|  |
| --- |
| Texas Instruments |
| Keystone II Multicore Workshop |
| ARM-based Lab Manual |

April 2014

Contents

[Prerequisites 1](#_Toc384896533)

[Hardware 1](#_Toc384896534)

[Software 1](#_Toc384896535)

[Workshop Network 2](#_Toc384896536)

[Setting up a Serial Terminal Session to the EVM via USB 3](#_Toc384896537)

[Setting up a VNC View to the Ubuntu Server 3](#_Toc384896538)

[Lab 1 – EVM Board Bring-up and OOB Demonstration 6](#_Toc384896539)

[Purpose 6](#_Toc384896540)

[Task 1: Loading the U-BOOT 3](#_Toc384896541)

[Verify the Current U-BOOT **Error! Bookmark not defined.**](#_Toc384896542)

[Program SPI NOR Flash with U-boot GPH image 3](#_Toc384896543)

[Task 2: Load and Run standard “Hello World” application 6](#_Toc384896544)

[Lab 2 – Build a New ARM Program 11](#_Toc384896545)

[Projects and source code 11](#_Toc384896546)

[Purpose 11](#_Toc384896547)

[Task 1: Modify the File system 11](#_Toc384896548)

[Example Simple Code 12](#_Toc384896549)

[Build the Executable 12](#_Toc384896550)

[Unzip and Decompress the File System and add the new executable 13](#_Toc384896551)

[Compress and Zip the New File System 14](#_Toc384896552)

[Reboot the EVM and Run the New Program 14](#_Toc384896553)

[Task 2 (optional): Build U-boot, Boot Monitor and Kernel **Error! Bookmark not defined.**](#_Toc384896554)

[U-BOOT Source Code Extraction **Error! Bookmark not defined.**](#_Toc384896555)

[U-BOOT Build Instructions **Error! Bookmark not defined.**](#_Toc384896556)

[Boot Monitor (skern.bin) Source Code Extraction and Build Instructions **Error! Bookmark not defined.**](#_Toc384896557)

[Linux Kernel and Device Tree Source Code Extraction and Build Instructions **Error! Bookmark not defined.**](#_Toc384896558)

[Lab 3: Boot Using NFS-mounted File System 15](#_Toc384896559)

[Purpose 15](#_Toc384896560)

[Task 1: Build a file system on a Linux Host, Use the NFS server 15](#_Toc384896561)

[Task 2: Configure U-BOOT to mount the file server and boot 15](#_Toc384896562)

[Task 3: Build a new C program in the file system, and debug it 16](#_Toc384896563)

[Lab 4: Build, Run and Optimize DSP Project using CCS 27](#_Toc384896564)

[Purpose 27](#_Toc384896565)

[Project Files 27](#_Toc384896566)

[Task 1: Build and Run the Project 27](#_Toc384896567)

[Task 2: Define the Target 35](#_Toc384896568)

[Task 3: Connect to the EVM 36](#_Toc384896569)

[Task 4: Load and Run CASE 1 37](#_Toc384896570)

[Task 5: Use Optimization and Disable Symbol Debug for the fir Filter Routine 38](#_Toc384896571)

[Task 6: Optimize Software Pipeline 40](#_Toc384896572)

[Task 7: Enable the Cache 42](#_Toc384896573)

[Task 8: Running in Parallel on Multiple Cores 43](#_Toc384896574)

[Lab 5: Load and Run DSP Code Using MPM Server 47](#_Toc384896575)

[Purpose 47](#_Toc384896576)

[Project Files 47](#_Toc384896577)

[Task 1: Build and Run the Project 47](#_Toc384896578)

[Task 2: Using MPM to load, run and observe results 50](#_Toc384896579)

[Lab 6: ARM-DSP Communication Using MPM & Shared DDR 51](#_Toc384896580)

[Purpose 51](#_Toc384896581)

[Linux and DSP simple memory management 51](#_Toc384896582)

[Project Files 54](#_Toc384896583)

[Task 1: Build and Run the Project 54](#_Toc384896584)

[Lab 7: Build and Run ARM Code Using CCS on Linux 57](#_Toc384896585)

[Purpose 57](#_Toc384896586)

[Prerequisites 57](#_Toc384896587)

[Task 1: CCS - Start a new cross compiler project and build it 57](#_Toc384896588)

[Task 2: GDB from the tera-terminal 63](#_Toc384896589)

[Move the Debug directory outside of CCS 63](#_Toc384896590)

[Task 3: Connect a SSH terminal into the target 64](#_Toc384896591)

[Task 4: Optional – Use gdb to debug the application 74](#_Toc384896592)

[Task 5: Configure the host proxy 78](#_Toc384896593)

[Lab 8: ARM Optimization Using SMP Linux 79](#_Toc384896594)

[Projects and source code 79](#_Toc384896595)

[Purpose 79](#_Toc384896596)

[Task 1: Copy and observe the source files 79](#_Toc384896597)

[Task 2: Compile and build the project, run it on a single core 80](#_Toc384896598)

[Task 3: Run the code on multiple cores 84](#_Toc384896599)

[Lab 9 – Inter-Processor Communication (IPC) Lab 85](#_Toc384896600)

[Projects and source code 85](#_Toc384896601)

[Purpose 85](#_Toc384896602)

[Task 1: Run the demo from a Web Server 85](#_Toc384896603)

[Step 1 85](#_Toc384896604)

[Step 2 85](#_Toc384896605)

[Step 3 85](#_Toc384896606)

[Task 2: Run the demo from the terminal 90](#_Toc384896607)

[Purpose 90](#_Toc384896608)

[Step 1 – Understand the file demo\_ipc.sh 90](#_Toc384896609)

[Step 2 – Run the file demo\_ipc.sh 91](#_Toc384896610)

[Task 3: Rebuild the Executable 92](#_Toc384896611)

[Purpose 92](#_Toc384896612)

[Step 1 – Get a private copy of IPC from the release 92](#_Toc384896613)

[Step 2 – Build the ARM executable 93](#_Toc384896614)

[Step 3 – Build the DSP executable 98](#_Toc384896615)

[Task 4 – Optional: Modify source code and rebuild the executable 100](#_Toc384896616)

[Purpose 100](#_Toc384896617)

[Step 1 – Modify the ARM code 100](#_Toc384896618)

[Step 2 – Build the ARM code 101](#_Toc384896619)

[Step 3 – Modify the DSP code 101](#_Toc384896620)

[Step 4 – Build the DSP code 102](#_Toc384896621)

[Step 5 – Run the file demo\_ipc.sh 102](#_Toc384896622)

[Lab 10: Build U-boot, Boot Monitor and Kernel 103](#_Toc384896623)

[U-BOOT source code extraction and build instructions 103](#_Toc384896624)

[Boot Monitor (skern.bin) source code extraction and build instructions 104](#_Toc384896625)

[Linux Kernel and device tree source code extraction and build instructions 105](#_Toc384896626)

# Prerequisites

The following hardware and software are needed to perform the labs in this manual.

## Hardware

1. Update BMC and UCD on EVMK2H (optional):
   1. The wiki page <http://processors.wiki.ti.com/index.php/EVMK2H_Hardware_Setup> gives instructions on how to detect if the board needs BMC (Baseboard Management Controller) update. It also instructs how to do the update the BCM using CCS.  
      NOTE: A PDF version of the wiki page (KeyStone2\_EVM\_hardwareSetUp.pdf) is also available. Ask your instructor.
   2. The user must also check the UCD Power Management version (see EVMK2H Hardware Setup at link above) and update if necessary.  
      NOTE: Instructions and scripts that show how to update the UCD are provided in the zip file XTCIEVMK2X-UCD-Update.zip (ask your instructor).
2. TI training EVMs were already updated. The above concern ONLY customers’ EVMs

## Software

The following software packages must be pre-installed on the student laptop before the workshop starts. Note: During the workshop the laptop is attached to local network and has limited access to internet.

1. Download the MCSDK and CCS
   1. For details regarding the instructions in this section, refer to the [MCSDK User Guide for KeyStone II](http://processors.wiki.ti.com/index.php/MCSDK_User_Guide_for_KeyStone_II).
   2. The latest release of MCSDK is found here:  
      <http://software-dl.ti.com/sdoemb/sdoemb_public_sw/mcsdk/latest/index_FDS.html>
      1. For this lab you can use the Windows or the Linux version, depends on your laptop. Linux MCSDK was pre-installed on an Ubuntu server that will be used in some of the labs.
   3. From the same download page as the MCSDK, locate and download the latest CCS version and the emupack version that goes with the CCS. Follow the instructions on the page. Note, installing CCS requires licensing from TI.
2. Installing VNC Viewer

VNC server that supports graphic interface was installed on the Ubuntu server. Each laptop must have a VNC viewer. Texas Instruments and many other corporations purchased global licenses for Real VNC enterprise users and it can be downloaded from internal software download site (EDS). Limited functionality Real VNC viewer is available as freeware from multiple sites.

1. FTP Client

FTP server is installed on the Ubuntu server. Moving files between the student Laptop and the Ubuntu server can be done with the enterprise version of Real VNC or (if the student uses a freeware real VNC) by using ftp client on the laptop. The student must confirm that ftp client is installed on the laptop.

1. For communication between the student PC and the EVM, the FTDI driver is required. As needed, download the 32-bit driver here: <http://www.ftdichip.com/Drivers/D2XX.htm>
2. Terminal emulator such as Tera-term or Putty (or other). TI installed Tera-Term on TI’s laptops.
3. It is assumed that the user knows how to use the tools, VNC, terminal emulator, and ftp client.

### Workshop Network



The diagram above shows the workshop network environment:

* There are up to 10 lab stations. EVMs at each station are numbered from 1 to 10. Each station has the following:
  + One EVMK2H
  + One laptop that is connected to the EVM via JTAG cable, called the first laptop.
  + One optional laptop that is not connected to the EVM, called the second laptop in the station.
* All EVMs and students laptops are connected to the local network 192.168.0.XX via a wired connection to a switch or a router.
* The Ubuntu server is connected as well. The Ubuntu server has access to an external network with a global IP that have access to the Web.
* The IP addresses of the local network (192.168.0.XX) is given by Ubuntu server DHCP

### Setting up a Serial Terminal Session to the EVM via USB

The EVM has two (mini) USB ports.

* + One of the ports accesses the JTAG connection and can be used to connect CCS to the board. This USB connector is part of the emulator daughter (mezzanine) card.
  + The second USB port is part of the mother board and can be used to connect two serial terminals into the EVM. We will refer to these serial terminals as Tera-Terminal (to distinguish from the window viewer terminal to the Ubuntu machine). The tera-terminals are connected using a single USB cable but can be opened as two tera-terminals.
    - The first serial terminal (ARM Tera-terminal) is connected to the ARM terminal (e.g., the lower COM port)
    - The second serial terminal (BMC Tera-terminal) is connected to the FPGA/BMC on the board.
    - The user must open the two tera-terminals and set the serial rate to 115200 Baud.

### Setting up a VNC View to the Ubuntu Server

1. Launch the VNC Viewer application from the desktop of your laptop/PC.
2. The server IP will be given by the instructor. For static configuration, when DHCP is not available, the server IP is 192.168.0.100.

* The login instance for student N is :N. For example, student 3 will VNC to address 192.168.0.100:3, while student number 7 will use 192.168.0.100:7.
* The VNC password for all students is “vncserve ”

## Updating the U-BOOT

The U-BOOT that is programmed into flash on the EVM must be updated when moving ti a new version of the MCSDK. The following process will be done at the beginning of every training session

### Update SPI NOR Flash with U-boot GPH image

The following process is used to update the U-BOOT image in SPI Flash. It must be done every time a new release of MCSDK is used.

1. Power cycle the EVM and stop the autoboot by pressing any key.
2. The image sub-directory of the MCSDK release (for MCSDK release 3.0.4.18 the image directory path is /tiTools/MCSDK\_3\_4\_18\mcsdk\_linux\_3\_00\_04\_18\images) has a gph file - **u-boot-spi-keystone-evm.gph.** This file was copied to the TFTP root directory (see the table below for the path to the TFTP root directory)   
     
   Make sure the tftp server is running. Then issue the following commands to U-Boot console:

**setenv serverip 192.168.0.100**

**dhcp 0xc300000 u-boot-spi-keystone-evm.gph**

**sf probe**

**sf erase 0 < u-boot-spi-keystone-evm.gph in hex up rounded to sector boundary of 0x10000>**

**sf write 0xc300000 0 <size of u-boot-spi-keystone-evm.gph image in hex>**

NOTE: The size of the image will be displayed as part of the DHCP command.

# Server Directory Structure

The following directories and sub-directories were added to facilitate the workshop. The directory name includes the absolute path of any directory.

|  |  |  |
| --- | --- | --- |
| Directory | Purpose | Comments |
| /tiTools | Contains TI tools that are to be used by all students: Linaro cross compiler tool chain, CCS, and MCSDK | Sub-directories are MCSDK, CCS and Linaro tools chain (cross compiler) |
| /tiTools/CCS | CCS installation location |  |
| /ti/TOOLS/MCSDK\_X\_YY | MCSDK installation directory, version number is X\_YY |  |
| /tiTools/gcc | Linaro tools chain – cross compiler |  |
| /titftpboot | Root directory for the TFTP server. | Each student has a sub-directory |
| / titftpboot /studentN | TFTP directory for student N, where N is 1 .., 10 | Student has to copy images from the MCSDK to this directory for ramfs boot |
| /opt/filesys | Root directory for the NFS server that enables mounting of the server file system into the EVM | Each student has a sub-directory |
| / opt/filesys /studentN | NFS directory for student N, where N is 1..,10 | Each student should build private file system into this directory |
| /usr/global/scripts | This directory has scripts that initialize environment variables. scriptInsideTI.sh is used when the server is inside TI network, and scriptOutsideTI.sh is used when the server is outside TI firewall. Other scripts may be developed for other locations | The student must run the script for every terminal by doing  Source /usr/global/scripts/scriptXXX.sh |
| / usr/global/projects | Contains the source code for projects that are used during the Lab. | It has two sub-directories, DSP and ARM. Students will copy the source code files from this directory to their private directories |
| / usr/global/projects /ARM | Source code for ARM projects |  |
| / usr/global/projects /DSP | Source code for DSP projects |  |
| /home/studentN | Home private directory of student N N=1 ..,10 | All changes to files are done in the student private directory |
| // usr/global/git | All sources for TI Arago distribution |  |

# 

# Lab 1 – EVM Board Bring-up

## Purpose

The purpose of this lab is to boot the EVM from TFTP server. In addition to the kernel, device tree and the monitor, the file system is loaded from the TFTP server.

### Task 1: Load and Run standard “Hello World” application

1. In order for the U-BOOT to get files from a sub-directory, the tftp download path for u-boot command needs to be specified via the **tftp\_root** value. In our server, the root address of TFTP is /tftpboot. Each student has a private sub-directory /tftpboot/studentN where N is the student number.
2. Make a subdirectory **/tftpboot/studentN** if it does not exist already and copy the MCSDK release binary images into this directory. The binary images are located in the /tiTools/MCSDK\_Y\_Z\_XX/mcsdk\_linux/images directory on the Ubuntu server, where X,Y and Z are the release number.

**cd /tftpboot/studentN**

Where N is the student number you have been assigned for this lab.

