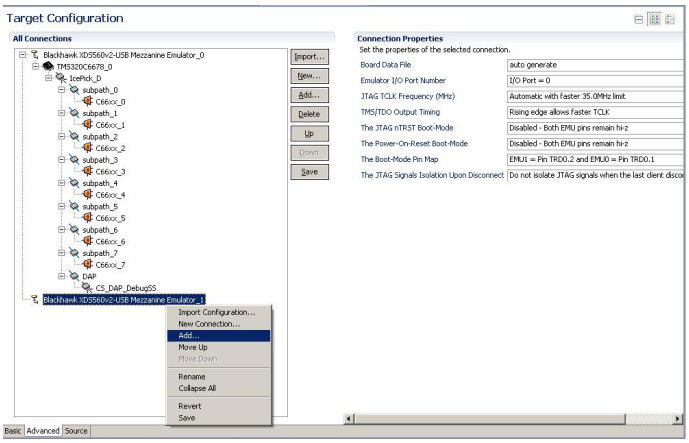
6. In the following steps we'll add the second board to the target configuration. Click the "Advanced" tab at the bottom of the "board_name".ccxml file display.

7. Highlight the first Blackhawk connection and Click "New...".

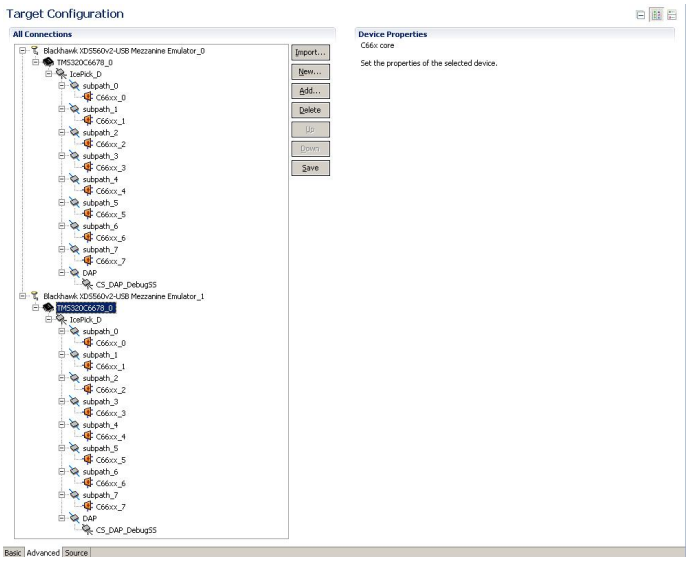


8. Select "Blackhawk XDS560v2-USB Mezzanine Emulator", or the second emulator type you're using, and click "Finish".

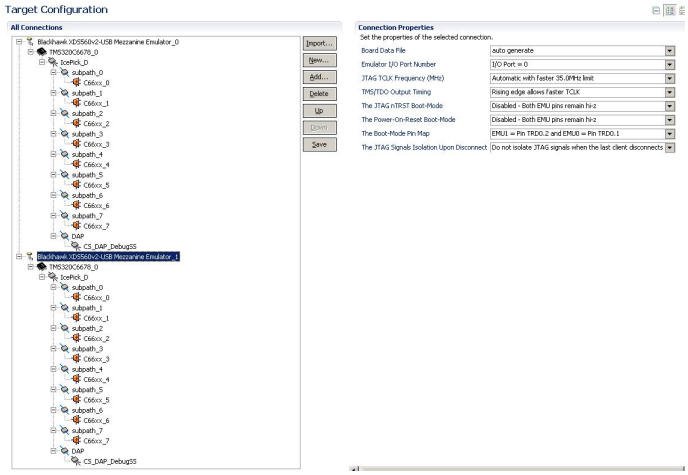
9. Right Click the new Blackhawk connection and select "Add...".



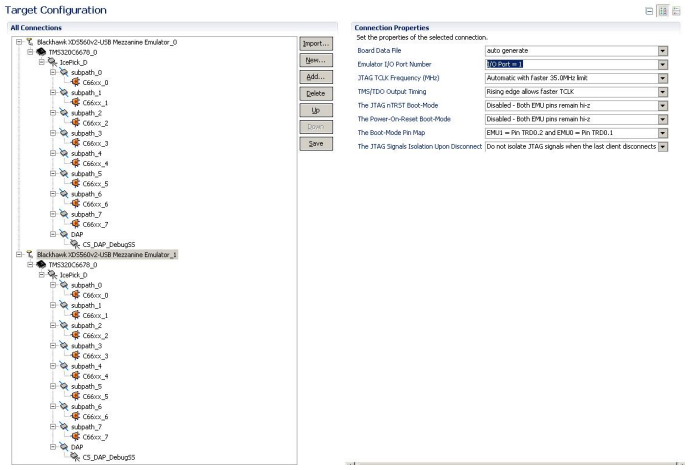
10. In the Device selection tab highlight the TMS320C6678, or the processor you're using, and click "Finish". You're target configuration should now have two Blackhawk emulators each with a c6678 device.



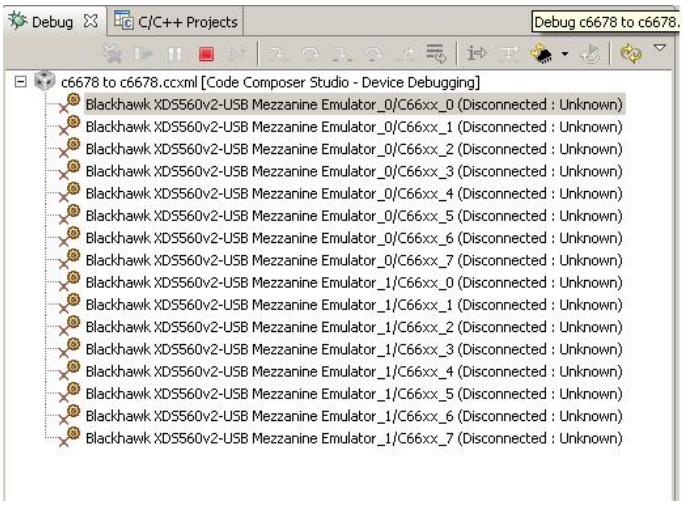
11. Once again, highlight the second Blackhawk Emulator so that the "Connection Properties" show.



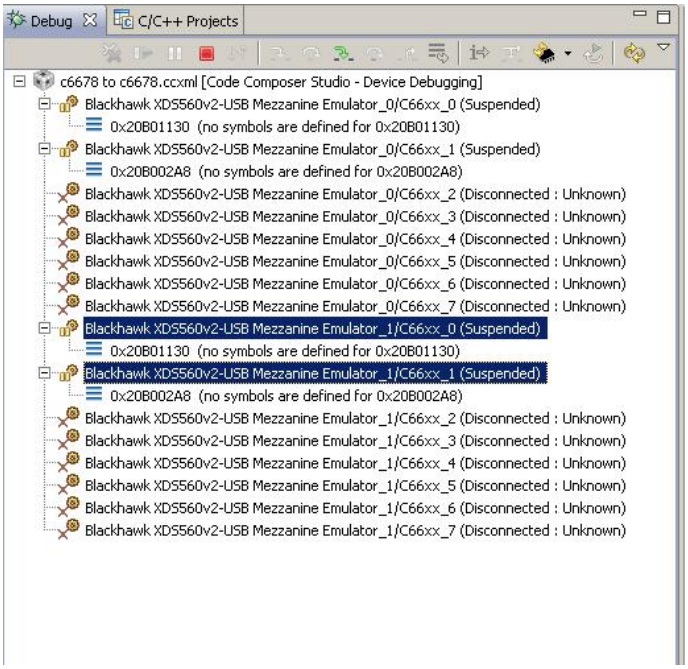
12. Under the Emulator I/O Port Number drop down menu change the setting to "I/O Port = 1" and then click "Save".



13. Start the new target configuration by right-clicking the target configuration in the "Target Configuration" tab and selecting "Launch Selected Configuration". When the launch completes you'll see sixteen cores, for two c6678 boards, in the Debug tab.



14. Connect to the desired cores.



Q: How do I get the latest GEL files for these EVMs?

The GEL files for supported EVMs are provided separately from the MCSDK. If you use CCS 5.1, use the Eclipse Update Manager to check for new updates and follow installation instructions if there is an update. If you use CCS 5.0, or have any problems with using the Eclipse Update Manager in CCS 5.1, you can manually download the GEL updates. See the MCSDK download page listed above for details.

Q: How do I change SoC speed on my EVM?

The SoC speed for the EVM can be changed by setting appropriate PLL multiplier and Divider values. Please refer to the device data sheet for details on setting the Multiplier and Divider values. The Gel file from the emupack also has sample multiplier and divider values for a given SoC speed.

Please refer to section 2.5.3 section of the TMS320C6678 [94] data sheet for the sample multiplier and divider values.

Please refer to section 2.4.3 section of the TMS320C6670 [95] data sheet for the sample multiplier and divider values.

This can be changed in

- platform library (If platform library is used to program the PLL settings)
  - please update multiplier and divider values in ***platform_init()*** function, located under *pdk_C667#_1_0_0_##\packages\ti\platform\evmc667#l\platform_lib\src\platform.c* file. Please rebuild platform library after this change.
- GEL file (If GEL files are used to program the PLL)
  - please update **PLL1_M** and **PLL1_D** values in evmc667#l.gel file, located under *\ccsv5\ccs_base\emulation\boards\evmc667#l\gel* file. Please reload the gel file after this change.
- IBL (If IBL is used for PLL settings, e.g., for i2c boot modes)
  - please update the *ibl.pllConfig[ibl_MAIN_PLL].prediv* variable for the divider and *ibl.pllConfig[ibl_MAIN_PLL].mult* variable for multiplier values in ***c667#_ibl_config()*** function located under *mcsdk_2_00_##_##\tools\boot_loader\ibl\src\util\iblconfig\src\device.c* file. Please rebuild ibl after this change.