**cp /tiTools/MCSDK\_3\_XX\_Z\_YY/mcsdk\_linux\_3\_00\_0X\_XX/images/\*.\* .**

Where 0X\_XX is the most current MCSDK release number (e.g., 04\_18)

1. U-BOOT loading and running Linux Kernel using TFTP with ramfs file system
   1. First verify that the DIP switch (sw1) are in ARM SPI boot mode:  
      1 OFF 2 OFF 3 ON 4 OFF
   2. Power up EVM, look at the ARM tera-terminal window

1. It is very important to set the environment variables correctly for U-BOOT. Here are the instructions how to do so:
   1. After power cycle, press the return key to stop autoboot in the ARM tera-terminal
   2. Enter the following command to reset the current environment variables.

**env default -f –a**

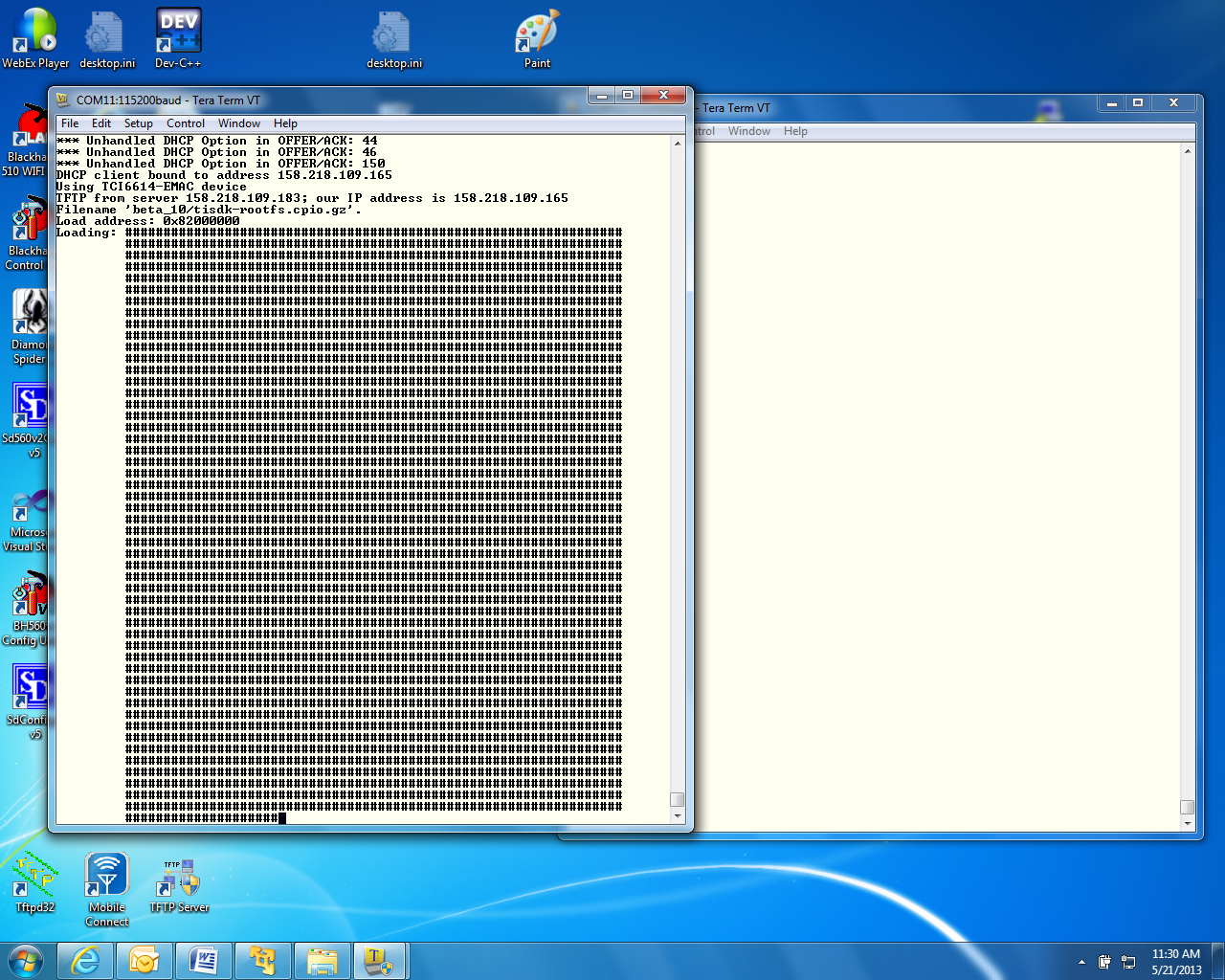
* 1. MCSDK release has multiple file systems. The strip-down file system name is arago-console-image.cpio.gz. This file system is small and does not have most of the applications. Two other file systems, tisdk-rootfs and tisdk-rootfs-rt (real time) include all TI applications and tools, but they are too large to be loaded from TFTP. We will use these file system during mounting boot and USB boot. We use a smaller version of tisdk-rootfs.cpio.gz from release 3\_15 in this Lab. Note – this file system will be used ONLY in lab 1.
  2. The file system used in this example istisdk-rootfs\_3\_15.cpio.gz.
  3. The following are the step needed to configure the environment variables in u-boot
     1. In the terminal write “print variableName” where variable name is the environment variable that you want to change
     2. If the return value is the correct value, you are done with this variable
     3. If not, do “setenv variableName ‘variable value’ where variable value is given in the instructions. You can use copy past and modify for long variable values
     4. After all variables are configured do “saveenv”
     5. For example, to set the name\_fs to arago-console-image.cpio.gz do the following
        1. **print name\_fs**
        2. If the value is not tisdk-rootfs\_3\_15.cpio.gz already do
        3. **setenv name\_fs ‘tisdk-rootfs\_3\_15.cpio.gz ‘**
        4. After configuring all the variables do **saveenv**

The following is a list of variables (variableName) and the values they should be (variable value)

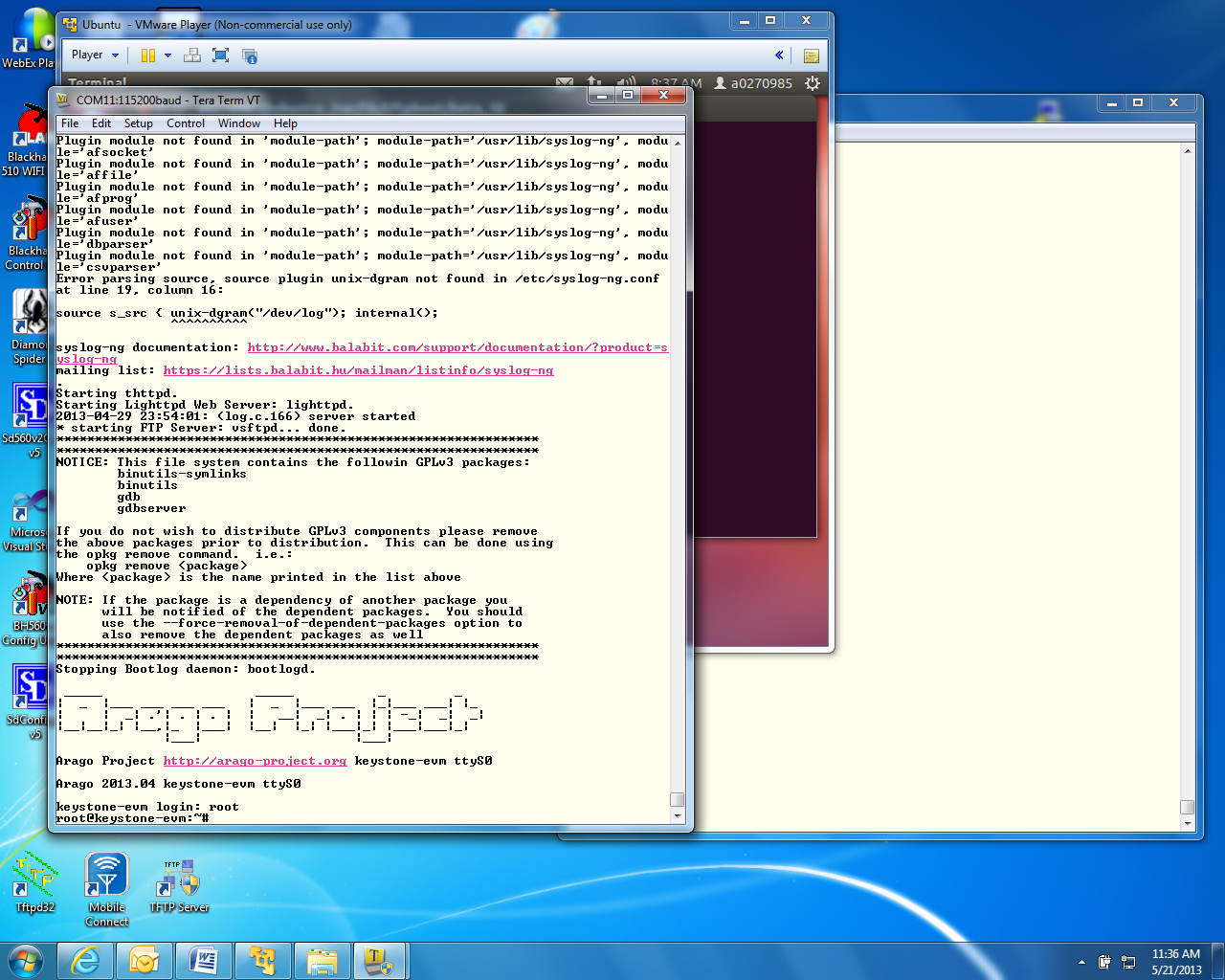
|  |  |
| --- | --- |
| Variable | Value |
| **args\_ramfs** | **'setenv bootargs ${bootargs} earlyprintk rdinit=/sbin/init rw root=/dev/ram0 initrd=0x802000000, 80M’** |
| name\_fs | tisdk-rootfs\_3\_15.cpio.gz |
| name\_fdt | **uImage-k2hk-evm.dtb** |
| **name\_kern** | **uImage-keystone-evm.bin** |
| **name\_mon** | **skern-keystone-evm.bin** |
| serverip | 192.168.0.100 |
| boot | ramfs |
| tftp\_root | studentN where N is the student number |
|  |  |

* 1. At the end do not forget to save the setting **saveenv**
  2. Boot the EVM using either a hardware or software reboot.
* Hardware reboot = power cycle
* Software reboot = type **reboot** in the BMC terminal window
* From U-boot prompt, type **boot**

1. The tera-terminal will start as follows:



1. When booting ends, login as **root** (no password)



After login as root, run the hello world program ./hello and look for the hello world response

# Lab 2 – Build a New ARM Program

## Projects and source code

All projects and source code are available on the Ubuntu server. The directory **/tiProjects** has two sub-directories:

* **/tiProjects/ARM** contains ARM projects
* **/tiProjects/DSP** contains DSP projects

NOTE: When the DSP projects are built using CCS on the student PC, projects should be moved via FTP from the server to the student laptop.

## Purpose

The purposes of this lab are:

1. To demonstrate how to build a simple ARM program using all cross compiler tools on Ubuntu server.
2. Build a new file system and load the net file system to the EVM
3. Run the built code

### Task 1: Modify the File system

First, you will modify the file system that was previously loaded onto the EVM

Modifying the file system involves three steps:

1. First, a new main function is developed. Using the cross compiler tools on Ubuntu, the function is compiled and an executable is built.
2. Next, the current compressed file system is unzipped and de-compressed into a temporary directory, and the new executable that was built in the previous step is added.
3. Last, the new file system is compressed, zipped, and moved to the tftp directory. The EVM is then booted, the new program is executed, and produces the expected results.

### Example Simple Code

.

1. The example code myHello.c is located on the Ubuntu server in **/tiProjects/ARM/myHello** directory
2. Copy this file to the student directory **/home/studentN/temp**

NOTE: If the temp directory does not exist, create it as follows:

* **cd ~** takes you to the home directory
* **mkdir temp**
* **cd temp**
* **sudo cp /tiProjects/ARM/myHello/myHello.c .**

1. **For system that is outside of TI fire wall, /tiScript/scriptOutside.sh** is a script that defines all the paths and exports for each individual user. The user must call this script for each new terminal:  
     
   **source /tiScript/scriptOutside.sh**
2. **For system inside of TI fire wall, /tiScript/scriptInsideTI.sh** is a script that defines all the paths and exports for each individual user.
3. The Linaro toolchain and all other shared software are installed on the Ubuntu server in directory **/tiTools/.** A path to the Linaro tool chain is defined in the script  **from above**
4. Before compiling look at the source code of myHello.c. You can modify it, add printf or any other C instruction that you like.

### Build the Executable

1. To use the cross compiler to build the executable, use the following command:   
     
   **arm-linux-gnueabihf-gcc -o myHello –g myHello.c**  
     
   The cross compiler tools will compile the file and build an executable called **myHello** in the same directory.
2. To verify that the compilation was done for the ARM processor and not for the native Intel (or other) processors do the following:  
     
   **file myHello**
3. The results should show the ARM architecture:   
     
   **“myHello: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.31, BuildID[sha1]=0x953dac672e7159d481d5a6d3bbb5356e5f870d21, not stripped”**

### Unzip and Decompress the File System and add the new executable

The compressed file system arago-console-image.cpio.gz has a cpio.gz compression. We will build a new file system in the student home directory.

1. Copy the current compressed file system to the new directory:

**sudo cp /tiTFTP/studentN/ arago-console-image.cpio.gz .**

1. Unzip the compressed file system

**sudo gzip –d arago-console-image.cpio.gz**

1. Uncompress the file system from the cpio file. This operation builds the complete file system.   
     
   **sudo cpio –i –F arago-console-image.cpio**
2. Remove arago-console-image.cpio

**sudo rm arago-console-image.cpio**

1. The file system resides in the temp directory. Copy the executable that was built previously (myHello) into the **usr/bin** directory in the file system. The complete path is **/home/studentN/temp/bin**

**sudo cp myHello /home/studentN/temp/bin/.**

**sudo cp myHello.c /home/studentN/temp/bin/.**

### Compress and Zip the New File System

1. The next step is to compress the file system back into a new file system. This is done by piping all the directories and the files in the file system into the cpio zipped format. From the temp directory (/home/studentN/temp) do the following:
   1. **sudo find . | sudo cpio –H newc –o –O myArago.cpio**
   2. **sudo gzip myArago.cpio**
2. The result file is **myArago.cpio.gz This file should be copied to the TFTP directory of the student**

**sudo cp myArago.cpio.gz /tiTFTP/studentN/.**

1. At this point, studentN has a new file systems myArago.cpio.gz.
2. The user should change the environment variable  **name\_fs** in the EVM U-BOOT to **myArago.cpio**

## Reboot the EVM and Run the New Program

1. Reboot the EVM and stop the U-BOOT before it starts loading
2. Change name\_fs to the new filesystem “ **setenv** **name\_fs myArago.cpio.gz “**
3. **saveenv**
4. **boot**
5. After boot, login as a **root**.
6. Run myHello ( **/bin/myHello**)
7. Observe the results.

# Lab 3: Boot Using NFS-mounted File System

## Purpose

The purposes of this lab are:

1. to demonstrate how to boot the EVM when the file system resides on a different server that is mounted on the EVM
2. Develop code on the EVM and use the native gcc tools to build a debuggable executable
3. Use gdb debugger to debug the developed code

### Task 1: Build a file system on a Linux Host, Use the NFS server

The NFS server is installed on the Ubuntu server in the directory /tiNFS. Each student has a sub-directory where he or she builds the file server, and the Uboot is configured to reach this directory for each student. The file system to be mounted should be built on the local Ubuntu machine.