References

[1] http://www.ti.com/product/tms320c6657
[2] http://www.ti.com/tool/tmdxevm6657
[3] http://www.ti.com/product/tms320c6670
[4] http://www.ti.com/product/tms320tci6618
[5] http://www.ti.com/tool/tmdsevm6670
[6] http://www.ti.com/product/tms320c6678
[7] http://www.ti.com/product/tms320tci6608
[8] http://www.ti.com/tool/tmdsevm6678
[9] http://focus.ti.com/docs/training/catalog/events/event.jhtml?sku=OLT110048
[10] http://learningmedia.ti.com/public/hpmp/KeyStone/01_MCSDK_Intro_Mandarin/Index.html
[11] http://processors.wiki.ti.com/index.php/Keystone_Device_Architecture
[12] http://focus.ti.com/docs/training/catalog/events/event.jhtml?sku=OLT110027
[13] http://processors.wiki.ti.com/index.php/SYS/BIOS_Online_Training
[14] http://processors.wiki.ti.com/index.php/SYS/BIOS_1.5-DAY_Workshop
[15] http://processors.wiki.ti.com/index.php/Multicore_System_Analyzer_Tutorials
[16] http://www.ti.com/lit/wp/spry168a/spry168a.pdf
[17] http://focus.ti.com/general/docs/video/Portal.tsp?lang=en&entryid=0_xitw1jig
[18] http://processors.wiki.ti.com/index.php/CCSv5_Getting_Started_Guide
[19] http://processors.wiki.ti.com/index.php/Xds_560
[20] http://processors.wiki.ti.com/index.php/XDS100
[21] http://focus.ti.com/lit/ug/spru187t/spru187t.pdf
[22] http://focus.ti.com/lit/ug/spru186v/spru186v.pdf

[23] http://processors.wiki.ti.com/index.php/MCSA
[24] http://rtsc.eclipse.org/docs-tip/Demo_of_the_RTSC_Platform_Wizard_in_CCSv4
[25] http://rtsc.eclipse.org/docs-tip/Runtime_Object_Viewer
[26] http://focus.ti.com/general/docs/video/Portal.tsp?entryid=0_55svdeqr&lang=en
[27] http://www.advantech.com/Support/TI-EVM/6670le_sd.aspx
[28] http://www.advantech.com/Support/TI-EVM/6678le_sd.aspx
[29] http://www.ti.com/lit/ds/symlink/tms320c6657.pdf
[30] http://focus.ti.com/lit/ds/symlink/tms320c6670.pdf
[31] http://focus.ti.com/lit/ds/symlink/tms320c6678.pdf
[32] http://www.ti.com/lit/gpn/tms320tci6618
[33] http://software-dl.ti.com/sdoemb/sdoemb_public_sw/salld/
[34] http://www.linux-c6x.org/wiki/index.php/Main_Page
[35] http://focus.ti.com/docs/toolsw/folders/print/telecomlib.html
[36] http://www.ti.com/tool/c66xcodecs
[37] http://www.ti.com/tool/s2meddus
[38] http://processors.wiki.ti.com/index.php/Software_libraries
[39] http://www.ti.com/tool/demovideo-multicore
[40] http://www.eclipse.org/rtsc/
[41] http://processors.wiki.ti.com/index.php/CSL
[42] http://www.opensource.org/licenses/bsd-license.php
[43] http://processors.wiki.ti.com/index.php/Programming_the_EDMA3_using_the_Low-Level_Driver_%28LLD%29
[44] http://www.ti.com/lit/ug/sprugr9d/sprugr9d.pdf
[45] http://processors.wiki.ti.com/index.php/BIOS_MCSDK_2.0_User_Guide#Related_Software
[46] http://www.ti.com/lit/sprugy6
[47] http://www.ti.com/lit/ug/sprugs4/sprugs4.pdf
[48] http://www.ti.com/lit/sprugw1
[49] http://www.ti.com/lit/sprugs6a
[50] http://www.ti.com/lit/ug/sprugv7b/sprugv7b.pdf
[51] http://www.ti.com/lit/sprugy4
[52] http://www.ti.com/lit/sprugw8
[53] http://www.ti.com/lit/sprugz1
[54] http://www.ti.com/lit/sprugs0
[55] http://www.ti.com/lit/sprugs1
[56] http://www.ti.com/lit/ug/sprugs2c/sprugs2c.pdf
[57] http://processors.wiki.ti.com/
[58] http://software-dl.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/bios/index.html
[59] http://software-dl.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/ipc/index.html
[60] http://focus.ti.com/docs/toolsw/folders/print/bioslinuxmcsdk.html
[61] http://www-s.ti.com/sc/techlit/spru523.pdf
[62] http://www-s.ti.com/sc/techlit/spru524.pdf
[63] http://www-s.ti.com/sc/techlit/sprufp2.pdf
[64] http://processors.wiki.ti.com/index.php/Network_Developers_Kit_FAQ
[65] http://processors.wiki.ti.com/index.php/Rebuilding_the_NDK_Core
[66] http://software-dl.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/ndk/index.html
[67] http://www.gnu.org/licenses/gcc-exception.html
[68] http://e2e.ti.com/support/embedded/f/355.aspx
[69] http://software-dl.ti.com/sdoemb/sdoemb_public_sw/dsplib/latest/index_FDS.html
[70] http://software-dl.ti.com/sdoemb/sdoemb_public_sw/imglib/latest/index_FDS.html
[71] http://focus.ti.com/docs/toolsw/folders/print/mathlib.html
[72] http://linux-c6x.org/wiki/index.php/IBL_version_1.0.0.11
[73] http://processors.wiki.ti.com/index.php/MAD_Utils_User_Guide
[74] http://processors.wiki.ti.com/index.php/Multicore_System_Analyzer
[75] http://rtsc.eclipse.org/docs-tip/Main_Page
[76] http://www.criticalblue.com
[77] http://www.criticalblue.com/prism/ti/
[78] http://processors.wiki.ti.com/index.php/MCSDK_Image_Processing_Demonstration_Guide#Image_Processing_Demo_Analysis_with_Prism
[79] http://www.polycoresoftware.com
[80] http://www.polycoresoftware.com/products.php

[81] http://processors.wiki.ti.com/index.php/BIOS_MCSDK_2.0_Getting_Started_Guide
[82] http://processors.wiki.ti.com/index.php/TMDXEVM6678L_EVM_Hardware_Setup#Boot_Mode_Dip_Switch_Settings
[83] http://processors.wiki.ti.com/index.php/TMDXEVM6670L_EVM_Hardware_Setup#Boot_Mode_Dip_Switch_Settings
[84] http://processors.wiki.ti.com/index.php/TMDXEVM6657L_EVM_Hardware_Setup#Boot_Mode_Dip_Switch_Settings
[85] http://processors.wiki.ti.com/index.php/BIOS_MCSDK_2.0_User_Guide#IBL
[86] http://processors.wiki.ti.com/index.php/BIOS_MCSDK_2.0_User_Guide#Using_Program_EVM
[87] http://wfcache.advantech.com/support/TMDXEVM6678L_Factory_Images.zip
[88] http://wfcache.advantech.com/support/6670/TMDXEVM6670L_Factory_images.zip
[89] http://www.ti.com/litv/pdf/sprugv1
[90] http://processors.wiki.ti.com/index.php/SYS/BIOS_Training:_Timers_and_Clocks
[91] http://focus.ti.com/lit/an/spra829/spra829.pdf
[92] http://www.ti.com/litv/pdf/sprugv8b
[93] http://www.ti.com/litv/pdf/sprugp1
[94] http://www.ti.com/lit/ds/sprs691c/sprs691c.pdf
[95] http://www.ti.com/lit/ds/sprs689d/sprs689d.pdf

MCSDK HUA Guide

Overview

The High-Performance DSP Utility Application (HUA) is the Out-of-Box (OOB) demonstration for the Multicore Software Development Kit (MCSDK) which demonstrates, through illustrative code and web pages, how you can interface your own DSP application to the various TI MCSDK software elements including SYS/BIOS, Network Development Kit (NDK), the Chip Support Library (CSL), and Platform Library. The purpose of the demonstration is to illustrates the integration of key components in MCSDK and provide a multicore software development framework on an evaluation module (EVM.)

This document covers various aspects of the demonstration, including a discussion on the requirements, software design, instructions to build and run the application, and troubleshooting steps. Currently, only SYS/BIOS is supported as the embedded OS.

Access to the demo application is done through a PC web browser. The welcome web page provides a starting point with links to more information on TI multicore DSPs and support forums.

In addition at the top of the web page are a number of tabs which implement basic functionality including:

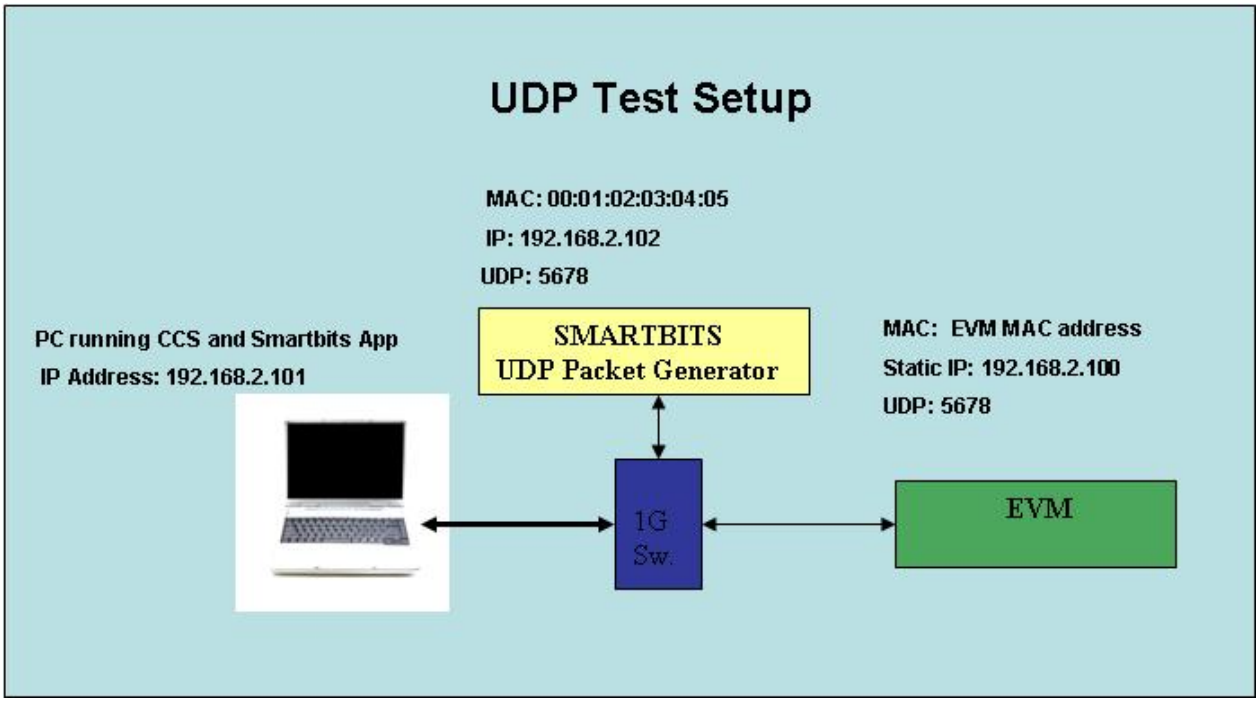
- **Information**: Generates a page displaying a collection of information related to the platform and its operation such as system up time, platform settings, device type, number of cores, core speeds, software element versions, and network stack information. All this information is collected using API calls to the various MCSDK software elements.
- **Statistics**: Generates a page reporting standard Ethernet statistics from the networking stack.
- **Task List**: generates a page reporting the current active SYS/BIOS tasks on the device including information such as Task Priority, Task State, Stack Size Allocated, and Stack Size Used for each task.
- **Benchmarks**: Takes the user to a web page with a list of supported benchmarks that a the user can run on the platform.
- **Diagnostics**: Takes the user to a web page that allows the user to execute a range of platform diagnostics tests.
- **Flash**: Takes the user to a web page that display flash hardware information and allows the user to read and write the flash on the platform.
- **EEPROM**: Takes the user to a web page that allows the user to read the EEPROM.