1. Change the directory into the NFS mount private directory for studentN **/tiNFS/studentN** (where N is the student number).
2. Copy a tar version of the compressed file system **tisdk-rootfs.tar.gz** (part of the release in the images directory) into **/tiNFS/studentN** . This file system has the complete TI LINUX applications.
3. Untar the file system: **sudo tar zxf tisdk-rootfs.tar.gz**
4. Delete the original compress file: **sudo rm tisdk-rootfs.tar.gz**

### Task 2: Configure U-BOOT to mount the file server and boot

1. Power cycle the EVM. In the ARM tera-terminal window, stop the autoboot.
2. Change the following environment variables as described in Lab 1:

|  |  |
| --- | --- |
| Variable | Value |
| nfs\_serverip | 192.168.0.100 |
| boot | net |
| nfs\_root | /tiNFS/studentN where N is the student number |
| args\_net | **setenv args\_net 'setenv bootargs ${bootargs} rootfstype=nfs root=/dev/nfs rw nfsroot=${nfs\_serverip}:${nfs\_root},${nfs\_options} ip=dhcp'** |

1. Save the new environment variables:

**saveenv**

1. Boot the EVM.

### Task 3: Build a new C program in the file system, and debug it

1. After login as root you are in root home directory. Create a n application directory

**“mkdir applications”**

1. **Change directory to applications** cd applications
2. In the server window, copy myHello.c from /tiProjects/ARM/myHello to the mounted applications directory of root that is in /tiNFS/studentN/home/root/applications

**sudo cp /tiProjects/ARM/myHello/myHello.c /tiNFS/studentN/home/root/applications/.**

1. Back in the EVM terminal, locate the gcc native compiler and verify that it exists

Which gcc

1. The response should be /usr/bin/gcc
2. Compile and build the application similar to the method used in the previous lab, but add the debug flag (**-g**) to the command and use the native gcc :  
     
   **gcc –g -o myHello myHello.c**
3. Make sure that **myHello**.c and **myHello** are both in the applications directory  
     
   **ls –ltr myHello\***
4. Start a debug session:  
     
   **gdb myHello**
5. Simple gdb commands:
   * **list** see the source
   * **b** set a break point
   * **r** run to the break point
   * **s** step
   * **n** next (step over)
   * **c** run to the next breakpoint
   * **finish** end
6. There are many gdb quick guides on the Web. Here is a URL to one of them:

<http://condor.depaul.edu/glancast/373class/docs/gdb.html>

1. Put some break point, look at the source (using the list or l command), run to breakpoint, do next and so on and so forth

# Lab 4: Boot Using USB Flash Drive

## Purpose

The purpose of this lab is to demonstrate how to boot the EVM from a USB flash drive. The Kernel, monitor, and device tree reside on one partition of the USB. The file system resides on a second partition.

NOTE – USB boot is not available on release 2 and 3 of the EVM unless they were changed to supply the USB with power. Release 1.1 of the EVM has USB power.

## Task 1: Preparing the USB

NOTE: This procedure may already be done. In that case the instructor will provide pre-prepared USB memory stick. Preparing the USB requires Ubuntu host to be physically available for USB stick, so if the students have to prepare USB, it will be done one after another.

### On Ubuntu Host

### Step 1: Install gparted

$sudo apt-get install gparted

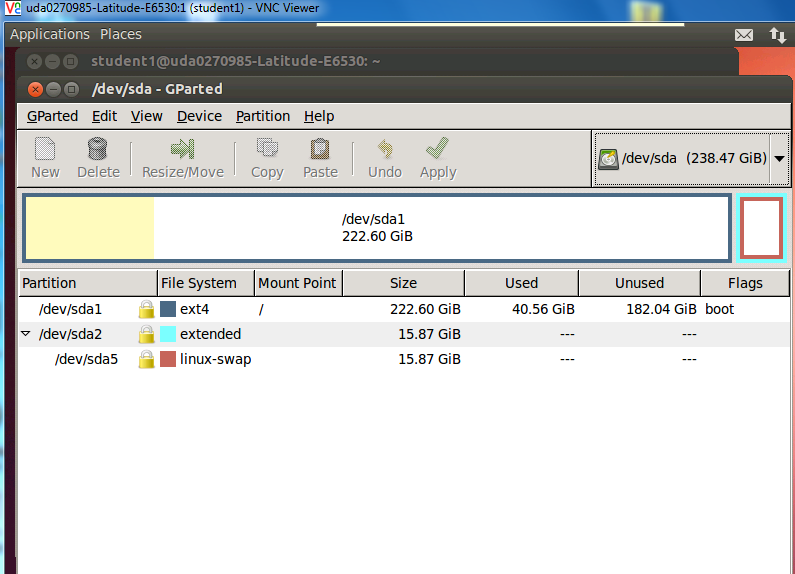
Used GParted 0.11.0 for this test. So exact steps may vary if version is different

### Step 2: Partition the Device

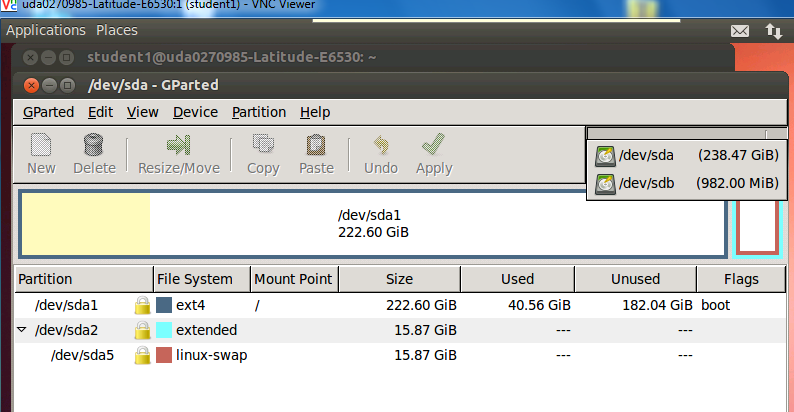
Before start partitioning the USB, verify what /dev are currently connected so it will be easy to identify the device name of the USB.

#### Prepare the USB for new partitions

* 1. **Do not** connect the USB stick to the server
  2. Start gparted **sudo gparted**
  3. The following is a screen shot from gparted
  4. In the upper right corner click the small arrow and verify what device are connected and what is their sizes. In the screen shot below there is a single device /dev/sda with size of 238.47GB



* 1. Quit gparted (Gparted pull down menu and quit) , insert the USB stick into a USB port, and start gparted again. This time the device list will show a new device. See the screen shot below



* 1. The new device /dev/sdb has size of 982MB. This is the USB stick that need re-format.
  2. Select the new device /dev/sdb in our case (can be different if the server has multiple disk, for example),
  3. Got to top level "Partition" pull down menu and select Unmount to unmount the device. If there is existing partition, delete the same (Partition -> delete)

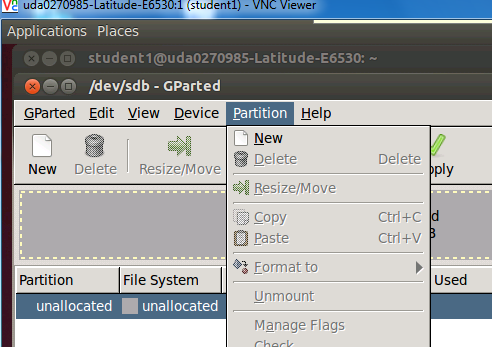


After delete the old partition start building two new partitions, partition 1 for boot images and partition 2 for rootfs

#### Add two new partitions

Now Partition and filesystem status shows as unallocated

* 1. Create fat32 partition for boot (contains boot images)



Partition -> new

New size = 32MiB

File system = fat32

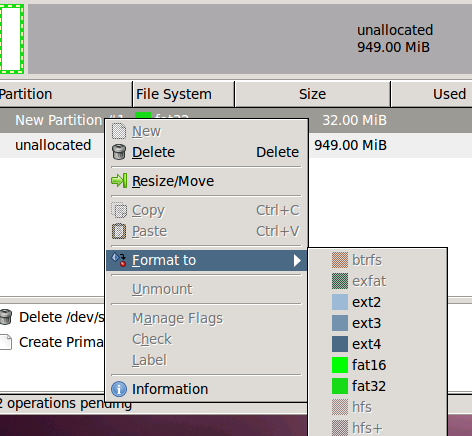
label = boot

Keep rest of the fields default

click add to add partition

select the partition #1 just created and format it (Partition -> format to).

select fat32 format



2. Create ext4 format for rootfs (contains root filesystem)

select the unallocated partition

Partition -> new

File system = ext4

label = rootfs

Keep rest of the fields default

click add to add partition

select the partition #2 just created and format it (Partition -> format to).

select ext4 format

3. Apply the changes and Quit

Edit -> Apply All Operations

GParted -> Quit

### Step 3: Copy Images and rootfs Files to Partitions

Assume that the USB device name is /dev/sdb Unmount if the devices are auto mounted.

$sudo umount /dev/sdb1

$sudo umount /dev/sdb2

1. Copy images to partition #1 (boot)

$sudo mount -t vfat /dev/sdb1 /mnt/test (first partition is mounted to /mnt/test)

Change directory to image release directory

cd /tiTools/MCSDK\_XX\_YY/mcsdk\_linux\_x\_y/image

Copy the kernel, device tree and monitor to the first partition

$sudo cp skern-keystone-evm.bin /mnt/test/

$sudo cp uImage-k2hk-evm.dtb /mnt/test/

$sudo cp uImage-keystone-evm.bin /mnt/test/

$ls /mnt/test

**skern-keystone-evm.bin uImage-k2hk-evm.dtb uImage-keystone-evm.bin**

$sudo umount /dev/sdb1

2. Copy rootfs files to partition #2 (rootfs)

Change directory to the student directory where the NFS mounted filesystem was built (in lab 3)

cd /tiNFS/studentN where N is the student number

$sudo mount -t ext4 /dev/sdb2 /mnt/test

cd ..

$sudo cp -r studentN/\* /mnt/test

$ls /mnt/test/

bin boot dev etc home init lib lost+found media mnt proc sbin srv sys tmp usr var

$sudo umount /dev/sdb2

## Task 2: Reboot the EVM

### Configure U-BOOT Environment Variables

Insert USB flash drive to usb slot on EVM and Power ON EVM Type the following commands to setup the env for usb boot:

setenv boot usb

setenv args\_usb 'setenv bootargs ${bootargs} rootfstype=ext4 root=/dev/sda2 rw ip=dhcp'

setenv init\_usb 'usb start; run set\_fs\_none args\_all args\_usb'

setenv get\_fdt\_usb 'fatload usb 0:1 ${addr\_fdt} ${name\_fdt}'

setenv get\_kern\_usb 'fatload usb 0:1 ${addr\_kern} ${name\_kern}'

setenv get\_mon\_usb 'fatload usb 0:1 ${addr\_mon} ${name\_mon}'

Make sure that name\_mon, name\_kern and name\_fdt are the same as were loaded into partition 1, in our case these are the expected values:

name\_fdt=uImage-k2hk-evm.dtb

name\_kern=uImage-keystone-evm.bin

name\_mon=skern-keystone-evm.bin

If any of the above value is not correct, use setenv to configure the correct value

saveenv

boot

Boot takes about 1 minute

Login as root and run the program that was developed for the NFS case (myHello)

# Lab 5: Build, Run and Optimize DSP Project using CCS

## Purpose

In previous labs, you developed and debugged an ARM program. The purpose of this lab is to develop and debug a multicore C66x program using CCS IDE. This lab has the following parts:

1. Using CCS, build a simple FIR project that runs on a single core.
2. Optimize the code by achieving software pipeline, understand what can prevent the compiler from generating software pipeline code.
3. Optimize execution by enabling cache.
4. Perform parallel processing of the code and observe multi-cores processing speed up.

CCS IDE is used to execute the lab.

Before starting, the EVM should be configured to no-boot mode. To do so, set the dipswitch (SW1) on the EVM to: 1 Off 2 Off 3 Off 4On

The EVM emulator is the mezzanine card on the top of the EVM. The mini USB cable should be connected to the mezzanine card and to a computer with CCS installed.

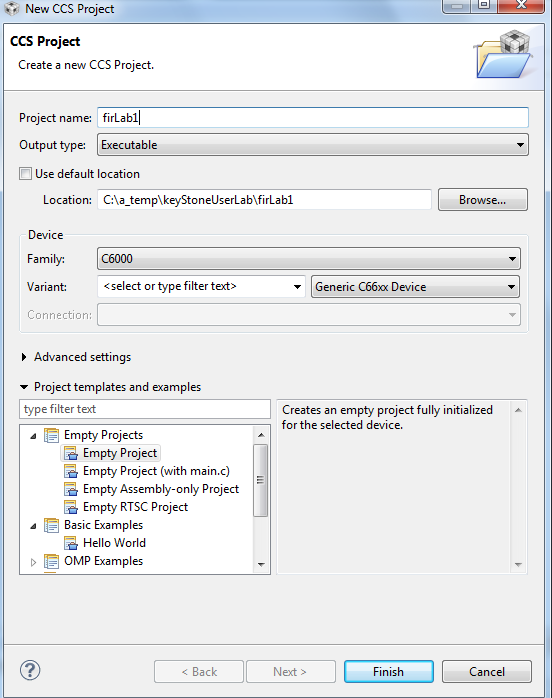
## Project Files

The following files are used in this lab:

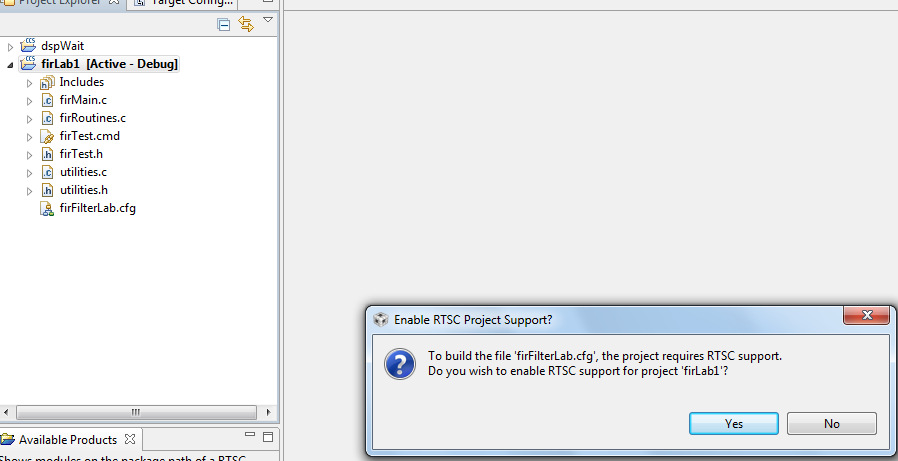
* firMain.c
* firRoutines.c
* firTest.cmd
* firTest.h
* utilities.c
* Utilities.h
* firFilterLab.cfg

### Task 1: Build and Run the Project

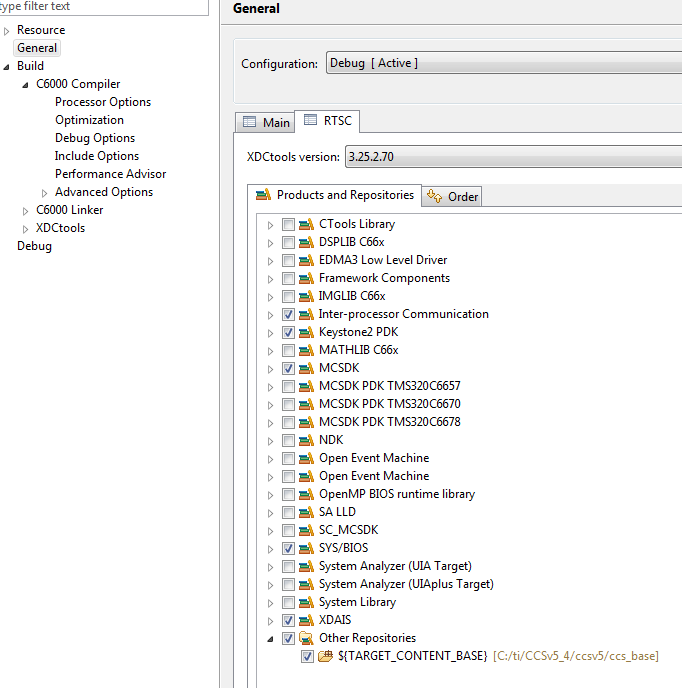
1. FTP into the Ubuntu server and get all the files in directory **/tiProjects/DSP/firFilter** into the local directory **c:\ti\labs\firFilter** on your PC. If this directory does not exist, create it.
2. Open CCS.
3. Create new project through the CCS menu item *File* 🡪 *New* 🡪 *CCS Project*.
4. Enter **firLab1**as a *Project Name*.
5. Click the check box to *Use default location.*
6. Set the *Family* to *C6000* and *Variant* to *Generic C66xxx Device*
7. Then press *Finish* to create the new project. See the screen shot.  
   NOTE: You will use the default location and not the location in the screen shot.