Benchmarks

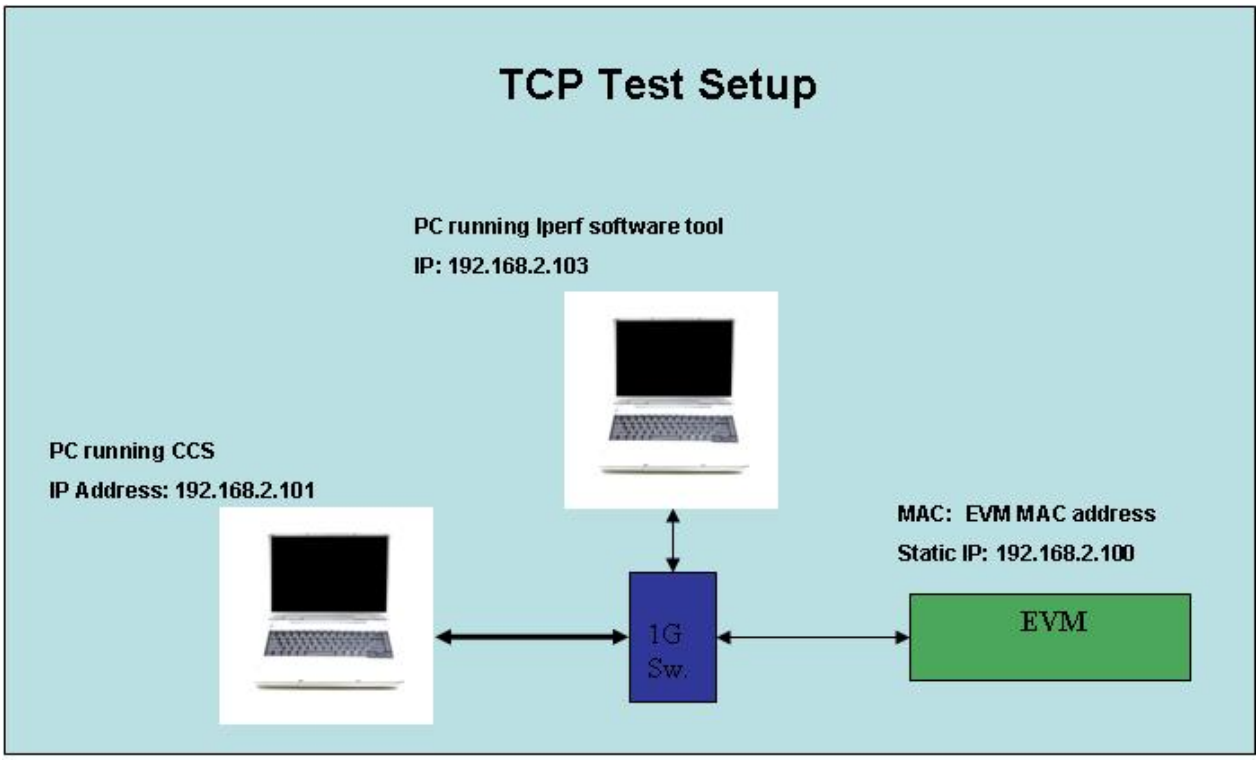
The *Benchmark* tab takes the user to a page with a list of supported benchmarks that the user can run on the platform. Currently there are two supported benchmarks for release 2.0.

- **Network Throughput Test/Benchmark**: Allows the user to configure and execute a network throughput test between the PC and the EVM. The user can configure direction (Transmit or Receive); Protocol (UDP, TCP); and amount of data to send. Upon completion test results will be displayed, e.g., data loss, test time, and effective throughput.
- **Network Loopback Test/Benchmark**: Allows the user to configure and execute UDP and TCP network loopback throughput test between a test equipment and the EVM. A UDP packet generator (Smartbits) is required to measure UDP throughput, and IPERF software test tool is also required to run and measure TCP throughput.

UDP test setup is depicted below:



TCP test setup is depicted below:



An example of IPERF command used for testing: iperf -c 192.168.2.100 -i 10 -t 600 -w 64K -d

Diagnostics

The *Diagnostics t*ab allows the user to execute a range of platform diagnostics tests. These are diagnostics provided as part of the platform library.

The diagnostic tests supported include:

- **External RAM Test**: Tests a defined section of external RAM through a process of writing and reading back a series of patterns. The diagnostic will display a PASS/FAIL indication after the test executes.
- **Processor Internal Memory Test**: Test the internal memory associated with a user specified processing core through a process of writing and reading back a series of patterns. The diagnostic will display a PASS/FAIL indication after the test executes. This diagnostic can only be executed for processing cores other than 0 and is not applicable to single core devices.
- **Flash LED**: Allows the user to turn ON and OFF specified platform LEDs
- **UART Test**: Allows the user to send a text message to the UART port. For this test the user must have a PC connected to the UART port on the platform.

Flash

The *Flash* page displays information related to the Flash hardware and allows the user to read and write to the flash. For reading, the user can specify a block to read from flash and then page through the data. For writing the user can either write an arbitrary file (binary blob) or a bootable image. The bootable image option allows you to write an image the EEPROM boot loader can load and execute.

EEPROM

The *EEPROM* page allows a user to read the EEPROM paging through the data in 1K blocks.

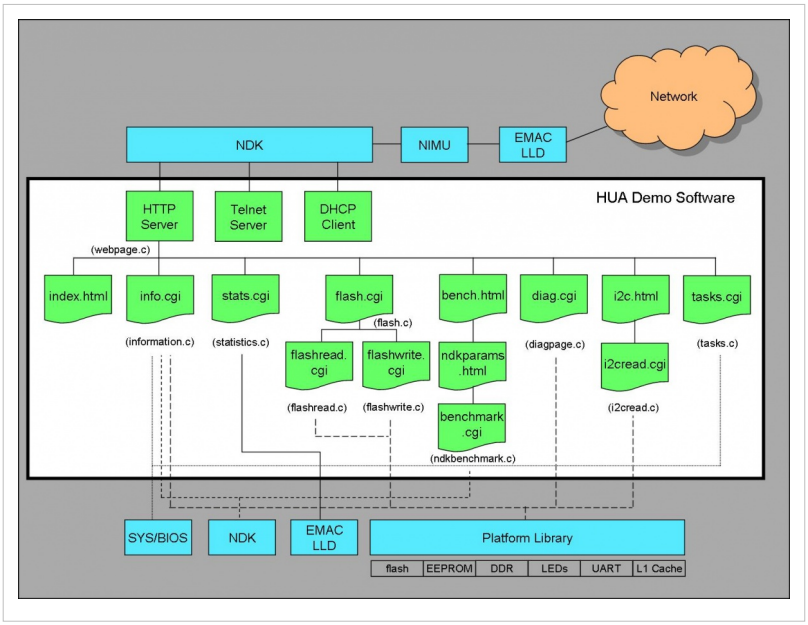
Requirements

The following materials are required to run this demonstration:

- TMS320C6x low cost EVMs [Check MCSDK release notes for supported platforms]
- Power cable
- Ethernet cable
- Windows PC with CCSv5

Software Design

The high level software architecture for the HUA is shown below.



As can be seen in the diagram, the Utility provides an HTTP and Telnet Server. These servers use standard socket interfaces to the IP stack (NDK) which in turn interfaces to the Ethernet through the NIMU and EMAC Driver components.

The HTTP server serves pages that allow either various operations to be performed on the EVM (e.g., diagnostics) or provide information (e.g., statistics). The web pages are either dynamically created through a CGI-BIN interface (.cgi) or are static pages that are served directly back (.html).

Tasks

As this is an embedded system, it uses SYS/BIOS to provide tasking and OS primitives such as semaphores, timers and so forth. The main thread is the task *hpdspuaStart*. This task will configure the IP stack and bring the system up into a free running state.

Note: The main for the Utility simply start SYS/BIOS. SYS/BIOS in turn will run the task.

Platform Initialization

Platform initialization is performed by a function within the utility called *EVM_init()*. This function is configured to be called by SYS/BIOS before it starts up. Platform initialization configures DDR, the I2C bus, clocking and all other items that are platform dependent.

Build Instructions

Please follow the steps below to re-compile the libraries (These steps assume you have installed the MCSDK and all the dependent packages).

- Open CCS->Import Existing... tab and import project from C:\Program Files\Texas Instruments\mcsdk_2_00_00_xx\demos\hua.
- It should import two projects hua_evmc6678l and hua_evmc6670l.
- Right click on each project->Properties to open up the properties window.
- Goto CCS Build->RTSC and check if in other repository have link to <MCSDK INSTALL DIR> (the actual directory).
- The project should build fine.

Run Instructions

The pre-compiled libraries are provided as a part of MCSDK release.

Please follow the procedures below to load images using CCS and run the demo.

Please refer to the hardware setup guide for further setup details.

- Connect the board to a Ethernet hub or PC using Ethernet cable.
- The demo runs in Static IP mode if User Switch 1 is OFF else if it is ON then it runs in DHCP mode. See the Hardware Setup section for the location of User Switch 1.
- If it is configured in static IP mode, the board will come up with IP address 192.168.2.100, GW IP address 192.168.2.101 and subnet mask 255.255.254.0
- If it is configures in DHCP mode, it would send out DHCP request to get the IP address from a DHCP server in the network.
- Connect the debugger and power on the board.
- In CCS window, launch the target configuration file for the board.
- It should open debug perspective and open debug window with all the cores.
- Connect to core 0 and load demos\hua\evmc66xxl\Debug\hua_evmc66xxl.out.
- Run HUA on core 0, in the CIO console window, the board should print IP address information (for eg: Network Added: If-1:192.168.2.100)
- Open a web browser in the PC connected to the HUB or the board.
- Enter the IP address of the board, it should open up the HUA demo web page.
- Please follow the instructions in the web page to run the demo.

Note: If you want to run the demo in static IP address mode, make sure the host PC is in same subnet or can reach the gateway. A sample setup configuration is shown below.

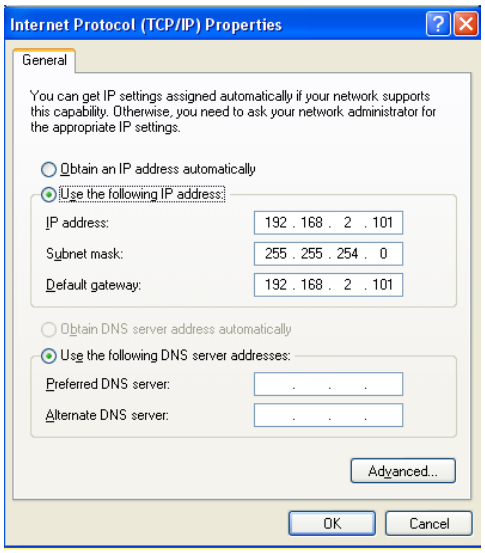
In Windows environment

Set up TCP/IP configuration of 'Wired Network Connection' as shown in Wired Network Connection in Windows.

In Linux environment

Run following command to set the static IP address for the current login session on a typical Linux setup.

```
sudo ifconfig eth0 192.168.2.101 netmask 255.255.254.0
```



Troubleshooting

Data verification error when using CCS to load HUA

Check if the EVM GEL is properly configured and run when CCS connects the target. The GEL will initialize the PLL and external memory so that HUA can be loaded and run from external memory.

The CIO console window does not show the IP address

Check if the EVM is connected to a network with DHCP server running.

The CIO console window shows the static IP address, but can not ping it

Check if the EVM is connected to a static network, and the PC that is used to ping the EVM has the same subnet address as the EVM does.

MCSDK Image Processing Demonstration Guide

Multicore Software Development Kit
Image Processing Demonstration
User's Guide

Last updated: //

Overview

The Image Processing Demonstration illustrates the integration of key components in the Multicore Software Development Kit (MCSDK) on Texas Instruments (TI) multicore DSPs and System-on-Chips. The purpose of the demonstration is to provide a multicore software development framework on an evaluation module (EVM).

This document covers various aspects of the demonstration application, including a discussion on the requirements, software design, instructions to build and run the application, and troubleshooting steps. Currently, only SYS/BIOS is supported as the embedded OS.

This application shows implementation of an image processing system using a simple multicore framework. This application will run TI image processing kernels (a.k.a, *imagelib*) on multiple cores to do image processing (eg: edge detection, etc) on an input image.

There are three different versions of this demonstration that are included in the MCSDK. However, not all three versions are available for all platforms.

- *Serial Code*: This version uses file i/o to read and write image file. It can run on the simulator or an EVM target platform. The primery objective of this version of the demo is to run Prism and other software tools on the code to analyze the basic image processing algorithm.
- *IPC Based*: The IPC based demo uses SYS/BIOS IPC component to communicate between cores to perform an image processing task parallel. See below for details.
- *OpenMP Based*: (Not available for C6657) This version of the demo uses OpenMP to run the image processing algorithm on multiple cores.

Note: The current implementation of this demonstration is not optimized. It should be viewed as the initial implementation of the BIOS MCSDK software eco-system for creating an image processing functionality. Further analysis and optimization of the demonstration are under progress.

Note: There are three versions of the demo provided in the release. The IPC based version runs on multiple cores and shows explicit IPC programming framework. The serial version of the demo runs on the simulator. The OpenMP version uses OpenMP to communicate between cores to process the input images. Unless explicitly specified, the IPC based version is assumed in this document.

Requirements

The following materials are required to run this demonstration:

- TMS320C6x low cost EVMs [Check Image Processing release notes for supported platforms]
- Power cable
- Ethernet cable
- Windows PC with CCSv5

Software Design

The following block diagram shows the framework used to implement the image processing application:

The following diagram shows the software pipeline for this application: .

More about processing algorithms

The application will use imagelib APIs for its core image processing needs.

Following steps are performed for edge detection

- Split input image into multiple overlapping slices
- If it is a RGB image, separate out the Luma component (Y) for processing (See YCbCr [1] for further details)
- Run Sobel operator [2] (IMG_sobel_3x3_8) to get the gradient image of each slices
- Run the thresholding operation (IMG_thr_le2min_8) on the slices to get the edges
- Combine the slices to get the final output

Framework for multicore

The current framework for multicore is either IPC Message Queue based framework or OpenMP. Following are the overall steps (the master and threads will be run on 1 or more cores)

- The master thread will preprocess the input image (described in User interface section) to make a gray scale or luma image
- The master thread signal each slave thread to start processing and wait for processing complete signal from all slave threads
- The slave threads run edge detection function (described above) to generate output edge image of the slice
- Then the slave threads signal master thread indicating the processing completed
- Once master thread receives completion signal from all threads it proceeds with further user interface processing (described in User interface section)

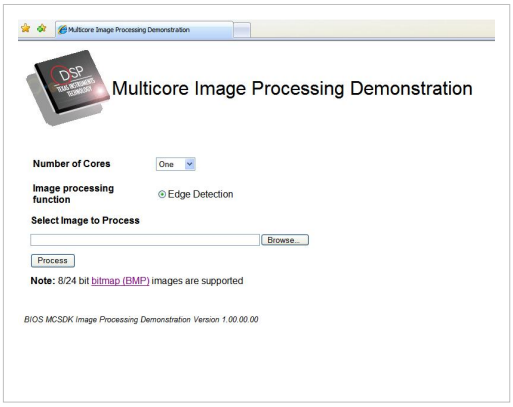
Profiling of the algorithm

- The profiling information live processing time will be presented at the end of the processing cycle
- Core image processing algorithms is instrumented using UIA for analysis and visualization using MCSA (Multicore System Analyzer)

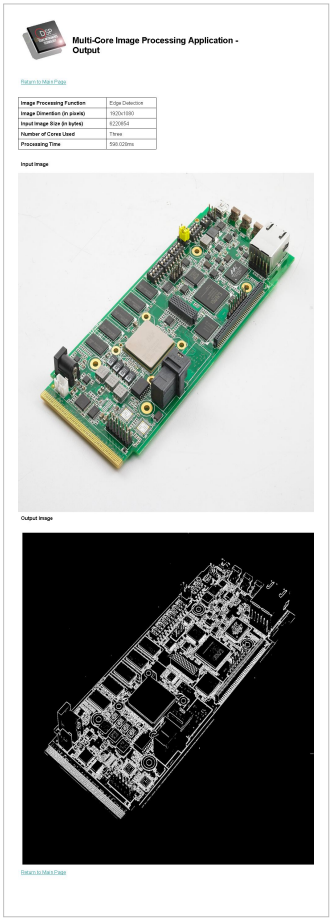
User interface

The user input image will be a BMP image. The image will be transferred to external memory using NDK (http). Following are the stapes describing application user interface and their interaction

- At the time of bootup the board will bring configure IP stack with static/dynamic IP address and start a HTTP server
- The board will print the IP address in CCS console
- The user will use the IP address to open the index/input page (see link Sample Input Page)
- The application will support BMP image format
- The master thread will extract the RGB values from BMP image
- Then the master thread will initiate the image processing (as discussed above) and wait for its completion
- Once the processing completes, it will create output BMP image
- The master thread will put input/output images in the output page (see link Sample Output Page)



Software outline of the OpenMP demo



- The main task is called by OpenMP in core 0, spawns a task to initialize NDK, then gets/sets IP address and starts a web service to transfer user inputs and images. The main task then creates a mailbox and waits on a message post to the mailbox.
- The NDK calls a callback function to the application to retrieve the image data from user. The function reads the image and posts a message with the image information to the main task. Then it waits on a mailbox for a message post from main task.
- After receiving the message, the main task extracts RGB, splits the image into slices and processes each slices in different cores. A code snippet of this processing is provided below.

```
pragma omp parallel for shared(p_slice, number_of_slices, ret_val) private(i) for (i = 0; i < number_of_slices; i++ ) {
```

```
 DEBUG_PRINT(printf("Processing slice # %d\n", i);)
   /* Process a slice */
   process_rgb (&p_slice[i]);
 if (p_slice[i].flag != 0) {
   printf("mc_process_bmp: Error in processing slice %d\n", i);
```

```
pragma omp atomic
```

```
   ret_val = -1;
 }
 DEBUG_PRINT(printf("Processed slice # %d\n", i);)
```

```
}
```

if (ret_val == -1) {

```
 goto close_n_exit;
```

}

- After processing is complete, the main task creates the output image and sends the image information to the callback task using a message post. Then it waits on the mailbox again.
- The NDK callback task wakes up with the message post and sends the result image to the user.

Different Versions of Demo

Software Directory Structure Overview

The Image Processing Demonstration is present at <MCSDK INSTALL DIR>\demos\image_processing

- <MCSDK INSTALL DIR>\demos\image_processing\ipc\common directory has common slave thread functions which runs on all cores for the IPC based demo; The image processing function runs in this slave thread context
- <MCSDK INSTALL DIR>\demos\image_processing\ipc\master directory has main thread, which uses NDK to transfer images and IPC to communicate to other cores to process the images
- <MCSDK INSTALL DIR>\demos\image_processing\ipc\slave directory has the initialization function for all slave cores
- <MCSDK INSTALL DIR>\demos\image_processing\openmp\src directory has the main thread, which uses NDK to transfer images and OpenMP to communicate between cores to process the image
- <MCSDK INSTALL DIR>\demos\image_processing\ipc\evmc66##l\[master|slave] directories have the master and slave CCS project files for the IPC based demo
- <MCSDK INSTALL DIR>\demos\image_processing\openmp\evm66##l directory has the CCS project files for the OpenMP based demo
- <MCSDK INSTALL DIR>\demos\image_processing\######\evmc66##l\platform directory has the target configuration for the project
- <MCSDK INSTALL DIR>\demos\image_processing\serial directory has the serial version of the implementation
- <MCSDK INSTALL DIR>\demos\image_processing\utils directory has utilities used on the demo, like MAD config files
- <MCSDK INSTALL DIR>\demos\image_processing\images directory has sample BMP images

Serial Code

Run Instructions for Serial based demo application

The pre-compiled libraries are provided as a part of MCSDK release.

Please follow the procedures below to load images using CCS and run the demo.

Please refer the hardware setup guide for further the setup details.

- Connect the board to a Ethernet hub or PC using Ethernet cable.
- The demo runs in Static IP mode if User Switch 1 (SW9, position 2) is OFF else if it is ON then it runs DHCP mode. See the Hardware Setup section for the location of User Switch 1.
- If it is configured in static IP mode, the board will come up with IP address 192.168.2.100, GW IP address 192.168.2.101 and subnet mask 255.255.254.0
- If it is configured in DHCP mode, it would send out DHCP request to get the IP address from a DHCP server in the network.
- There is one image to be loaded on core 0. The image name is <MCSDK INSTALL DIR>\demos\image_processing\serial\Debug\image_processing_serial_simc6678.out.
- Connect the debugger and power on the board.
- It should open debug perspective and open debug window.
- Connect to only core 0, if the board is in no-boot mode make sure gel file is run to initialize ddr.
- Load image_processing_seria_simc6678.out to core 0.
- Run the corre 0, in the CIO window, the board should pint IP address information (eg: Network Added: If-1:192.168.2.100)
- Open a web browser in the PC connected to the HUB or the board.

- Enter the IP address of the board, it should open up the image processing demo web page.
- Please follow the instructions in the web page to run the demo.
- Note that sample BMP images are provided in <MCSDK INSTALL DIR>\demos\image_processing\images

Build Instructions for Serial based demo application

Please follow the steps below to re-compile the Serial based demo image (These steps assume you have installed the MCSDK and all dependent packages).

- Open CCS->Import Existing... tab and import project from <MCSDK INSTALL DIR>\demos\image_processing\serial.
- It should import image_processing_serial_simc6678 project.
- The project should build fine for Release and Debug profile.

IPC-Based

Run Instructions for IPC based demo application

The pre-compiled libraries are provided as a part of MCSDK release.