1. Then in the *Project Explorer* view, right-click on the newly-created *firLab1* project, and click on *Add Files…*
2. Browse to **C:\ti\labs\firFilter**, select all the files in this directory, and click *Open*. When prompted how files should be imported into the project, leave it as default of *Copy File.* If you defined the new project with **main.c,**remove the **main.c** file that may be created.
3. As soon as the file **firFilter.cfg** is imported into the project, CCS will ask you to enable RTSC support. See the screen shot below. Select *Yes*. Note, if CCS does not ask you to enable RTSC, rename the cfg file to some other name, and rename it back to firFilterLab.cfg



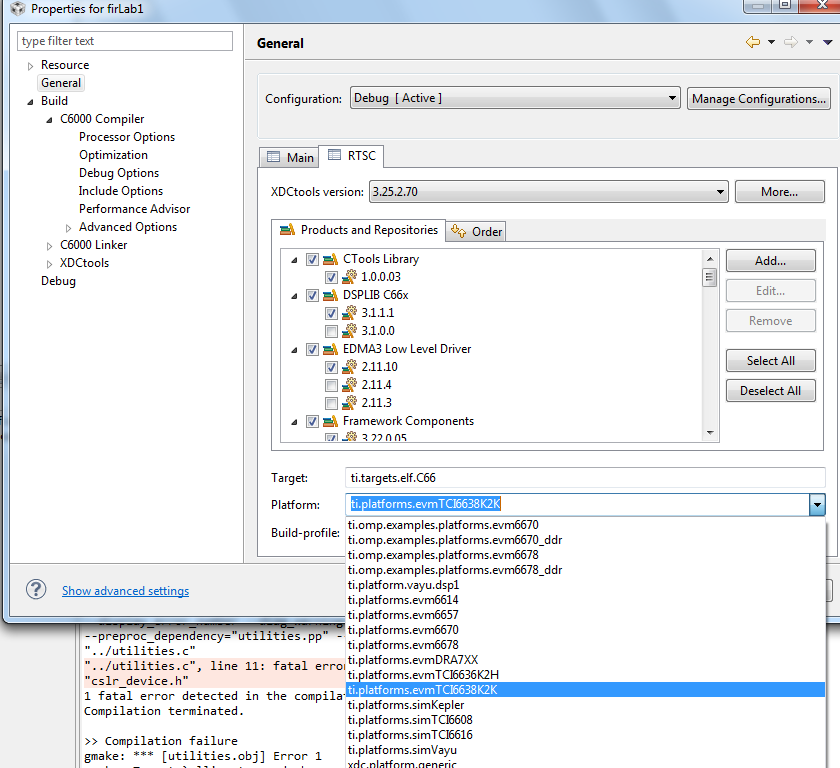
1. Open Project Properties and select general->RTSC. Look at the RTSC modules that are selected in the screen shot below and make sure that you select ONLY the same RTSC modules (or packages). When a project starts, RTSC attempts to include all the modules in the release. So unselect any module that is not in the screen shot.  
     
   NOTE: The TARGET CONTENT BASE should reflect the location of CCS on your system.



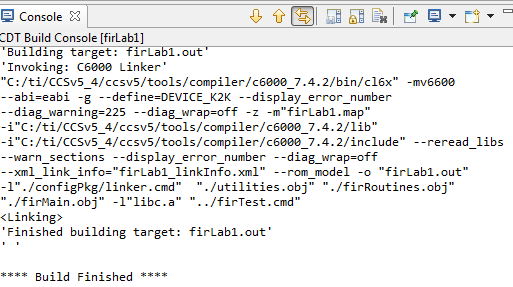
1. Click on the platform tab and select **ti.platform.evmTCI6638K2K**platform as shown in the next screen shot.

NOTE: RTSC projects require the user to select three types of information.

* The device family in the CCS create page determines what core is used and thus what version of the compiler should be used (different cores have different intrinsic functions).
* The platform that is defined here determines the memory configuration of the core.
* To build the correct RTSC drivers, the device name should be defined. This is done by adding a predefined symbol with the device name. More about it later.



1. Right click on the project name and select *rebuild*. If the build goes correctly, you will see the following in the console window.  
     
   NOTE: Look at the debug directory to ensure that the file **firLab1.out** is there. Ignore any warnings.



1. Examine the code in **firMain.c**. There are five cases, but only case 1 is not commented out.
   1. DSP 0 generates input data (**inputData**) and a set of filter coefficients (**filterCoef**)
   2. Depending on the case, a set of fir filters is applied to the data and the results are written to the out file (**outputFilter**).
   3. A set of timer registers (**TSCL** and **TSCH**) measure the execution time of the fir filter.
   4. The standard **printf** function prints the results on the console.

### Task 2: Define the Target

In this lab, the DSP code is run from no-boot mode. The no-boot mode requires setting the dipswitch SW1 of the EVM to: 1 Off 2 Off 3 Off 4 On.  
  
Since no-boot mode is chosen, the device configuration (DDR configuration, PLL configuration and so on) must be done in a gel file.

#### Create a new target in CCS

NOTE – the paths and other variables that are defined in this document may not reflect exactly the directory structure on your system. Use common sense and search to find the exact paths.

1. Create a new target configuration:
   1. Select the CCS menu option *View 🡪 Target Configurations*.
   2. Select *User Defined*.
   3. Right-click and select *New Target Configuration*.
2. Enter the name of the new target configuration in the *File Name:* text box.
   1. Set the File name based on the EVM model, *<model>.ccxml*  
      For example, ‘TCI6638.ccxml’
   2. Leave the *Location* the default value:  
      “C:\Documents and Settings\student\ti\CCSTargetConfigurations”
   3. Click the *Finish* button. The .ccxml file will now open in a GUI-based view with the *Basic* tab active.
3. Define the new target configuration by selecting the connection type in the *Basic* Tab.
4. The *Connection* drop-down menu identifies the emulator type. For example, ‘Texas Instruments XDS2xx USB Emulator.”
   1. *Board or Device* identifies the TI processor device, set it to 6638 and select TCI6638
   2. Under *Save Configuration*, click the *Save* button.
5. Configure setup in *Advance* Tab
   1. Click the *Advanced* tab at the bottom of the screen.
   2. Select Core 0 on the target device:
      * *TCI6638\_0* 🡪 *IcePick\_C\_0* 🡪 *Subpath\_1* 🡪 *C66xx\_0*
   3. You will now see a sub-window called *Cpu Properties* that allows you to choose an *initialization script*.
   4. Locate the appropriate GEL file, then click *Open*:
      * Depend on your ccs version, select: either **C:\ti\CCSv5\_4\ccsv5\ccs\_base\emulation\boards\evmtci6638k2k\gel\evmtci6638k2k.gel or C:\ti\CCS\_5\_5\ccsv5\ccs\_base\emulation\boards\xtcievmk2x\gel\** **xtcievmk2x.gel**
      * Repeat the process for all C66x cores, that is *C66xx\_1, C66xx\_2, … C66xx\_7*

Click the *Save* button.

### Task 3: Connect to the EVM

1. Click the Open Perspective (available at the right top corner of the CCS window).
2. Switch to the Debug Perspective by selecting the CCS menu option *Window* 🡪 *Open Perspective* 🡪 *CCS Debug*.
3. Select the CCS menu option *View* 🡪 *Target Configurations*. Select the target configuration you created
4. Launch the target configuration as follows:
   1. Select the target configuration .ccxml file.
   2. Right click and select *Launch Selected Configuration*.
5. This will bring up the *Debug* window. NOTE: This may take some time, but you will eventually see all the device cores.
   1. Select all C66x cores (select + Ctrl)
   2. Right click and choose *group cores*.
   3. Select the group, then right click and select *Connect Target.*

### Task 4: Load and Run CASE 1

1. Select the core group and load the .out file created earlier in the lab.
   1. Select the CCS menu option *Run* 🡪 *Load* 🡪 *Load Program*
   2. Click *Browse project…*
   3. Select *firLab1.out* by unwrapping the *firLab1->Debug* and click *OK.*
   4. Click *OK* to load the application to the target (all Cores).
2. Run the application by selecting the CCS menu option *Run* 🡪 *Resume*.
3. A successful run should produce a console output as shown below. Record the cycles time:

[C66xx\_0] start generating input data

finish generating input data

case 1 -> time consumed By core -> 0 610749952.000000

**Issues to think about:**

Look at the function CACHE\_disableCaching ((Uint8) 144) which disables cache-ability for memory region. What memory region is it?

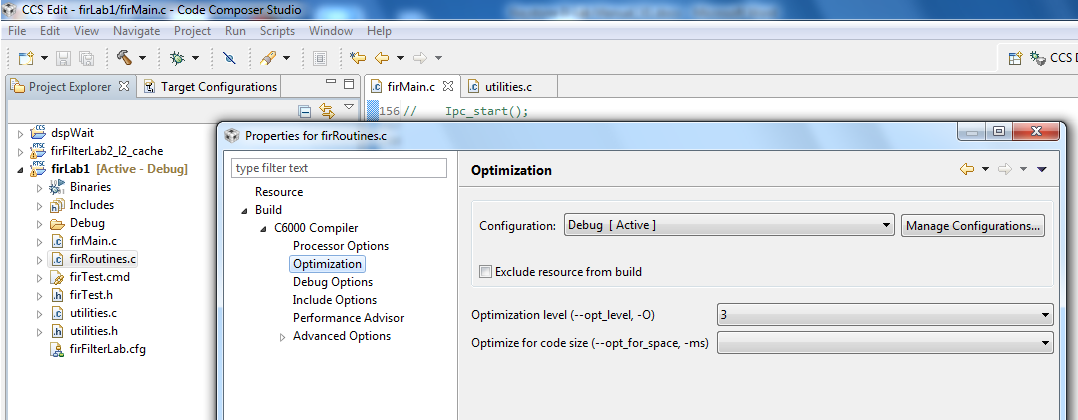
1. See Table 4-20 in the C66 CorePac User’s Guide <http://www.ti.com/lit/ug/sprugw0c/sprugw0c.pdf>

Look at the User Guide, the code, and the map file.w

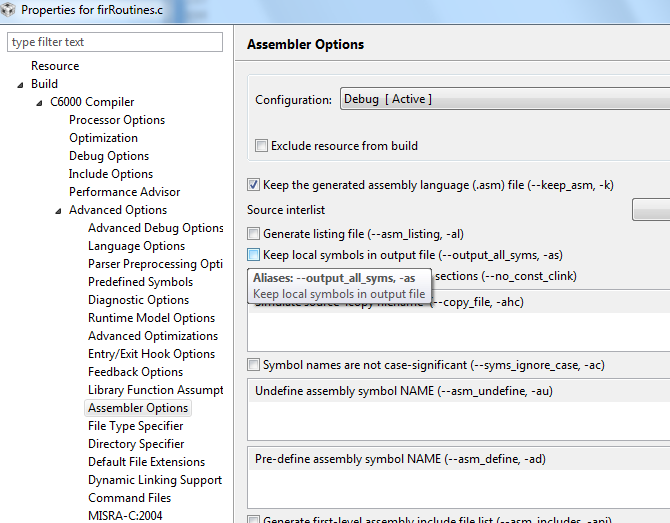
### Task 5: Use Optimization and Disable Symbol Debug for the fir Filter Routine

As the project is still in development/debug state, there is no optimization and full debug support. The next step is to optimize the fir filter and disable the debug information. However, leave the other parts of the project without optimization and with full debug support. The properties for the file firRoutines.c will be changed. No other file will be effected.

1. In the project explorer, select the file **firRoutines.c** and right click. Open the properties dialogue window as shown below.
2. Select *build->optimization*. In the dialogue window set optimization to 3
3. From the debug options dialogue select *suppress all symbolic debug generation* from the pull-down menu.



1. From the *build ->C6000 compiler -> Advanced Debug,* select *Assembly options* and check *Keep the generated assembly language (.asm)* file as shown in the screen shot below.



1. Click OK and rebuild the project. Load and run.
2. A successful run should produce a console output as shown below. Record the cycles time:

**Start generating input data**

**Finish generating input data**

**Case 1: Time consumed by core -> 0 500579008.000000**

**QUESTION:**

Is the code really optimized? Only 15% improvement.

1. Look at the assembly file firRoutines.asm in the debug directory and search for the function firRealFilter. Look for the loop and see if the compiler could get software pipeline

What is the reason that the loop does not qualify for software pipeline?

### Task 6: Optimize Software Pipeline

The reason why the fir filter loop is not qualified for a software pipeline is because it calls myMultiply. The next task is to inline this function. myMultiply is an artificial function (i.e., no one will develop this function in real code). So it is easy to “inline” it. Look at the definition of myMultiply in the utilities.c file and inline it.

1. Change the function firRealFilter by inline myMultiply function
2. Save and build the project. Load and run.
3. A successful run should produce a console output as shown below. Record the cycle time:

**start generating input data**

**finish generating input data**

**case 1 -> time consumed By core -> 0 273086080.000000**

1. Next, tell the compiler the minimum number times that each loop will be executed. The filter size in this program is 8. Assume that the filter size will always be more than 4 and divided by 4, so adding a pragma( **#pragma** MUST\_ITERATE(4,,4); ) will tell the compiler that the inner loop must be performed at least 4 times and the number of iterations is divided by 4.
2. The outer loop presents the size of the output vector. The number of elements is 16K, but eventually we would like to run it on all 8 cores, so each core will have about 2K element. It is enough if we tell the compiler that the number of elements is more than, say 64. However, if you look carefully, you will notice that the number of output results is 16K – filter size + 1, so this is an odd number. You can tell the compiler that the number of elements is more than 64. In that case use something like (**#pragma** MUST\_ITERATE(64,,1); ) or, if you agree to ignore the last fake result, you can tell the compiler (**#pragma** MUST\_ITERATE(4,,2); ))
3. Add the two pragma directives before the two loops (internal and external) in the function save and build.
4. If the external loop is **pragma** MUST\_ITERATE(64,,1)

**start generating input data**

**finish generating input data**

**case 1 -> time consumed By core -> 0 221407008.000000**

1. If the external loop is **pragma** MUST\_ITERATE(64,,2);

**start generating input data**

**finish generating input data**

**case 1 -> time consumed By core -> 0 221306848.000000**

**QUESTION:** To summarize the code optimization section, complete the following table:

|  |  |  |
| --- | --- | --- |
| **Optimization Technique** | **Cycles** | **Improvements compared with previous line** |
| No Optimization |  |  |
| Compiler optimization 3, no symbolic debug |  |  |
| Software Pipeline |  |  |
| Adding pragma must iterate |  |  |

### Task 7: Enable the Cache

Enabling the cache is done in CASE 2. Un-comment the line **#define CASE\_2** above the **main()** in **firMain.c**

**QUESTION:**

What instruction(s) enable the cache?

1. The function CACHE\_enableCaching ((Uint8) 128) ; was discussed in Task 4. The function CACHE\_setL2Size ((CACHE\_L2Size) 4); is part of the file csl\_cachAux.h in the \MCSDK\_3\_14\pdk\_keystone2\_3\_00\_02\_14\packages\ti\csl directory. Note, version number and location of MCSDK may be different for your setting.
2. Un-comment the line **#define CASE\_2** in **firMain.c**
3. Save, build, load and run. The results will look like the following:

**start generating input data**

**finish generating input data**

**case 1 -> time consumed By core -> 0 221223008.000000**

**case 2 -> time consumed By core -> 0 7491409.000000**

**QUESTION: Complete the table.**

|  |  |  |
| --- | --- | --- |
| **Optimization Technique** | **Cycles** | **Improvements compare with previous line** |
| No Optimization |  |  |
| Compiler optimization 3, no symbolic debug |  |  |
| Software Pipeline |  |  |
| Adding pragma must iterate |  |  |
| Enabling cache |  |  |

**QUESTION: What are the most important steps to optimize code running on a single core?**

### Task 8: Running in Parallel on Multiple Cores

Multiple cores are enabled in CASE 3 (2 cores), CASE 4 (4 cores) and CASE 5 (8 cores).

Un-comment the lines **#define CASE\_3 #define, CASE\_4** and **#define CASE\_5** above the **main()** in **firMain.c**

1. Un-comment the line #define CASE\_3 #define CASE\_4 #define CASE\_5 in firMain.c
2. Save, build, load and run. The results will be look like the following:

**finish generating input data**

**case 1 -> time consumed By core -> 0 288423616.000000**

**case 2 -> time consumed By core -> 0 7493824.000000**

**case 3 -> time consumed By core -> 0 3680093.000000**

**[C66xx\_1] case 3 -> time consumed By core -> 1 3678251.000000**

**[C66xx\_0] case 4 -> time consumed By core -> 0 1839643.000000**

**[C66xx\_1] case 4 -> time consumed By core -> 1 1838608.000000**

**[C66xx\_2] case 4 -> time consumed By core -> 2 1839438.000000**

**[C66xx\_3] case 4 -> time consumed By core -> 3 1836440.000000**

**[C66xx\_0] case 5 -> time consumed By core -> 0 918711.000000**

**[C66xx\_1] case 5 -> time consumed By core -> 1 921884.000000**

**[C66xx\_2] case 5 -> time consumed By core -> 2 921973.000000**

**[C66xx\_3] case 5 -> time consumed By core -> 3 920785.000000**

**[C66xx\_6] case 5 -> time consumed By core -> 6 922374.000000**

**[C66xx\_4] case 5 -> time consumed By core -> 4 923078.000000**

**[C66xx\_5] case 5 -> time consumed By core -> 5 921646.000000**

**[C66xx\_7] case 5 -> time consumed By core -> 7 920075.000000**

For each case, the total time that is consumed to perform the FIR filter is the maximum time of all the cores.

**QUESTION: Complete the table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Case** | **Cycles per core** | **Execution cycles (this is the cycles of the core with the highest cycles count)** | **Accumulate execution time for all the cores** | **Penalty of the accumulation execution time compared to single core (CASE 2)** |
| Case 2 – single core |  |  |  |  |
| Case 3 – 2 cores |  |  |  |  |
|  |
| Case 4 – 4 cores |  |  |  |  |
|  |
|  |
|  |
| Case 5 – 8 cores |  |  |  |  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**QUESTIONS:**

1. What is the purpose of the function **waitBarrier(barrier\_1, coreNum, jointNumber)**? What would happen if the function is commented out?
2. Try to comment out the function (3 places) and see what happens.
3. What is the purpose of the function **waitAboutNSeconds(10)** inside the function **waitBarrie**? What would happen if the function is commented out? Do you understand why?

1. Try to comment out the function and see what happens. Think about timing between cores.
2. Can you think about other (better) methods to synchronize the execution of all the cores?
3. Semaphores?, QMSS queues based solution?, openMP? .

# Lab 6: Load and Run DSP Code Using MPM Server

In this lab, you build a DSP project similar to the previous lab. Before starting, you should change the boot mode of the EVM to no-boot mode. Set SW1 of the EVM back to 1 Off 2 Off 3 On 4 Off.

Read the instructions in Lab 3 to ensure that the EVM boots using NFS-mounted file system.

## Purpose

Building a DSP code that is managed by the ARM. The ARM will reset C66x Core 0, load it with an executable, run it, and retrieve the results.

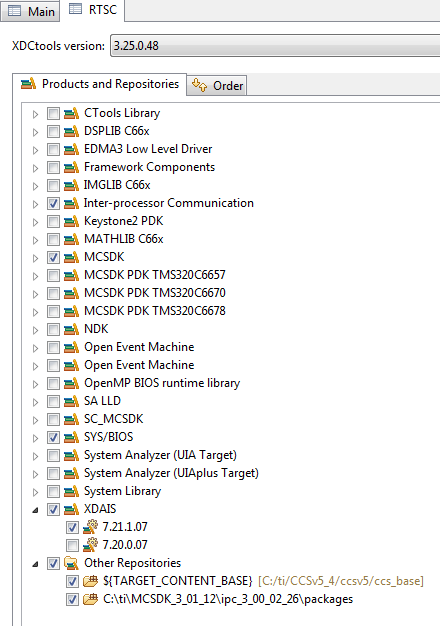
## Project Files

The following files are used in this lab:

* Main.c
* mpmsrv\_keystone2\_example1.cfg

### Task 1: Build and Run the Project

1. ftp into the Ubuntu server and get all the files that are in directory **/usr/local/projects/DSP/mpm** into a local directory **c:\ti\labs\mpm\_example** on your PC. If this directory does not exist, create it.
2. Open CCS.
3. Create new project through the CCS menu item *File* 🡪 *New* 🡪 *CCS Project*.
4. Enter *mpm\_example* as a *Project Name*.
5. Click the check box to *Use default location.*
6. Set the *Family to C6000* and *Variant* to *Generic C66xxx Device*
7. Then press *Finish* to create the new project
8. Then in the *Project Explorer* view, right-click on the newly-created *mpm\_example* project, and click on *Add Files…*
9. Browse to **C:\ti\labs\mpm\_example**, select all the files in this directory, and click *Open*. When prompted how files should be imported into the project, leave it as default of *Copy File.* If you defined the new project with **main.c**,remove the **main.c** file that may be created.
10. As soon as the file **mpmsrv\_keystone2\_example1.cfg** imported into the project, CCS will ask you to enable RTSC support, Select yes.
11. Open Project Properties and select general->RTSC. Look at the RTSC modules that are selected in the screen shot below and make sure that you select ONLY the same RTSC modules (or packages). When a project starts, RTSC attempts to include all the modules in the release. So unselect any module that is not in the screen shot.  
      
    NOTE: The TARGET CONTENT BASE should reflect the location of CCS on your system.



1. Click on the platform tab and select **ti.platform.evmTCI6638K2K**platform

NOTE: RTSC projects require the user to select three types of information.

* The device family in the CCS create page determines what core is used and thus what version of the compiler should be used (different cores have different intrinsic functions).
* The platform that is defined here determines the memory configuration of the core.
* To build the correct RTSC drivers, the device name should be defined. This is done by adding a predefined symbol with the device name. More about it later.

1. Right click on the project name and select *rebuild*. If the build goes correctly, you will see the following in the console window. NOTE: Look at the debug directory to ensure that the file **MPM\_example.out** is there. Ignore any warnings.

**'Building target: MPM\_example.out'**

**'Invoking: C6000 Linker'**

**"C:/ti/CCSv5\_4/ccsv5/tools/compiler/c6000\_7.4.2/bin/cl6x" -mv6600 --abi=eabi -g --display\_error\_number --diag\_warning=225 --diag\_wrap=off -z -m"MPM\_example.map" -i"C:/ti/CCSv5\_4/ccsv5/tools/compiler/c6000\_7.4.2/lib" -i"C:/ti/CCSv5\_4/ccsv5/tools/compiler/c6000\_7.4.2/include" --reread\_libs --warn\_sections --display\_error\_number --diag\_wrap=off --xml\_link\_info="MPM\_example\_linkInfo.xml" --rom\_model -o "MPM\_example.out" -l"./configPkg/linker.cmd" "./main.obj" -l"libc.a"**

**<Linking>**

**'Finished building target: MPM\_example.out'**

**'**

**\*\*\*\* Build Finished \*\*\*\***

### Task 2: Using MPM to load, run and observe results

In this part, we assume that the EVM is boot in net mode, that is, the file system is on the server and it is mounted to the EVM as used in Lab3

1. FTP the out file into the server to **/opt/filesys/studentN/bin** where N is the student number.
2. Reboot the EVM using NFS.
3. From the terminal login as **root**
4. **cd /bin**
5. Use MPM to reset, load, and run core 0 with **MPM\_example.out** by using the following MPM commands:

**mpmcl reset dsp0**

**mpmcl load dsp0 MPM\_example.out**

**mpmcl run dsp0**

1. After the end of run look at the trace buffer printing by using the following command

**cat /debug/remoteproc/remoteproc0/trace0**

1. Change the main.c file as you wish, build it again, ftp to the file system (step 7) load the code to a different dsp (use N here) and run it:

**mpmcl reset dspN**

**mpmcl load dspN MPM\_example.out**

**mpmcl run dspN**

1. After the end of run look at the trace buffer printing by using the following command:

**cat /debug/remoteproc/remoteprocN/trace0**

# 

# Lab 6: ARM-DSP Communication Using MPM & Shared DDR

In this lab, you build a DSP project similar to Lab 4 that uses the DDR. Unlike the previous Lab where the code and the data were only in L2, in this lab some DDR is used by the DSP.

## Purpose

Building a DSP code that uses the DDR and is managed by the ARM.

## Linux and DSP simple memory management

The previous project uses private L2 memory for program and data. This DSP project uses DDR. How does the system manage the DDR resources between the DSP and the ARM?

The Linux uses part of the DD. So if a DSP program uses some of the DDR, it must tell the Linux. This is done in the U-BOOT environment.

To do it correctly, the user must follow the following steps:

1. Stop autoboot and look at the messages from the U-BOOT. It looks like the following:

**I2C: ready**

**Detected SO-DIMM []**

**DRAM: 1 GiB**

**NAND: 512 MiB**

**Net: TCI6638\_EMAC, TCI6638\_EMAC1**

**Hit any key to stop autoboot: 0**

The size of the DRAM is 1 GiB in this case. It can be 2G or 8 G depends on the EVM revision

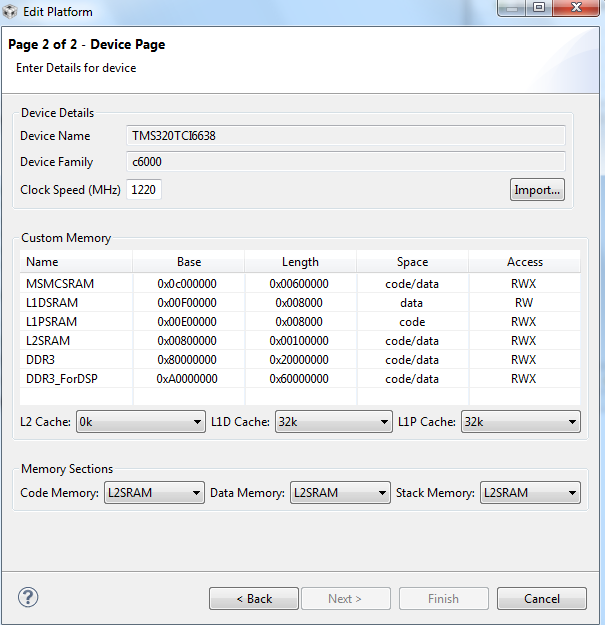
1. Now determine how much DRAM the DSP needs. Obviously it must be less than the total DRAM in the EVM. Assume that Linux uses 512M and the DSP will use 512M. The user must tell the U-BOOT that 512M is reserved for the DSP.
2. After stopping the autoboot, configure the memory that is assigned to the DSP (if it is not configured already).

**setenv mem\_reserve 512M** (For EVM with 2G DRAM it can be 1536M  
 or even larger if the DRAM is more than 2G)

**saveenv and boot**

1. The memory that is reserved for the DSP is located at the end of the available memory. For the 1G DRAM case, available memory is between 0x80000000 and 0xbfffffff, so the 512M reserved for the DSP start at address 0xa0000000 to address 0xbfffffff
2. Next, you need to build the DSP code and ensure that it uses only the assigned DDR. One way to do this when using RTSC is to re-define the platform. The release platform defines DDR starting at 0x80000000 and with size 0x80000000.
3. To modify the platform in CCS, go to the debug perspective *tools -> RTSC Tools ->platform -> edit/view*. Make sure that the repository is the location of XDCTools/packages (in MCSDK\_3\_14 setting it is in *“MCSDK\_3\_14\xdctools\_3\_25\_02\_70\packages”* and click on package name. Choose the package that corresponds to the EVM, (the current EVM is *ti.platforms.evmtci6638K2K*), CPU core is CPU, and click NEXT.

Changing the platform as illustrated in the following screen shot.



1. In a link command file XXX.cmd, make sure that all DDR sections are assigned to the memory DDR3\_ForDSP (in the case above). Verify the correct address in the map file after rebuilding the project.
2. As was mentioned before, the DSP in this lab does not use the DDR. So for this lab, the above procedure is not necessary. However, it can be used as reference for more complex projects.

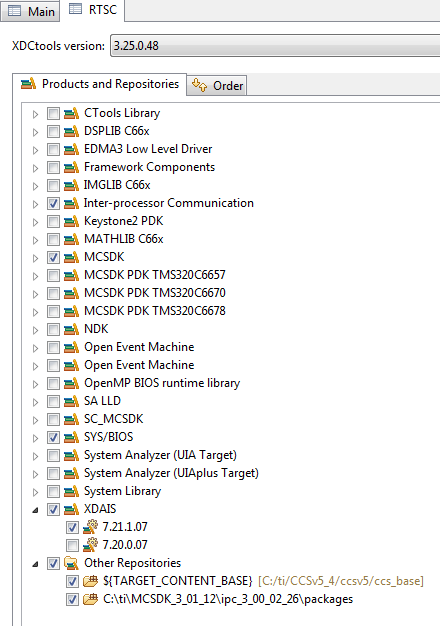
## Project Files

The following files are used in this lab:

* Main.c
* mpmsrv\_keystone2\_example1.cfg

### Task 1: Build and Run the Project

1. FTP into the Ubuntu server and get all the files that are in directory /usr/local/projects/DSP/mpm into a local directory on your Laptop c;\ti\labs\mpm\_example. If this directory does not exist, create it
2. Open CCS.
3. Create new project through the CCS menu item *File* 🡪 *New* 🡪 *CCS Project*.
4. Enter *mpm\_example* as a *Project Name*.
5. Click the check box to *Use default location.*
6. Set the *Family to C6000* and *Variant* to *Generic C66xxx Device*
7. Then press *Finish* to create the new project
8. Then in the *Project Explorer* view, right-click on the newly-created *mpm\_example* project, and click on *Add Files…*
9. Browse to ‘C:\ti\labs\mpm\_example,’ select all the files in this directory, and click *Open*. When prompted how files should be imported into the project, leave it as default of *Copy File.* If you defined the new project with main.cremove the main.c file that may be created.
10. As soon as the file mpmsrv\_keystone2\_example1.cfg imported into the project, CCS will ask you to enable RTSC support, Select yes.
11. Open Project Properties and select general->RTSC. Look at the RTSC modules that are selected in the screen shot below and make sure that you select ONLY the same RTSC modules (or packages). When a project starts, RTSC attempts to include all the modules in the release. So unselect any module that is not in the screen shot.  
      