Please follow the procedures below to load images using CCS and run the demo.

Please refer the hardware setup guide for further the setup details.

- Connect the board to a Ethernet hub or PC using Ethernet cable.
- The demo runs in Static IP mode if User Switch 1 (SW9, position 2) is OFF else if it is ON then it runs in DHCP mode. See the Hardware Setup section for the location of User Switch 1.
- If it is configured in static IP mode, the board will come up with IP address 192.168.2.100, GW IP address 192.168.2.101 and subnet mask 255.255.254.0
- If it is configures in DHCP mode, it would send out DHCP request to get the IP address from a DHCP server in the network.
- There are two images to be loaded to master (core 0) and other cores. The core 0 to be loaded with <MCSDK INSTALL DIR>\demos\image_processing\ipc\evmc66##l\master\Debug\image_processing_evmc66##l_master.out image and other cores (referred as slave cores) to be loaded with <MCSDK INSTALL DIR>\demos\image_processing\ipc\evmc66##l\slave\Debug\image_processing_evmc66##l_slave.out image.
- Connect the debugger and power on the board.
- In CCS window, launch the target configuration file for the board.
- It should open debug perspective and open debug window with all the cores.
- Connect to all the cores and load image_processing_evmc66##l_master.out to core 0 and image_processing_evmc66##l_slave.out to all other cores.
- Run all the cores, in the CIO console window, the board should print IP address information (for eg: Network Added: If-1:192.168.2.100)
- Open a web browser in the PC connected to the HUB or the board.
- Enter the IP address of the board, it should open up the image processing demo web page.
- Please follow the instructions in the web page to run the demo.
- Note that, sample BMP images are provided in <MCSDK INSTALL DIR>\demos\image_processing\images

Note: If you want to run the demo in static IP address mode, make sure the host PC is in same subnet or can reach the gateway. A sample setup configuration is shown below.

In Windows environment

Set up TCP/IP configuration of 'Wired Network Connection' as shown in Wired Network Connection in Windows.

In Linux environment

Run following command to set the static IP address for the current login session on a typical Linux setup.

```
sudo ifconfig eth0 192.168.2.101 netmask 255.255.254.0
```

Build Instructions for IPC based demo application

Please follow the steps below to re-compile the IPC based demo image (These steps assume you have installed the MCSDK and all the dependent packages).

- Open CCS->Import Existing... tab and import project from <MCSDK INSTALL DIR>\demos\image_processing\ipc.
- It should import two projects image_processing_evmc66##l_master and image_processing_evmc66##l_slave.
- Right click on each project->Properties to open up the properties window.
- Goto CCS Build->RTSC and check if in other repository have link to <MCSDK INSTALL DIR> (the actual directory).
- If IMGLIB C66x is unchecked, please select 3.0.1.0 to check it.
- The RTSC platform should have demos.image_processing.evmc66##l.platform.
- The project should build fine.

OpenMP-Based

Run Instructions for OpenMP based demo application

The pre-compiled libraries are provided as a part of MCSDK release.

Please follow the procedures below to load images using CCS and run the demo.

Please refer the hardware setup guide for further the setup details.

- Connect the board to a Ethernet hub or PC using Ethernet cable.
- The demo runs in Static IP mode if User Switch 1 is OFF else if it is ON then it runs in DHCP mode. See the Hardware Setup section for the location of User Switch 1.
- If it is configured in static IP mode, the board will come up with IP address 192.168.2.100, GW IP address 192.168.2.101 and subnet mask 255.255.254.0
- If it is configures in DHCP mode, it would send out DHCP request to get the IP address from a DHCP server in the network.
- There ONE image to be loaded to core 0. The image name is <MCSDK INSTALL DIR>\demos\image_processing\openmp\evmc66##l\Release\image_processing_openmp_evmc66##l.out.
- Connect the debugger and power on the board.
- In CCS window, launch the target configuration file for the board.
- It should open debug perspective and open debug window.
- Connect to only core 0, if the board is in no-boot mode make sure gel file is run to initialize ddr.
- Load image_processing_openmp_evmc66##l.out to core 0.
- Run the core 0, in the CIO console window, the board should print IP address information (for eg: Network Added: If-1:192.168.2.100)
- Open a web browser in the PC connected to the HUB or the board.
- Enter the IP address of the board, it should open up the image processing demo web page.
- Please follow the instructions in the web page to run the demo.
- Note that, sample BMP images are provided in <MCSDK INSTALL DIR>\demos\image_processing\images

Build Instructions for OpenMP based demo application

Please follow the steps below to re-compile the OpenMP based demo image (These steps assume you have installed the MCSDK and all the dependent packages).

- Open CCS->Import Existing... tab and import project from <MCSDK INSTALL DIR>\demos\image_processing\openmp\evmc66##l.
- It should import image_processing_openmp_evmc66##l project.
- The project should build fine for Release and Debug profile.

Multicore System Analyzer integration and usage

The System Analyzer provides correlated realtime analysis and visiblity into application running on single or multicore. Analysis and visibility includes Execution Graph, Duration Analysis, Context Aware Profile, Load Analysis and Statistics Analysis. Basic instrumentation using Unified Instrumentation Architecture (UIA) collects data in realtime and transport via Ethernet or JTAG to host where it it decoded, correlated, analyzed and visualized.

System Analyzer is automatically added to CCS5.0 by the MCSDK istaller. CCS5.1 is shipped with the System Analyzer included.

The Image Processing Demo has been instrumented for duration/benchmark and CPU load analysis. Detailed information on running the demo with System Analyzer is provided in System Analyzer and the MCSDK Demo [3] page.

Image Processing Demo Analysis with Prism

This section we will use Prism software [4] to analyze the serial version of image processing demo. The steps below would assume Prism with C66x support is installed in the system and user completed the *Prism Getting Started - Tutorials* provided in the help menu.

Bring up the demo with Prism software

- Bring up the CCS as specified in the Prism documentation
- Edit *macros.ini* file from *<MCSDK INSTALL DIR>\demos\image_processing\serial* directory and change *../../../../imglib_c66x_#_#_#_#* to static path to IMGLIB package
- Open *Import Existing CCS Eclipse Project* and select search directory *<MCSDK INSTALL DIR>\demos\image_processing\serial*
- Check *Copy projects into workspace* and click *Finish*. This will copy the project to the workspace for Prism.
- Clean and re-compile the project
- Open the C6678 simulator, load the image to core0
- Open *Tools->GEL files*, select *Load GEL* and load/open *tisim_traces.gel* from *<CCSv5 INSTALL DIR>\ccs_base_####\simulation_csp_ny\env\ccs\import* directory
- Then select *CPU Register Trace->StartRegTrace* to start the trace, then run the program, wait till it finishes, then select *CPU Register Trace->StopRegTrace* to stop the trace
- Open *Prism->Show Prism Perspective*. It will start Prism Perspective
- Right click on the project and select *Create New PAD File*, it would open the New Prism Analysis Definition window, hit next
- It will open *Architecture Selection* window, select C6671 (single core) template. Then select Finish to open PAD file
- Select *Run->Debug Configurations->prismtrace*, this will convert the simulator generated traces to the traces required by Prism

- The PAD window should have filled in with default trace (PGSI, PGT) file names generated in above step
- Select *Read Trace* to read the trace
- After it read the trace, then select the complete trace from overview window and hit *Load Slice*
- The *Functions* tab will show the functions and their cycle information during the execution
- Observe the *Core 0* scheduling in the schedule window, you can place a marker for this run

What If analysis

The Prism tool allows user to analyze *What If* scenarios for the code

- *What If* the code is run on multiple cores
  - In the function window, right click on *process_rgb* function, select *Force Task* and hit *Apply*. This would make *process_rgb* function simulated as a separate task
  - In the *Architecture* tab, select *C6678 (8 Core)* template, hit *Apply*
  - Observe in *Schedule* tab, the change in execution when selected the *process_rgb* is simulated to run on 8 cores
  - A marker can be placed to compare the improvement
- *What If* the dependencies are removed
  - The *Dependencies* window helps to see and analyze the dependencies (which are preventing the task to be executed in multiple cores simultaneously)
  - Un-check *Serialized* check-boxes against dependency rows and hit *Apply*
  - Add the comparison marker in the *Schedule* tab and check the improvement

The Prism supports more functionality then described in this section. Please see Prism documentation for more information.

Multicore booting using MAD utilities

The detailed information on the Multicore Application Deployment a.k.a MAD utility is provided in MAD user guide [73] page.

This section will provide you the detail instructions on how the tool and boot the demo from flash/ethernet.

Linking and creating bootable application image using MAD utilities

The BIOS MCSDK installation provides MAD tool in *<MCSDK INSTALL DIR>\tools\boot_loader\mad-utils*. This package contains necessary tools to link the application to a single bootable image.

The image processing demo has following updates to create MAD image:

- The master and slave images are linked with *--dynamic* and *--relocatable* options.
- The MAD config files used to link the master and slave programs are provided in *<MCSDK INSTALL DIR>\demos\image_processing\utils\mad\evmc66##l\config-files*. Following are few items to note on the config file.
  - *maptoolCfg_evmc#####.json* has the directory and file name information for the tools
  - *deployment_template_evmc#####.json* has the deployment configuration (it has device name, partition and application information). Following are some more notes on the configuration file.
    - For C66x devices, the physical address is 36 bits and virtual address is 32 bits for external devices, this includes MSMC SRAM and DDR3 memory subsystem.
    - The *secNamePat* element string is a regular expression string.
    - The sections *bss*, *neardata*, *rodata* must be placed in one partition and in the order it is shown here

- The build script *<MCSDK INSTALL DIR>\demos\image_processing\utils\mad\evmc66##l\build_mad_image.bat* can be used to re-create the image

Note: The compilation will split out lots of warning like *Incompatible permissions for partition ...*, it can be ignored for now. This is due to mis-match in partition permissions wrt. the sections placed in the partition

- The bootable image is placed in *<MCSDK INSTALL DIR>\demos\image_processing\utils\mad\evmc66##l\images*

Pre-link bypass MAD image

Please see MAD user guide for more information on pre-link bypassed MAD image. The build script *build_mad_image_prelink_bypass.bat* can be used to build images with this mode.

Booting the application image using IBL

This image can be booted using IBL bootloader.

Following things to be noted on booting the image

- The image type/format is *ibl_BOOT_FORMAT_BBLOB*, so the IBL needs to be configured to boot this format
- The branch address (Branch address after loading) of the image [it is set to *0x9e001040* (or *0x80001040* if you are using BIOS MCSDK v 2.0.4 or prior) in MAL application], is different from default IBL boot address, so the IBL configuration needs to be updated to jump to this address

The following sections will outline the steps to boot the image from Ethernet and NOR using IBL. Please see IBL documentation on the detail information on booting.