    NOTE: The TARGET CONTENT BASE should reflect the location of CCS on your system.



1. Click on the platform tab and select ***ti.platform.evmTCI6638K2K*** platform

NOTE: RTSC projects require the user to select three types of information.

* The device family in the CCS create page determines what core is used and thus what version of the compiler should be used (different cores have different intrinsic functions).
* The platform that is defined here determines the memory configuration of the core.
* To build the correct RTSC drivers, the device name should be defined. This is done by adding a predefined symbol with the device name. More about it later.

1. Right click on the project name and select *rebuild*. If the build goes correctly, you will see the following in the console window.  
     
   NOTE: Look at the debug directory to ensure that the file **firLab1.out** is there. Ignore any warnings.

**'Building target: MPM\_example.out'**

**'Invoking: C6000 Linker'**

**"C:/ti/CCSv5\_4/ccsv5/tools/compiler/c6000\_7.4.2/bin/cl6x" -mv6600 --abi=eabi -g --display\_error\_number --diag\_warning=225 --diag\_wrap=off -z -m"MPM\_example.map" -i"C:/ti/CCSv5\_4/ccsv5/tools/compiler/c6000\_7.4.2/lib" -i"C:/ti/CCSv5\_4/ccsv5/tools/compiler/c6000\_7.4.2/include" --reread\_libs --warn\_sections --display\_error\_number --diag\_wrap=off --xml\_link\_info="MPM\_example\_linkInfo.xml" --rom\_model -o "MPM\_example.out" -l"./configPkg/linker.cmd" "./main.obj" -l"libc.a"**

**<Linking>**

**'Finished building target: MPM\_example.out'**

**'**

**\*\*\*\* Build Finished \*\*\*\***

# Lab 7: Build and Run ARM Code Using CCS on Linux

## Purpose

The goal of this lab is to demonstrate some basic operations using CCS on ARM code as preparation for EVM debug from CCS.

## Prerequisites

CCS must be installed on the Linux machine.

## Task 1: CCS - Start a new cross compiler project and build it

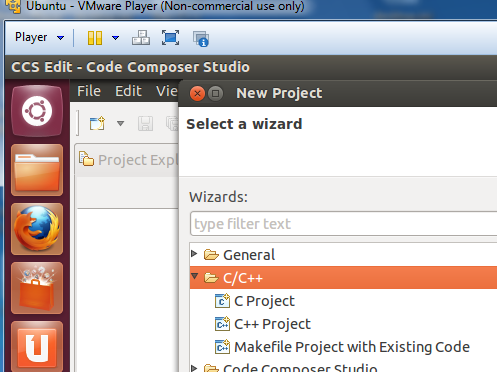
1. To start CCS, go to CCS eclipse directory and enter the following:

**sudo ./ccstudio**

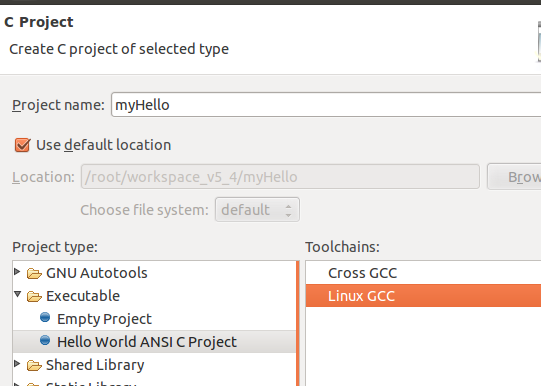
1. After CCS start, and in the edit perspective, start a new (not CCS) project:

*File -> New -> Project* (Not CCS Project)

*C/C++* and then *C project*



1. From the C project, choose Hello World ANCI C project, Toolchains Linux GCC, give the project a name (myHello), and finish.



1. Before continuing, to make the Lab a little more interesting, you can change the source code using CCS editor. The following is a version of myHello.

**int** **main**(**void**)

{

**int** i, j, k, l ;

**printf**("printf command start the program \n") ;

**puts** (" puts command start the program \n") ;

l = 12 ;

**for** (i=0; i<1000; i++)

{

j = l + i ;

k = 2 \* j ;

l = k + j ;

**if** (l > 10000)

{

**printf**(" inside loop, l > 10000 i= %d k = %d j = %d l = %d \n", i,k,j,l) ;

l = l - 10000 ;

**printf**(" after the normalization i= %d k = %d j = %d l = %d \n", i,k,j,l) ;

}

}

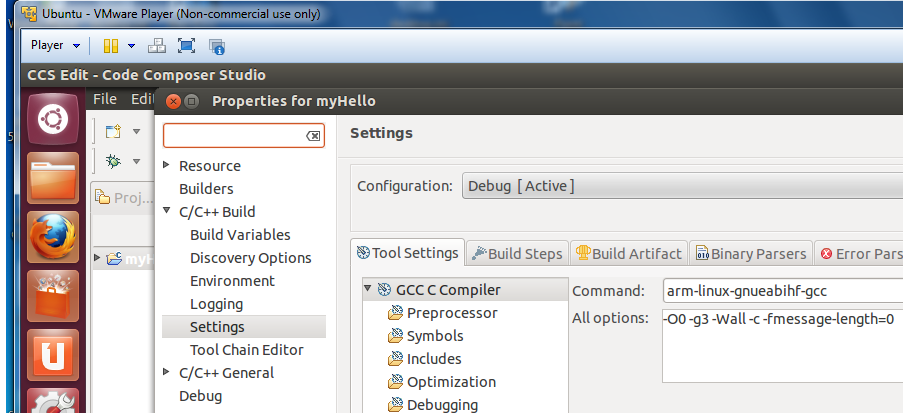
**printf**(" END ->>> i= %d k = %d j = %d l = %d \n", i,k,j,l) ;

**puts**("!!!Hello World!!!"); /\* prints !!!Hello World!!! \*/

**return** 1;

}

1. Next configure the project properties. Right click on the project and start properties. Open C/C++ build and open settings. In the dialogue window select GCC C compiler and enter arm-linux-gnueabihf-gcc. The same tool will be set for the linker as well.



For assembly, choose the arm-linux-gnueabihf-as assembler.

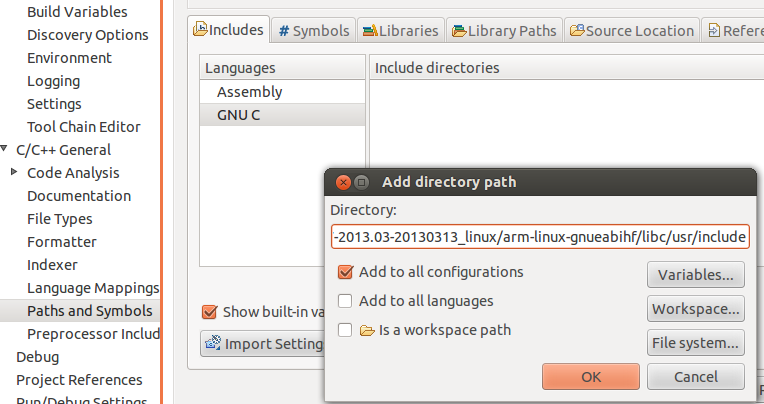
Next the include path to the tools should be configured. From project properties select

C/C++ General->Path and Symbols

Note: Depending on project requirements, more paths may be needed  
  
 From ***Includes*** tab Add GNU C Includes. The easier way is by going to ADD and then file system and browse:  
 DirectoryWhereTheToolsAre/gcc-linaro-arm-linux-gnueabihf-4.7-2013.03-20130313\_linux/arm-linux-gnueabihf/libc/usr/include  
 DirectoryWhereTheToolsAre /gcc-linaro-arm-linux-gnueabihf-4.7-2013.03-20130313\_linux/arm-linux-gnueabihf/libc/usr/include/arm-linux-gnueabihf  
 - ***Add to all configurations***

From ***Library Paths*** tab Add Library Paths

DirectoryWhereTheToolsAre /gcc-linaro-arm-linux-gnueabihf-4.7-2013.03-20130313\_linux/arm-linux-gnueabihf/libc/usr/lib/arm-linux-gnueabihf  
 - ***Add to all configurations***



Last the path to the tools should be configured. From project properties select

1. Right-click Project, select ***Properties***
2. Select ***C/C++ Build***🡪***Environment***
3. Click Add…

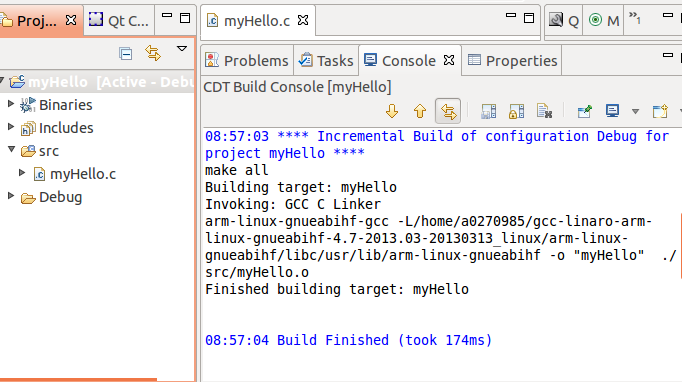
Enter Name: **PATH**, ***Add to all configurations***

Press ***OK***  (ccs will automatically populate with internal path)

1. Select **PATH**, click ***Edit…***
2. Prefix string with path to toolchain binaries  
    (e.g. /opt/gcc-linaro-arm-linux-gnueabihf-4.7-2013.03-20130313\_linux/bin)
3. Path must be delimited with a colon ‘:’

Do clean build and build

The consol should look like the following:



## Task 2: GDB from the tera-terminal

There are multiple ways to use gdb to run and debug myHello. It can be done outside of CCS or within CCS.

### Move the Debug directory outside of CCS

* First allocate where the program is located in the linux host system. From property -> resources you will see something like:

**/root/workspace\_v5\_4/myHello**

Where workspace\_v5\_4 is the default workspace for CCS.

* An alternative method is to search for the myHello project:

cd / takes us to the start of file system

**sudo find . –name myHello –print** (look for all files with the name myHello in them)

* Go to the myHello directory and change the permissions on the Debug sub-directory

**sudo chmod 777 Debug**

* move the Debug directory to the EVM file system (if the default setting was used, it is in /opt/filesys)

**cd /opt/filesys**

**cd bin**

**sudo cp –R /root/workspace\_v5\_4/myHello/Debug .**

* From the tera-terminal go to the bin subdirectory and look at the Debug subdirectory. The src sun-directory should have the source code.
* Start the gdb from the tera-terminal

**gdb myHello**

**display the source code**

**list**

* Put a breakpoint after the printf

**b 17**

* run the code and observe the printing on the terminal

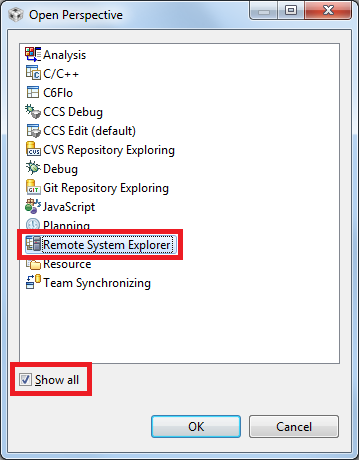
r

c

You can play with other gdb instructions.

## Task 3: Connect a SSH terminal into the target

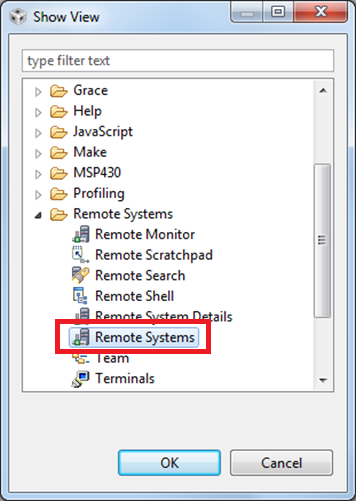
1. After reboot and login the target from the tera-terminal, do ifconfig eth0 and see what the IP address of the target is. Note, the IP address is assigned by the DHCP server.
2. Start CCS; Select ***Window🡪Open Perspective🡪Other***  
   Open Perspective: Check ***Show All***, Select ***Remote System Explorer***  
   Allow enablement of “Remote System Explorer Tools”



1. Select ***CCS Edit*** perspective



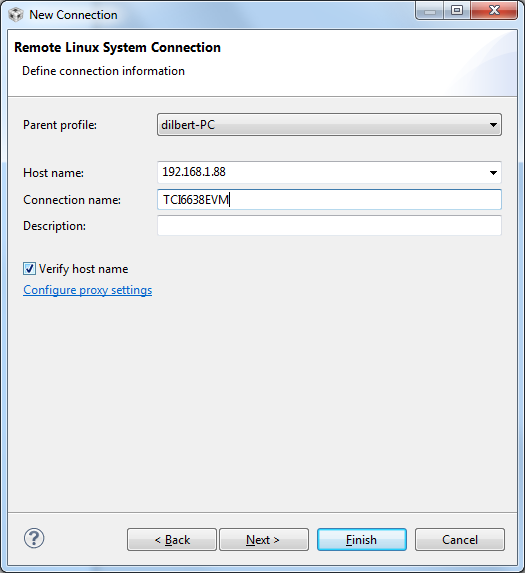
1. Select ***Window🡪Show View🡪Other***  
   Expand Remote Systems, Select ***Remote Systems***  
   Optionally drag Remote Systems to different location  
   (next to Project Explorer tab is a nice spot…)



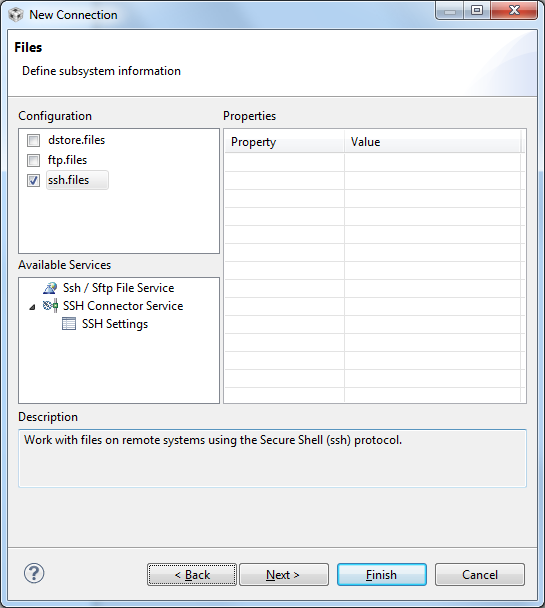
1. From Remote Systems window, Right-Click🡪***New🡪Connection…*** in the window, select linux and click next