Booting from Ethernet (TFTP boot)

- Change IBL configuration: The IBL configuration parameters are provided in a GEL file *<MCSDK INSTALL DIR>\tools\boot_loader\ibl\src\make\bin\i2cConfig.gel*. All the changes needs to be done in the function *setConfig_c66##_main()* of the gel file.
  - The IBL configuration file sets PC IP address 192.168.2.101, mask 255.255.255.0 and board IP address as 192.168.2.100 by default. If these address needs to be changed, open the GEL file, change *ethBoot.ethInfo* parameters in function *setConfig_c66##_main()*
  - Make sure the *ethBoot.bootFormat* is set to *ibl_BOOT_FORMAT_BBLOB*
  - Set the *ethBoot.blob.branchAddress* to *0x9e001040* (or *0x80001040* if you are using BIOS MCSDK v 2.0.4 or prior).
  - Note that the application name defaults to *app.out*

```
menuitem "EVM c66## IBL";
```

hotmenu setConfig_c66##_main() {

```
ibl.iblMagic = ibl_MAGIC_VALUE;
ibl.iblEvmType = ibl_EVM_C66##L;


...
```

```
ibl.bootModes[2].u.ethBoot.doBootp = FALSE;
ibl.bootModes[2].u.ethBoot.useBootpServerIp = TRUE;
ibl.bootModes[2].u.ethBoot.useBootpFileName = TRUE;
ibl.bootModes[2].u.ethBoot.bootFormat = ibl_BOOT_FORMAT_BBLOB;
```

```
SETIP(ibl.bootModes[2].u.ethBoot.ethInfo.ipAddr, 192,168,2,100);
SETIP(ibl.bootModes[2].u.ethBoot.ethInfo.serverIp, 192,168,2,101);
```

```
SETIP(ibl.bootModes[2].u.ethBoot.ethInfo.gatewayIp, 192,168,2,1);
SETIP(ibl.bootModes[2].u.ethBoot.ethInfo.netmask, 255,255,255,0);
```

```
...
```

```
ibl.bootModes[2].u.ethBoot.ethInfo.fileName[0] = 'a';
ibl.bootModes[2].u.ethBoot.ethInfo.fileName[1] = 'p';
ibl.bootModes[2].u.ethBoot.ethInfo.fileName[2] = 'p';
ibl.bootModes[2].u.ethBoot.ethInfo.fileName[3] = '.';
ibl.bootModes[2].u.ethBoot.ethInfo.fileName[4] = 'o';
ibl.bootModes[2].u.ethBoot.ethInfo.fileName[5] = 'u';
ibl.bootModes[2].u.ethBoot.ethInfo.fileName[6] = 't';
ibl.bootModes[2].u.ethBoot.ethInfo.fileName[7] = '\0';
ibl.bootModes[2].u.ethBoot.ethInfo.fileName[8] = '\0';
ibl.bootModes[2].u.ethBoot.ethInfo.fileName[9] = '\0';
ibl.bootModes[2].u.ethBoot.ethInfo.fileName[10] = '\0';
ibl.bootModes[2].u.ethBoot.ethInfo.fileName[11] = '\0';
ibl.bootModes[2].u.ethBoot.ethInfo.fileName[12] = '\0';
ibl.bootModes[2].u.ethBoot.ethInfo.fileName[13] = '\0';
ibl.bootModes[2].u.ethBoot.ethInfo.fileName[14] = '\0';
```

```
ibl.bootModes[2].u.ethBoot.blob.startAddress = 0x9e000000 /*0x80000000 for BIOS MCSDK v2.0.4 or prior*/; /* Load start address */

ibl.bootModes[2].u.ethBoot.blob.sizeBytes = 0x20000000;

ibl.bootModes[2].u.ethBoot.blob.branchAddress = 0x9e001040 /*0x80001040 for BIOS MCSDK v2.0.4 or prior*/; /* Branch address after loading */
```

```
ibl.chkSum = 0;
```

}

- Write IBL configuration:
  - Connect the board using JTAG, power on the board, open CCS, load the target and connect to core 0. Select *Tools->GEL Files* and in the GEL Files window right click and load GEL. Then select and load *<MCSDK INSTALL DIR>\tools\boot_loader\ibl\src\make\bin\i2cConfig.gel*.
  - Load I2C writer *<MCSDK INSTALL DIR>\tools\boot_loader\ibl\src\make\bin\i2cparam_0x51_c66##_le_0x500.out* to Core 0 and run. It will ask to run the GEL in console window. Run the GEL script from *Scripts->EVM c66##->setConfig_c66##_main*.
  - Open the CCS console window and hit enter to complete the I2C write.
- Booting the image:
  - Disconnect the CCS from board, power off the board.
  - Connect ethernet from board to switch/hub/PC and UART cables from board to PC.
  - Make sure your PC have the IP address specified above.
  - Set the board dip switches to boot from ethernet (TFTP boot) as specified in the hardware setup table (TMDXEVM6678L [82] TMDXEVM6670L [83])
  - Copy the demo image *<MCSDK INSTALL DIR>\demos\image_processing\utils\mad\evmc66##\images\mcip-c66##-le.bin* to tftp directory and change its name to *app.out*
  - Start a tftp server and point it to the tftp directory
  - Power on the board. The image will be downloaded using TFTP to the board and the serial port console should print messages from the demo. This will also print the configured IP address of the board

- Use the IP address to open the demo page in a browser and run the demo

Booting from NOR

- Change IBL configuration: The IBL configuration parameters are provided in a GEL file *<MCSDK INSTALL DIR>\tools\boot_loader\ibl\src\make\bin\i2cConfig.gel*. All the changes needs to be done in the function *setConfig_c66##_main()* of the gel file.
  - Make sure the *norBoot.bootFormat* is set to *ibl_BOOT_FORMAT_BBLOB*
  - Set the *norBoot.blob[0][0].branchAddress* to *0x9e001040* (or *0x80001040* if you are using BIOS MCSDK v 2.0.4 or prior)

menuitem "EVM c66## IBL";

hotmenu setConfig_c66##_main() {

```
ibl.iblMagic = ibl_MAGIC_VALUE;
ibl.iblEvmType = ibl_EVM_C66##L;
```

```
...
```

```
ibl.bootModes[0].bootMode = ibl_BOOT_MODE_NOR;
ibl.bootModes[0].priority = ibl_HIGHEST_PRIORITY;
ibl.bootModes[0].port = 0;
```

```
ibl.bootModes[0].u.norBoot.bootFormat = ibl_BOOT_FORMAT_BBLOB;
ibl.bootModes[0].u.norBoot.bootAddress[0][0] = 0; /* Image 0 NOR offset
 byte address in LE mode */
ibl.bootModes[0].u.norBoot.bootAddress[0][1] = 0xA00000; /* Image 1 NOR
 offset byte address in LE mode */
ibl.bootModes[0].u.norBoot.bootAddress[1][0] = 0; /* Image 0 NOR offset
 byte address in BE mode */
ibl.bootModes[0].u.norBoot.bootAddress[1][1] = 0xA00000; /* Image 1 NOR
 offset byte address in BE mode */
ibl.bootModes[0].u.norBoot.interface = ibl_PMEM_IF_SPI;
ibl.bootModes[0].u.norBoot.blob[0][0].startAddress = 0x9e000000
/*0x80000000 for BIOS MCSDK v2.0.4 or prior*/; /* Image 0 load start
address in LE mode */
ibl.bootModes[0].u.norBoot.blob[0][0].sizeBytes = 0xA00000; /* Image 0
size (10 MB) in LE mode */
ibl.bootModes[0].u.norBoot.blob[0][0].branchAddress = 0x9e001040
/*0x80001040 for BIOS MCSDK v2.0.4 or prior*/; /* Image 0 branch
address after loading in LE mode */


...
```

```
ibl.chkSum = 0;
```

}

- Write IBL configuration:
  - Connect the board using JTAG, power on the board, open CCS, load the target and connect to core 0. Select *Tools->GEL Files* and in the GEL Files window right click and load GEL. Then select and load *<MCSDK INSTALL DIR>\tools\boot_loader\ibl\src\make\bin\i2cConfig.gel*.

- Load I2C writer *<MCSDK INSTALL DIR>\tools\boot_loader\ibl\src\make\bin\i2cparam_0x51_c66##_le_0x500.out* to Core 0 and run. It will ask to run the GEL in console window. Run the GEL script from *Scripts->EVM c66##->setConfig_c66##_main*
- Open the CCS console window and hit enter to complete the I2C write

- Write NOR image:
  - Copy application image (*<MCSDK INSTALL DIR>\demos\image_processing\utils\mad\evmc66##l\images\mcip-c66##-le.bin*) to *<MCSDK INSTALL DIR>\tools\writer\nor\evmc66##l\bin\app.bin*
  - Connect the board using JTAG, power on the board, open CCS, load the target and connect to core 0. Make sure the PLL and DDR registers are initialized from the platform GEL (if it is not done automatically, run Global_Default_Setup function from the GEL file). Load image *<MCSDK INSTALL DIR>\tools\writer\nor\evmc66##l\bin\norwriter_evm66##l.out*
  - Open memory window and load the application image (*<MCSDK INSTALL DIR>\demos\image_processing\utils\mad\evmc66##l\images\mcip-c66##-le.bin*) to address *0x80000000*
  - Be sure of your **Type-size** choice 32 bits
  - Hit run for NOR writer to write the image
  - The CCS console will show the write complete message
- Boot from NOR:
  - Disconnect CCS and power off the board
  - Set the board dip switchs to boot from NOR (NOR boot on image 0) as specified in the hardware setup table (TMDXEVM6678L [82] TMDXEVM6670L [83])
  - Connect ethernet cable from board to switch/hub
  - Connect serial cable from board to PC and open a serial port console to view the output
  - Power on the board and the image should be booted from NOR and the console should show bootup messages
  - The demo application will print the IP address in the console
  - Use the IP address to open the demo page in a browser and run the demo

Performance numbers of the demo

The following table compares the performance between OpenMP and explicit IPC based image processing demo applications.

Note: The numbers are based on a **non-DMA** based implementation with **non-optimized RGB to Y** kernel. The L2 cache is set to 256KB.

Note: The results shown were taken during the BIOS-MCSDK 2.1.0 beta, results taken with current component versions may vary.