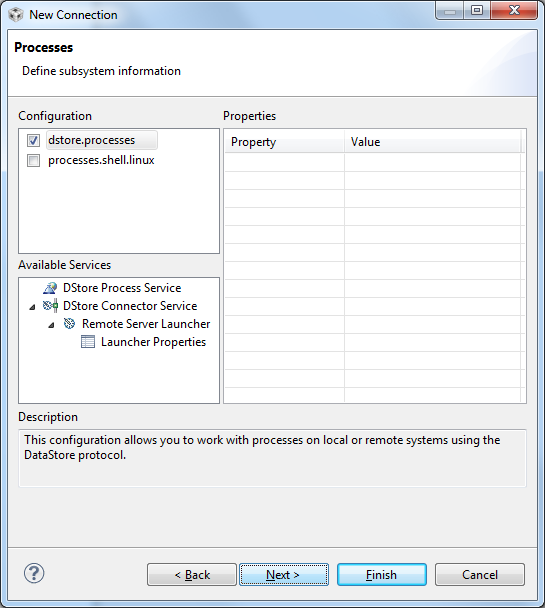
1. Configure the remote linux connection, enter the IP address as the host name, give connection name and profile name:



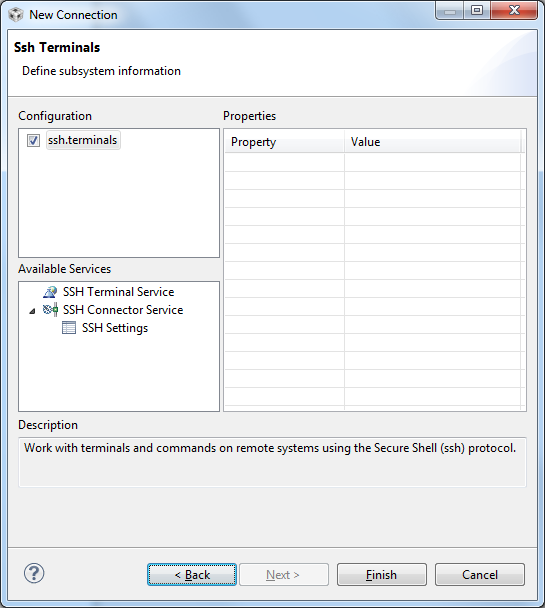
1. Files, select ssh.files



1. Click Next
2. Select processes.shell.linux

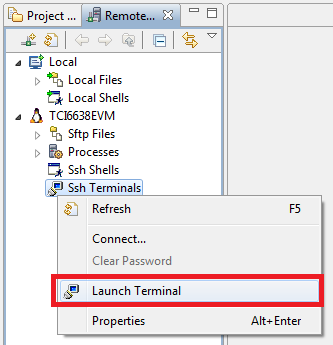


1. Click Next
2. Select ssh.shells
3. Click Next

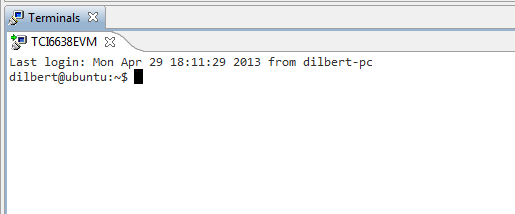


1. Select ssh.terminals
2. Finish
3. Opening a terminal:  
    Right-Click, Select ***Launch Terminal***  
    Enter password (for root user name leave the password blank)

After warnings a terminal will be opened into the target ARM



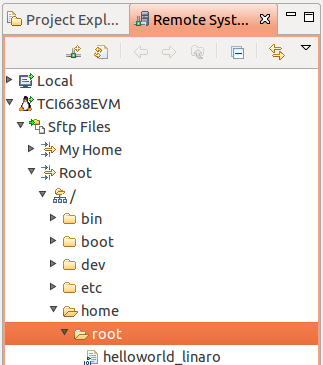
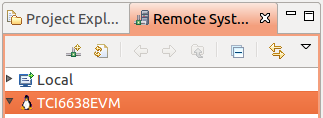
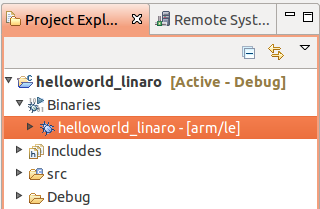




Note – the green arrow indicates that the connection is good. If there is not green arrow, and a proxy server is involved, there may be an additional step required to configure the proxy. Instructions how to configure the proxy are at the end of the Lab. (Task 5)

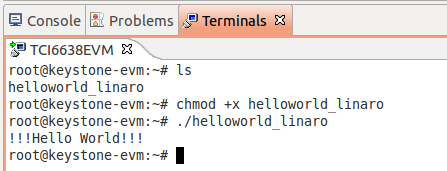
1. Copy and paste binaries onto the target

Select binary file and the source file from project  
Press ***Ctrl-C*** (or right-click, Copy)  
 Click on ***Remote Systems*** tab  
Navigate through the ftp Files tree to desired path  
Press ***Ctrl-V*** (or right-click, Paste)



1. Executing binary on target from CCS RSE Terminal window

1. From the ***Remote Systems*** tab, right-click ***Ssh Terminals***  
 Select ***Launch Terminal***  
2. For newly transferred files, give them executable  
 permissions:  
 chmod 777 ./myHello  
  
3. Execute application  
 ./myHello



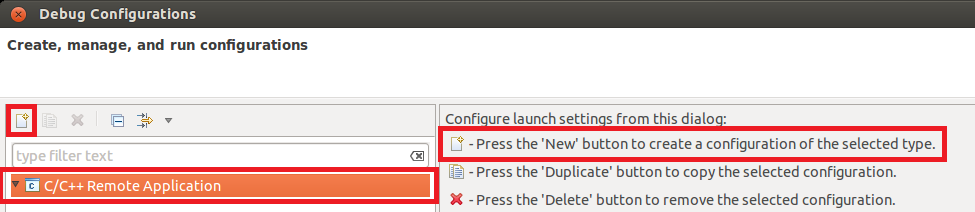
## Task 4: Optional – Use gdb to debug the application

In the edit prospective, select the project myHello

Select ***Run***🡪***Debug configurations*** from file menu

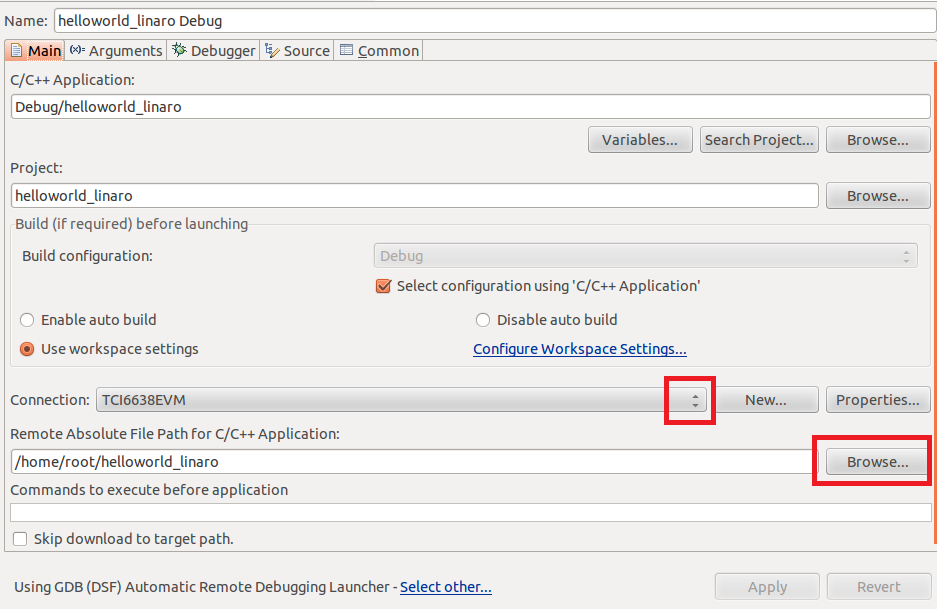
Select ***C/C++ Remote Application***

Click ***New*** button



Main tab:

* + Connection: select EVM [this is configured in WS1]
  + Specify Remote Absolute File Path for C/C++ Application (for example /bin/myHello if you pasted myHello in the bin directory)



. ***Debugger*** tab:

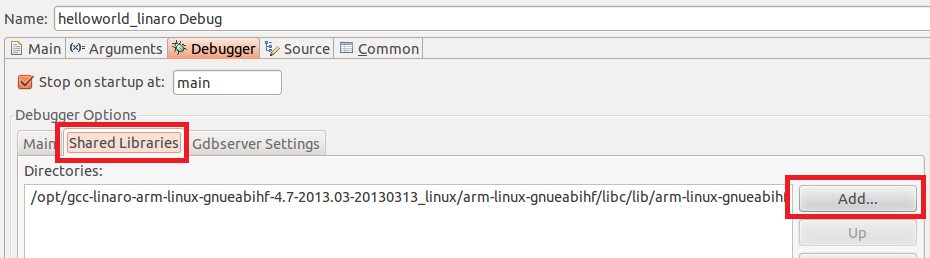
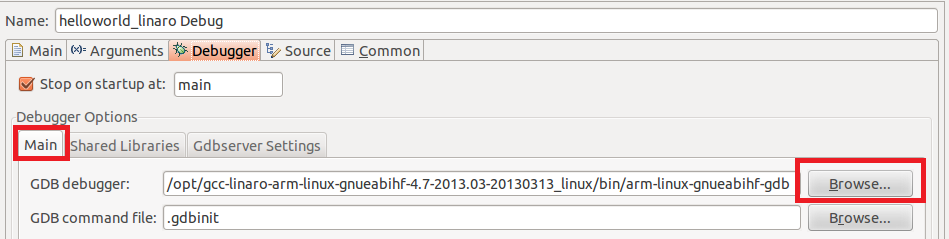
***Main*** sub-tab:

* + GDB debugger🡪***Browse…***:  
     opt/gcc-linaro-arm-linux-gnueabihf-4.7-2013.03-20130313\_linux/bin/arm-linux-gnueabihf-gdb

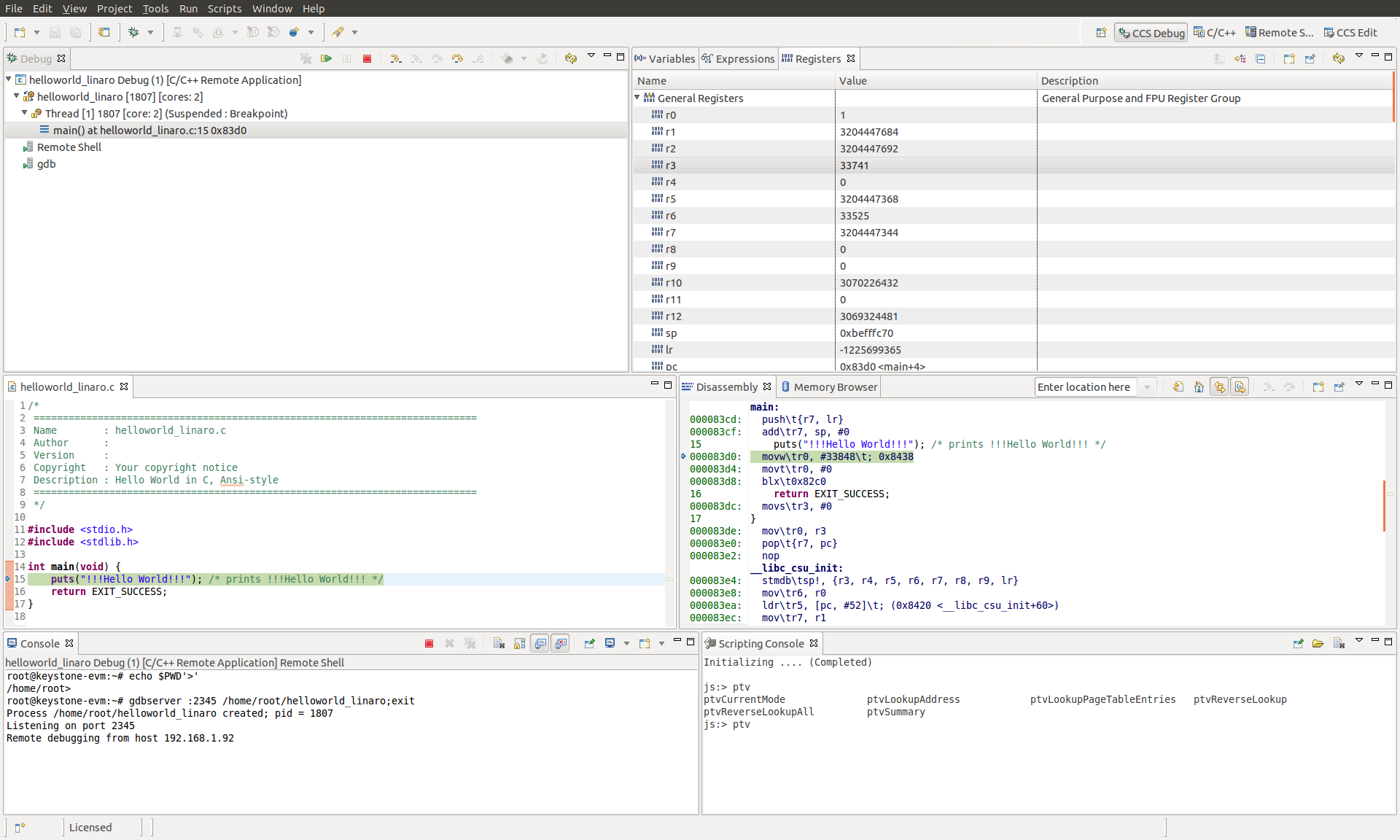
***Shared Libraries*** sub-tab:

* + ***Add…***  
    opt/gcc-linaro-arm-linux-gnueabihf-4.7-2013.03-20130313\_linux/arm-linux-gnueabihf/libc/lib/arm-linux-gnueabihf

Press ***Apply***, ***Debug***



Using the debug in the debug prospective:

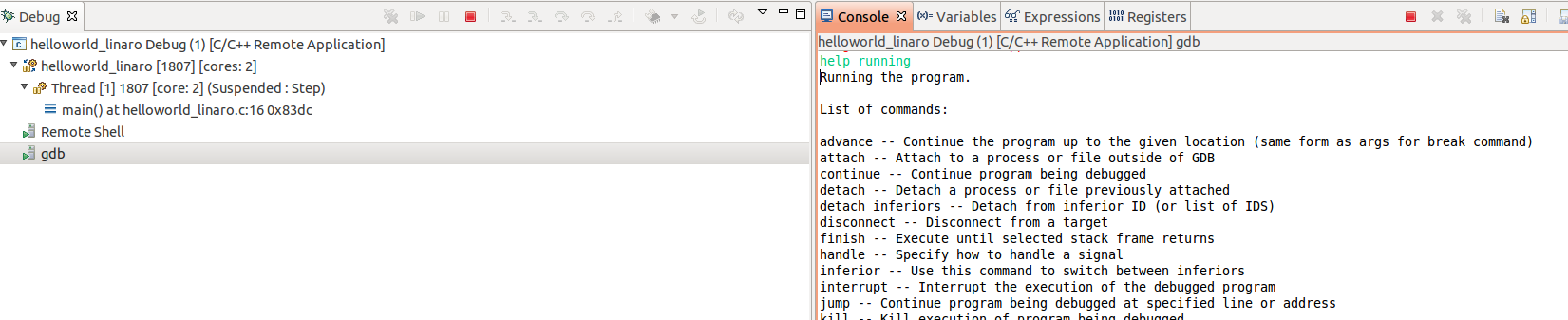


Using the debugger – one step into program:



Using the debugger – gdb console

Selecting gdb from the Debug panel allows direct control of gdb



Select the gdb and from the consol you can use almost all gdb commands, breakpoint, continue and so on.