	Processing time for a ~16MB BMP image (in msec)	
Number of Cores	**OpenMP based demo**	**Explicit IPC based demo**
1	290.906	290.378
2	150.278	149.586
3	101.697	100.75
4	77.147	77.485
5	63.154	63.318
6	54.709	54.663
7	49.144	47.659
8	42.692	42.461

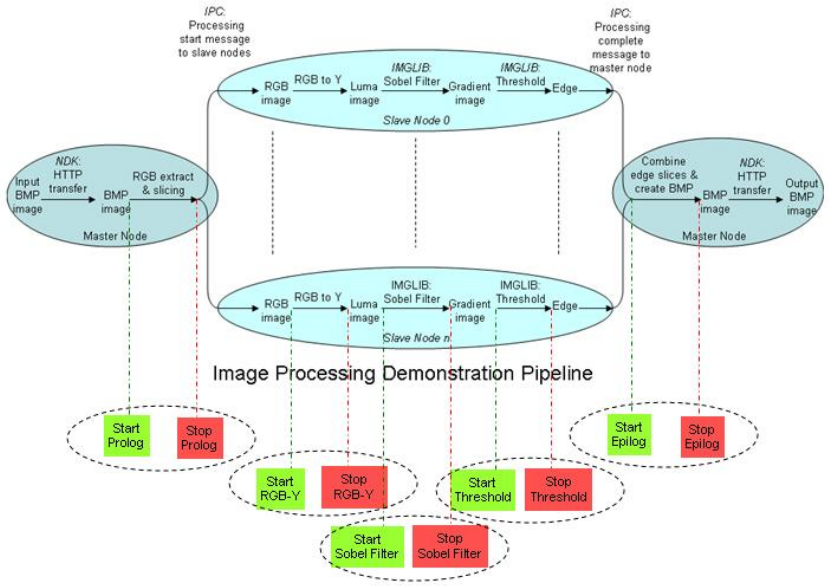
The performance numbers seem to be similar in both cases. This might be due to the nature of the demo application. It spends most of its processing time on actual image processing and sends, at most, 16 IPC messages between cores. So the contribution of the communication delays (IPC vs. OMP/IPC) are very minimal compared to any significant difference in processing times.

References

[1] http://en.wikipedia.org/wiki/Ycbcr
[2] http://en.wikipedia.org/wiki/Sobel_operator
[3] http://processors.wiki.ti.com/index.php/MCSA_and_the_MCSDK_Demo
[4] http://www.criticalblue.com/prism/

MCSA and the MCSDK Demo

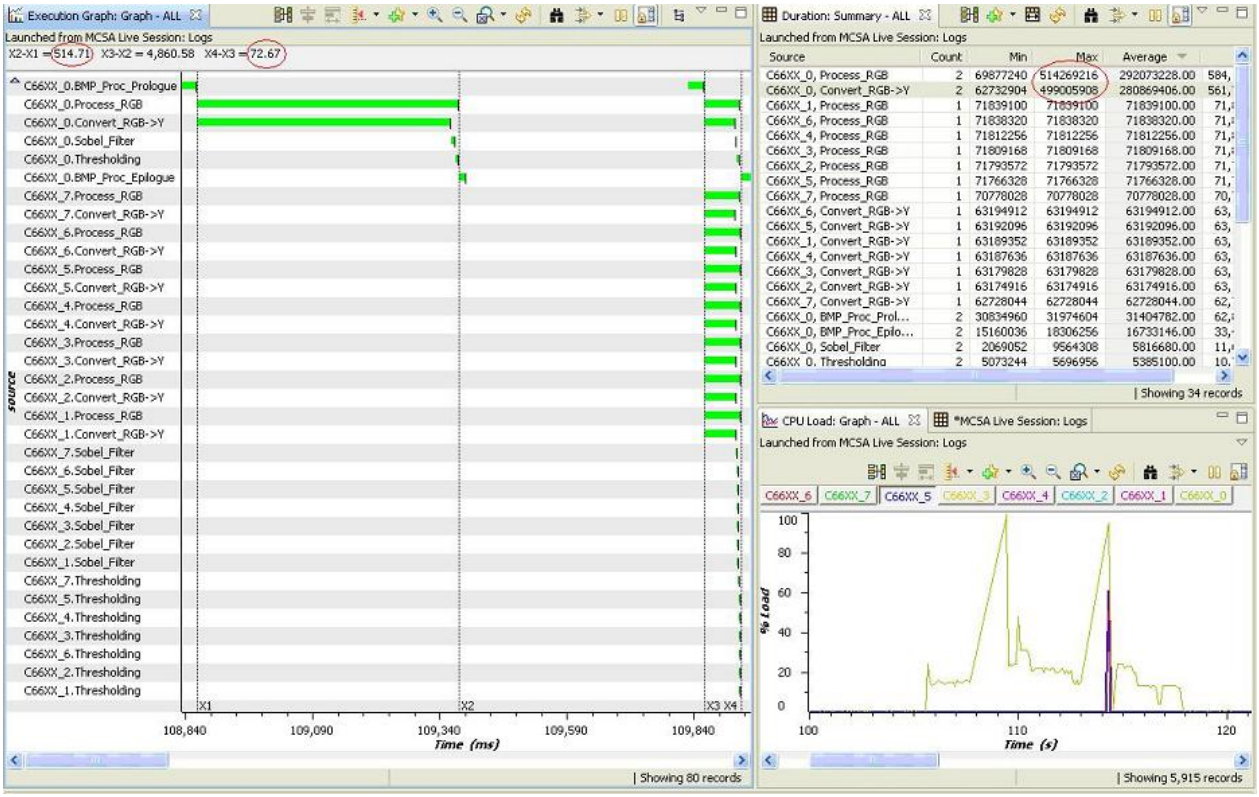
The BIOS MCSDK 2.0 Image Processing Demo [1] has been instrumented to allow users to view the application's real-time behavior and performance using the System Analyzer. Start/Stop event are logged at each processing phase in the application as shown in the image below. SYSBIOS CPU load logging has also been turned on to monitor CPU loading. Other instrumentation can be added or SYSBIOS logging enabled for additional analysis and visibility. Go here for more information on System Analyzer [74].



1 System

Analyzer Execution Graph below shows the demo application execution while processing a 16Mb image on a single core follow by processing the same image on 8 cores. The graph shows the processing stages (as per Start/Stop logs) of the image sequentially executing within the cores and the parallel execution of the image slices across the 8 cores. Execution graph Measurement Markers show that it took 514.71 ms to process the image on singe core but only 72.67 ms on 8 cores, an improvement of about 7 folds. The graph makes it easy to see that most of the processing time is spent in the 'RGB to Y' phase, a possible area for performance improvement. The Duration table benchmark shows the processing time ofeach phase.

Other System Analyzer features can be used to further analyze of the demo execution, e.g. the CPU load graph shows the increase in CPU activity when the image is processed.



Follow these steps to run the System Analyzer and view the real-time dynamics of the MCSDK demo.

- In CCS, launch the evm6678L target.
- In the CCS Debug View, select CPU 0, right-click and select Connect, and then load the master program's .out file.
- Select CPUs 1-7, right click and select Group Core(s).
- Click on the 'Group 1' item in the debug view, right-click and select Connect, then load the slave program's .out file
- Run the program: in the Debug View, select the 'Group 1' and click on the 'run' icon (green triangle). Then select CPU 0 and click on the run icon.
- A System Analyzer view will open once the target has established an IP address. Click on the IP address link in the view to open the demo's web page in a browser.
- In the demo web page select the 'Number of Cores' then 'Select Image to Process'
- Before clicking on the 'Process' button in the web, launch System Analyzer: in the CCS Tools menu, select System Analyzer, and then 'Live'.
- In the configuration options that are displayed, select 'Until stop is requested', and then click OK
- Wait for the 'SA Live Session: Logs' view to open and start displaying data
- In the demo's web page click on the Process button
- After the image has been processed, the web page will display the input image, output image and processing time.
- Allow a few seconds for System Analyzer to retrieve logs then Pause System Analyzer data collection. To pause right click on the 'SA Live Session: Logs' view to open it's context menu and select 'SA Live Session->Pause/Resume'
- To display the System Analyzer Multicore execution graph right click on the 'SA Live Session: Logs' view and select 'Analyze -> Execution Graph'
- In the System Analyzer Execution graph you will see along the vertical axis the names of the various tasks running on each CPU, and a timeline along the horizontal axis
- Zoom out to see the entire execution history by repeatedly clicking on the '-' magnifying glass icon

- Zoom in to see a smaller section of the execution history by selecting the range of interest along the timeline using the left mouse button and then releasing the mouse button.
- To display CPU load right click on the 'SA Live Session: Logs' view and select 'Analyze -> CPU Load'
- To display benchmark info right click on the 'SA Live Session: Logs' view and select 'Analyze -> Duration'

Each of the analysis features above has additional views which can be accessed from 'Windows -> Open Analysis View' or from the 'Open an existing analysis view' button in the main toolbar

References

[1] http://processors.wiki.ti.com/index.php/MCSDK_Image_Processing_Demonstration_Guide

Article Sources and Contributors

BIOS MCSDK 2.0 User Guide *Source*: http://processors.wiki.ti.com/index.php?oldid=121019 *Contributors*: A0187367, A0270985, A0792105, AravindBatni, Charlief, ChrisRing, Csmith, DanRinkes, EricDing, Frankfruth, Hao, Ipang, JackM, Justin32, RajSivarajan, Randyp, Rhillard, Sajeshsaran, Spiceisland, ToanTruong

MCSDK HUA Guide *Source*: http://processors.wiki.ti.com/index.php?oldid=77716 *Contributors*: Hao, RajSivarajan, Sajeshsaran, ToanTruong

MCSDK Image Processing Demonstration Guide *Source*: http://processors.wiki.ti.com/index.php?oldid=121431 *Contributors*: A0792105, ChrisRing, Csmith, DanRinkes, Jbtheou, Mdamato, RajSivarajan, Sajeshsaran

MCSA and the MCSDK Demo *Source*: http://processors.wiki.ti.com/index.php?oldid=73301 *Contributors*: A0792105, A0850941

Image Sources, Licenses and Contributors

Image:TIBanner.png *Source*: http://processors.wiki.ti.com/index.php?title=File:TIBanner.png *License*: unknown *Contributors*: Nsnehaprabha
Image:C66x-multicore.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:C66x-multicore.jpg *License*: unknown *Contributors*: RajSivarajan
File:Helpful_tips_image.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:Helpful_tips_image.jpg *License*: unknown *Contributors*: DanRinkes, PagePusher
Image:MCSDK200SoftwareStack.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:MCSDK200SoftwareStack.jpg *License*: unknown *Contributors*: RajSivarajan, Sajeshsaran
Image:Rmm structure overview.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Rmm_structure_overview.JPG *License*: unknown *Contributors*: Justin32
Image:QMSS Transport.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:QMSS_Transport.JPG *License*: unknown *Contributors*: Justin32
Image:SRIOtransport.PNG *Source*: http://processors.wiki.ti.com/index.php?title=File:SRIOtransport.PNG *License*: unknown *Contributors*: Justin32
Image:Ipc overview ladder.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Ipc_overview_ladder.JPG *License*: unknown *Contributors*: Justin32
Image:Ipc startup ladder.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Ipc_startup_ladder.JPG *License*: unknown *Contributors*: Justin32
Image:Ipc heap ladder.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Ipc_heap_ladder.JPG *License*: unknown *Contributors*: Justin32
Image:Ipc messageq ladder.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Ipc_messageq_ladder.JPG *License*: unknown *Contributors*: Justin32
Image:Ipc shared mem ladder.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Ipc_shared_mem_ladder.JPG *License*: unknown *Contributors*: Justin32
Image:Ipc qmss ladder.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Ipc_qmss_ladder.JPG *License*: unknown *Contributors*: Justin32
Image:Ndkarch.png *Source*: http://processors.wiki.ti.com/index.php?title=File:Ndkarch.png *License*: unknown *Contributors*: DanRinkes, Sajeshsaran
Image:Ndkarch-6657.png *Source*: http://processors.wiki.ti.com/index.php?title=File:Ndkarch-6657.png *License*: unknown *Contributors*: Ipang
Image:Importplatformlibproject.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:Importplatformlibproject.jpg *License*: unknown *Contributors*: AravindBatni, PagePusher
Image:Setprofileplatformlibproject.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:Setprofileplatformlibproject.jpg *License*: unknown *Contributors*: AravindBatni
Image:Import Project.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Import_Project.JPG *License*: unknown *Contributors*: Justin32
Image:Import NIMU.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Import_NIMU.JPG *License*: unknown *Contributors*: Justin32
Image:NIMU debug be set active.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:NIMU_debug_be_set_active.JPG *License*: unknown *Contributors*: Justin32
Image:Client big endian.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Client_big_endian.JPG *License*: unknown *Contributors*: Justin32
Image:Client big endian RTSC.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Client_big_endian_RTSC.JPG *License*: unknown *Contributors*: Justin32
Image:Client running.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Client_running.JPG *License*: unknown *Contributors*: Justin32
Image:Ndkdosbox.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:Ndkdosbox.jpg *License*: unknown *Contributors*: JackM
Image:Ndkdosboxbuild.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:Ndkdosboxbuild.jpg *License*: unknown *Contributors*: JackM
Image:Ndkdosboxbuilding.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:Ndkdosboxbuilding.jpg *License*: unknown *Contributors*: JackM
Image:Projectsettingshelloworld.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:Projectsettingshelloworld.jpg *License*: unknown *Contributors*: AravindBatni, Sajeshsaran
Image:Includepathhelloworld.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:Includepathhelloworld.jpg *License*: unknown *Contributors*: AravindBatni, Sajeshsaran
Image:Linkerinputhelloworld.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:Linkerinputhelloworld.jpg *License*: unknown *Contributors*: AravindBatni, Sajeshsaran
Image:LedRtscProject.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:LedRtscProject.JPG *License*: unknown *Contributors*: JackM
Image:LedPlayEx3.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:LedPlayEx3.JPG *License*: unknown *Contributors*: JackM
Image:LedPlayEx4.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:LedPlayEx4.jpg *License*: unknown *Contributors*: JackM
Image:IPC comm features.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:IPC_comm_features.JPG *License*: unknown *Contributors*: Justin32
Image:IPC Linux comm.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:IPC_Linux_comm.JPG *License*: unknown *Contributors*: Justin32
Image:IPC transport types.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:IPC_transport_types.JPG *License*: unknown *Contributors*: Justin32
Image:Threading model.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:Threading_model.jpg *License*: unknown *Contributors*: RajSivarajan
Image:Parallel for with reduction.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:Parallel_for_with_reduction.jpg *License*: unknown *Contributors*: RajSivarajan
Image:OpenMP Solution Stack.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:OpenMP_Solution_Stack.jpg *License*: unknown *Contributors*: RajSivarajan
Image:Import OpenMPEx1Project.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Import_OpenMPEx1Project.JPG *License*: unknown *Contributors*: RajSivarajan
Image:MCSDK components.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:MCSDK_components.JPG *License*: unknown *Contributors*: Gurnani
Image:OpenMPEx1Project EnableOMPCompile.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:OpenMPEx1Project_EnableOMPCompile.JPG *License*: unknown *Contributors*: RajSivarajan
Image:OpenMPEx1Project Output.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:OpenMPEx1Project_Output.JPG *License*: unknown *Contributors*: RajSivarajan
Image:Post.png *Source*: http://processors.wiki.ti.com/index.php?title=File:Post.png *License*: unknown *Contributors*: AravindBatni, Hao, RajSivarajan, Sajeshsaran
Image:Nandboot.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:Nandboot.jpg *License*: unknown *Contributors*: Sajeshsaran
Image:Norboot.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:Norboot.jpg *License*: unknown *Contributors*: Sajeshsaran
Image:Emacboot.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:Emacboot.jpg *License*: unknown *Contributors*: Sajeshsaran
Image:CCSHelp CheckForUpdates.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:CCSHelp_CheckForUpdates.jpg *License*: unknown *Contributors*: AravindBatni
Image:CCSWin InstallUpdate.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:CCSWin_InstallUpdate.jpg *License*: unknown *Contributors*: AravindBatni
Image:CCSWin AvailableSw.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:CCSWin_AvailableSw.jpg *License*: unknown *Contributors*: AravindBatni
Image:CCSInst AvailableSw.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:CCSInst_AvailableSw.jpg *License*: unknown *Contributors*: AravindBatni
Image:Ccs-help.png *Source*: http://processors.wiki.ti.com/index.php?title=File:Ccs-help.png *License*: unknown *Contributors*: DanRinkes, JackM, PagePusher, Sajeshsaran
Image:1 dual board device manager.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:1_dual_board_device_manager.JPG *License*: unknown *Contributors*: Justin32
Image:2 config first target.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:2_config_first_target.JPG *License*: unknown *Contributors*: Justin32
Image:3 new connection.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:3_new_connection.JPG *License*: unknown *Contributors*: Justin32
Image:4 new board add proc.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:4_new_board_add_proc.JPG *License*: unknown *Contributors*: Justin32
Image:5 two boards.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:5_two_boards.JPG *License*: unknown *Contributors*: Justin32
Image:6 connection properties.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:6_connection_properties.JPG *License*: unknown *Contributors*: Justin32
Image:7 new port.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:7_new_port.JPG *License*: unknown *Contributors*: Justin32
Image:8 sixteen cores.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:8_sixteen_cores.JPG *License*: unknown *Contributors*: Justin32
Image:9 connected to cores.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:9_connected_to_cores.JPG *License*: unknown *Contributors*: Justin32
File:UDP_loopback_diagram.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:UDP_loopback_diagram.JPG *License*: unknown *Contributors*: ToanTruong
File:TCP_loopback_diagram.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:TCP_loopback_diagram.jpg *License*: unknown *Contributors*: ToanTruong
Image:HuaArchitecture.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:HuaArchitecture.jpg *License*: unknown *Contributors*: Sajeshsaran
Image:Wirednwconnection.png *Source*: http://processors.wiki.ti.com/index.php?title=File:Wirednwconnection.png *License*: unknown *Contributors*: AravindBatni
Image:Inputpage.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:Inputpage.jpg *License*: unknown *Contributors*: Sajeshsaran
Image:Outputpage.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:Outputpage.jpg *License*: unknown *Contributors*: Sajeshsaran
File:instrument1.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:Instrument1.jpg *License*: unknown *Contributors*: A0792105
File:execGraph1.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:ExecGraph1.jpg *License*: unknown *Contributors*: A0792105

License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.
BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

License

1. Definitions

a. "**Adaptation**" means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered an Adaptation for the purpose of this License.
b. "**Collection**" means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(f) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined below) for the purposes of this License.
c. "**Creative Commons Compatible License**" means a license that is listed at http://creativecommons.org/compatiblelicenses that has been approved by Creative Commons as being essentially equivalent to this License, including, at a minimum, because that license: (i) contains terms that have the same purpose, meaning and effect as the License Elements of this License; and, (ii) explicitly permits the relicensing of adaptations of works made available under that license under this License or a Creative Commons jurisdiction license with the same License Elements as this License.
d. "**Distribute**" means to make available to the public the original and copies of the Work or Adaptation, as appropriate, through sale or other transfer of ownership.
e. "**License Elements**" means the following high-level license attributes as selected by Licensor and indicated in the title of this License: Attribution, ShareAlike.
f. "**Licensor**" means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.
g. "**Original Author**" means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.
h. "**Work**" means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.
i. "**You**" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.
j. "**Publicly Perform**" means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.
k. "**Reproduce**" means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. Fair Dealing Rights

Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant

Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections;
b. to create and Reproduce Adaptations provided that any such Adaptation, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";
c. to Distribute and Publicly Perform the Work including as incorporated in Collections; and,
d. to Distribute and Publicly Perform Adaptations.
e. For the avoidance of doubt:

    i. **Non-waivable Compulsory License Schemes.** In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;
    ii. **Waivable Compulsory License Schemes.** In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor waives the exclusive right to collect such royalties for any exercise by You of the rights granted under this License; and,
    iii. **Voluntary License Schemes.** The Licensor waives the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License.

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. Subject to Section 8(f), all rights not expressly granted by Licensor are hereby reserved.

4. Restrictions

The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(c), as requested. If You create an Adaptation, upon notice from any Licensor You must, to the extent practicable, remove from the Adaptation any credit as required by Section 4(c), as requested.
b. You may Distribute or Publicly Perform an Adaptation only under the terms of: (i) this License; (ii) a later version of this License with the same License Elements as this License; (iii) a Creative Commons jurisdiction license (either this or a later license version) that contains the same License Elements as this License (e.g., Attribution-ShareAlike 3.0 US)); (iv) a Creative Commons Compatible License. If you license the Adaptation under one of the licenses mentioned in (iv), you must comply with the terms of that license. If you license the Adaptation under the terms of any of the licenses mentioned in (i), (ii) or (iii) (the "Applicable License"), you must comply with the terms of the Applicable License generally and the following provisions: (I) You must include a copy of, or the URI for, the Applicable License with every copy of each Adaptation You Distribute or Publicly Perform; (II) You may not offer or impose any terms on the Adaptation that restrict the terms of the Applicable License or the ability of the recipient of the Adaptation to exercise the rights granted to that recipient under the terms of the Applicable License; (III) You must keep intact all notices that refer to the Applicable License and to the disclaimer of warranties with every copy of the Work as included in the Adaptation You Distribute or Publicly Perform; (IV) when You Distribute or Publicly Perform the Adaptation, You may not impose any effective technological measures on the Adaptation that restrict the ability of a recipient of the Adaptation from You to exercise the rights granted to that recipient under the terms of the Applicable License. This Section 4(b) applies to the Adaptation as incorporated in a Collection, but this does not require the Collection apart from the Adaptation itself to be made subject to the terms of the Applicable License.
c. If You Distribute, or Publicly Perform the Work or any Adaptations or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and (iv) , consistent with Ssection 3(b), in the case of an Adaptation, a credit identifying the use of the Work in the Adaptation (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). The credit required by this Section 4(c) may be implemented in any reasonable manner; provided, however, that in the case of a Adaptation or Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Adaptation or Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.
d. Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Adaptations or Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation. Licensor agrees that in those jurisdictions (e.g. Japan), in which any exercise of the right granted in Section 3(b) of this License (the right to make Adaptations) would be deemed to be a distortion, mutilation, modification or other derogatory action prejudicial to the Original Author's honor and reputation, the Licensor will waive or not assert, as appropriate, this Section, to the fullest extent permitted by the applicable national law, to enable You to reasonably exercise Your right under Section 3(b) of this License (right to make Adaptations) but not otherwise.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability

EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Adaptations or Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.
b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

a. Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.
b. Each time You Distribute or Publicly Perform an Adaptation, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.
c. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.
d. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.
e. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.
f. The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in the applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights are deemed to be included in the License; this License is not intended to restrict the license of any rights under applicable law.