Note – I was unable to move the pc backwards, or do reset or reload. If I want to go backwards I had to close the remote debug terminal and open it again. Obviously the r (run) command does not work (run can start execution from the beginning of the execution)

## Task 5: Configure the host proxy

CCS defaults to using the system proxy settings (should not be needed if the target and the host are on the same subnet).

Right-click on the connection, select ***Properties***

From the left pane, select ***Host.*** Click ***Configure proxy settings***

From ***Network Connections,*** set ***Active Provider*** to ***Direct,*** click ***Apply*** and ***OK***

# Lab 8: ARM Optimization Using SMP Linux

## Projects and source code

Unless instructed by the instructor otherwise, all projects and source code are available on the server. Directory /usr/local/projects has two sub-directories, ARM and DSP. The source for this ARM sub-directory is in the ARM subdirectory in a subdirectory called SMP.

## Purpose

The purpose of this lab is to demonstrate how the SMP LINUX distributes threads between multiple cores and as a result speed up the processing of time sensitive application running on the 4 A15 of KeyStone II.

The default application is a typical signal processing fir filter algorithm. Fir filters can be easily partition between multiple threads. The program was structured such that it is very easy to replace the fir filter with any generic easy to partition application.

## Task 1: Copy and observe the source files

It is assumed that the file system is mounted to the EVM (NFS boot, setenv boot net) and that the file system is in location /opt/filesys/studentN where N is the student number.

From the VNC window log-in as your student name (user name studentN, password WsN where N is the student number) and run the initialization script file

* Source /usr/local/staudentStartScript.sh

In the studentN file system location make a new directory (if it does not exist already) and name it applications and then make a subdirectory smp\_test

* cd /opt/filesys/studentN
* sudo mkdir applications
* cd applications
* sudo mkdir smp\_test
* cd smp\_test

Note – if the filesystem is located in a different directory change the instructions accordingly. For example, if the file system is in directory /opt/filesys/studentN/mcsdk\_3\_14 then the sequence of instruction is

* cd /opt/filesys/studentN/mcsdk\_3\_14
* sudo mkdir applications
* cd applications
* sudo mkdir smp\_test
* cd smp\_test

Next you copy the four source files to the new directory:

* sudo cp /usr/local/projects/SMP/smp\_test.c .
* sudo cp /usr/local/projects/SMP/multithreads.h .
* sudo cp /usr/local/projects/SMP/application.c .
* sudo cp /usr/local/projects/SMP/application.h .

Last copy the three Makefiles to the new directory

* sudo cp /usr/local/projects/SMP/Makefile\_no\_optimization .
* sudo cp /usr/local/projects/SMP/Makefile\_O2\_optimization .
* sudo cp /usr/local/projects/SMP/Makefile\_full\_optimization .

Question

Assume you want or need to change the algorithm that runs on the A15.

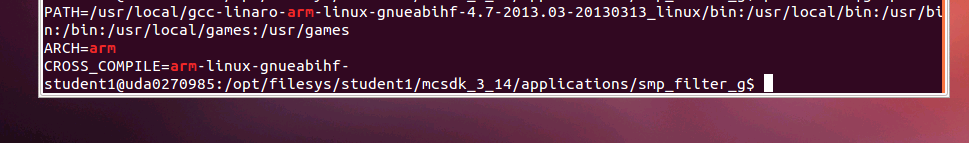
1. Does the file smp\_test.c need to be changed?
2. Does the include file multithreads.h needs to be changed?
3. Does the file application.c need to be changed or replaced?
4. Does the file application.h need to be changed or replaced?
5. Which instruction spans threads?
6. Optional – what does each of the parameters of the clone () function represent?

## Task 2: Compile and build the project, run it on a single core

From the VNC terminal you need to build the project. The path to the cross compiler should be define. To verify this do the following:

* sudo printenv |grep arm

The printing on the terminal will look like the following.



Make sure that the path to arm-linux-gnueabihf is defined and the arm-linux-gnueabihf is defined as the CROSS\_COMPIER. If not, run the initialization script:

* Source /usr/local/staudentStartScript.sh

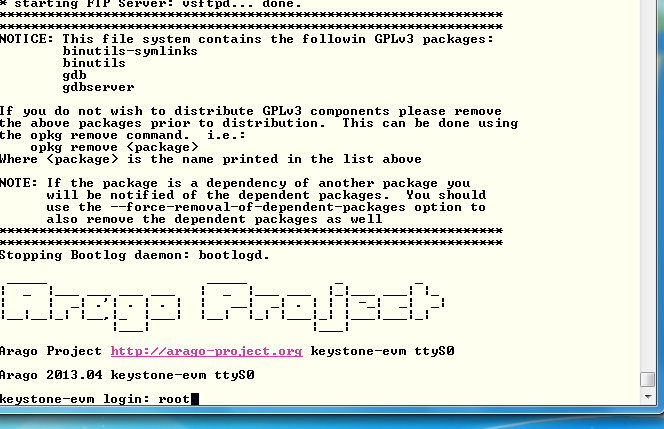
First build the project with full debug and no optimization:

* Make –f makefile\_No\_Optimization

Note that the optimization flag (dash capital O) is set to zero, and there is no –o (small o) thus the built project will have the default name a.out.

Before running the code you need to verify that the EVM is connected to the local network via Ethernet, which you have a terminal window (either Putty or Tera Term or other) into the EVM as explained in previous Labs, and that NSF boot from TFTP is working.

1. Power on the EVM
2. Wait for the login screen to come
3. Log as root:



1. Change the location of the terminal to the smp\_test directory:

* cd /applications/smp\_test
* ls

The 4 source files, the three Make files and the a.out file should be in the directory.

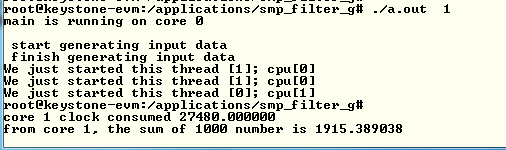
1. To run the code with a single thread first ensure that the file a.out has executable permission

* chmod +x a.out

1. Run the code

* ./a.out 1

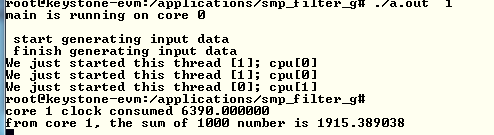
1. After a few seconds (wait until all the printing is done) the results should look similar to the follows:



1. Record the clock consumed value in the table below
2. Back in the VNC window re-build the executable with O2 optimization:

* Make –f makefile\_No\_Optimization

1. And run a.out again. The results should look like the following:



1. Back in the VNC window re-build the executable with full optimization:

* Make –f makefile\_full\_Optimization

1. And run a.out again.

Question

1. What is the speed up percentage of performance improvements when the optimization O2 is on?
2. What is the speed up percentage of performance improvements when the full optimization is on? - Compare to non-optimization, compare to –O2

## Task 3: Run the code on multiple cores

From this point on you only run the full optimized version of the a.out executable.

In this task you run the program on multiple cores. While the total processing time on all the cores remains almost the same, the elapsed time (the time that the slowest core consumes) will be reduced almost linearly by the number of cores that are involved.

The SMP operating system will distribute threads between cores. The number of thread in this program is limited to 32, but the number of cores is 4. So if the number of threads is bigger than 4, multiple threads will be assigned to each core.

Run the following cases and fill the table below:

* ./a.out 1
* ./a.out 2
* ./a.out 4
* ./a.out 8
* ./a.out 16
* ./a.out 32

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Number of threads | Core 0 total time consumed | Core 1 total time consumed | Core 2 total time consumed | Core 3 total time consumed | Slowest core time consumed | Total time consumed by the 4 cores |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

Question

1. The shared ARM L2 cache is 4MB. Are all the vectors fit into the cache?
2. The private L1 cache for each A15 is 32KB. What is the maximum input vector that will not cause L1 cache to be trashed?
3. What is the speed up when 4 cores are used compare to a single core?
4. How does the cache size effect the total time when multiple threads are used?

# Lab 9 – Inter-Processor Communication (IPC) Lab

## Projects and source code

The original files for this Lab are part of the MCSDK release. The student will copy MCSDK release into his private directory (studentN/MCSDK\_X\_XX) before changing any file.

## Purpose

The purpose of this lab is to demonstrate messages transfer between the ARM and the DSP cores. The source code may be a starting point for customer who needs sending messages and data between cores.

## Task 1: Run the demo from a Web Server

## Step 1

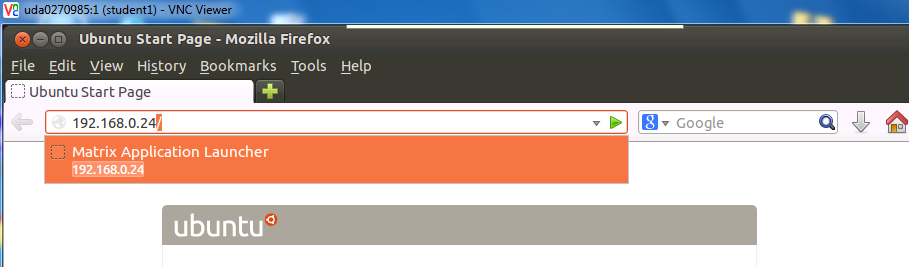
The file system that is loaded into the EVM should be tisdk based file system. Boot the EVM using NFS (mount) boot and wait until the display on the EVM gives the IP address of the board. Note that the display flips between several messages. IP address is one of the messages.

## Step 2

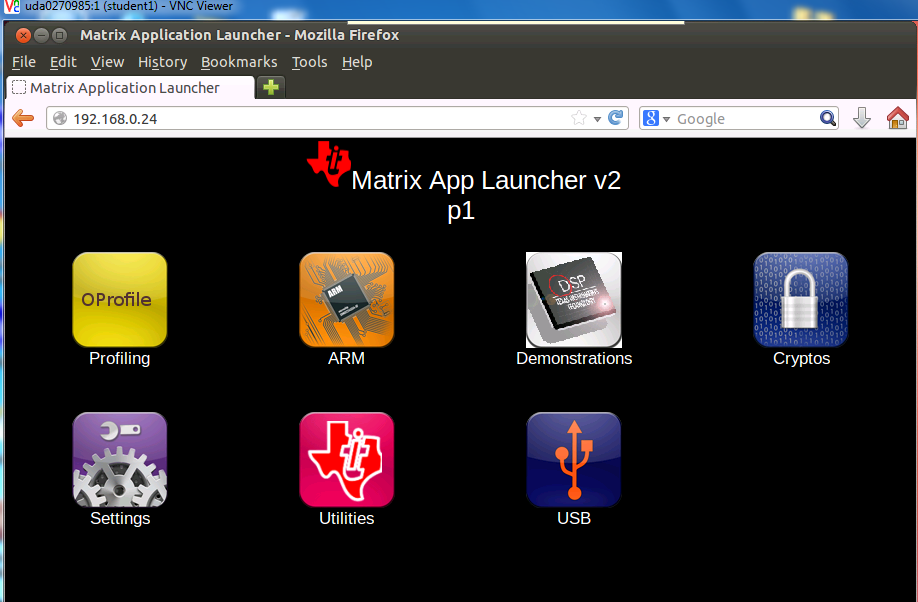
Open a terminal into the EVM and log in as root

## Step 3

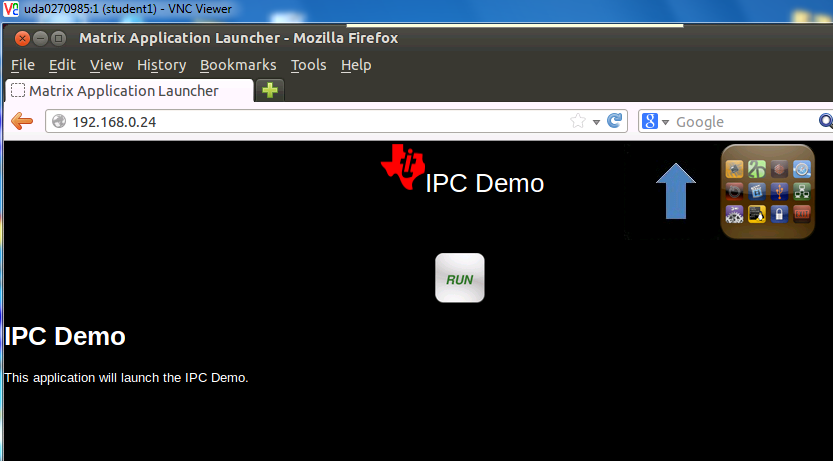
From a computer that is on the same sub-net as the EVM, or from vnc into a computer that is on the same subnet as the EVM start Firefox (or any other browser) and put the ip address of the EVM as shown in the following screen shot. The EVM in the screen shot has ip address of 192.168.0.24.



If the Firefox is connected via vnc, Ubuntu may ask you if you have a display device. Answer OK. The EVM respond with a set of out-of-the-box applications as seen in the next screen:



It is highly recommend going through each one of the applications, however, in this Lab we only use the IPC demo. Click on the Demonstrations tab and then the IPC Demo. The next screen shot will be displayed:

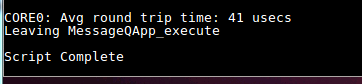


Click on RUN and follow the progress on the browser and on the terminal that is connected to the EVM.

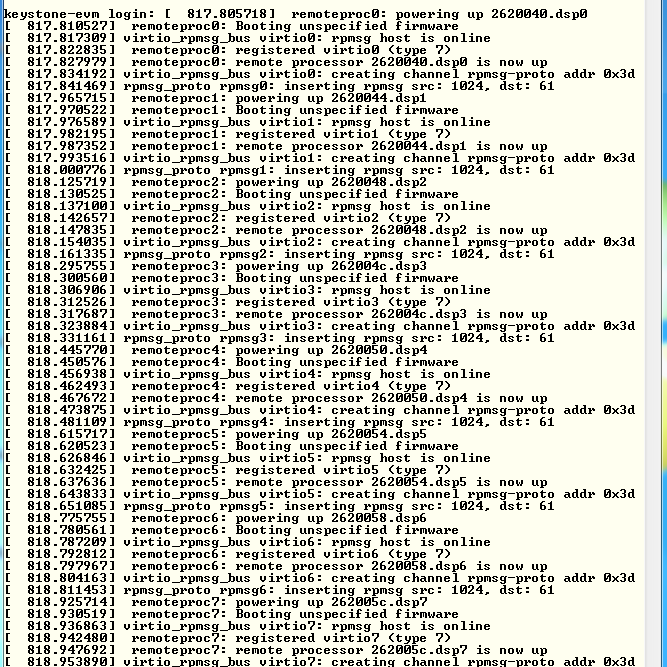
The browser display



You may have to use the arrow to see the complete execution of the demo. The last lines are the following:



The terminal that is connected to the EVM displays something like:



Follow the messages on the terminal and see what software modules are used (remoteproc, virtio, rpmsg)

## Task 2: Run the demo from the terminal

### Purpose

The purpose of this task is to familiar the user with the directory structure and the main files of the demo

### Step 1 – Understand the file demo\_ipc.sh

In the directory /usr/share/matrix-gui-2.0/apps/demo\_ipc there are three files demo\_ipc.desktop, demo\_ipc.sh, and desc\_demo\_ipc.html. Look at the file demo\_ipc.sh using vi, more, less, cat or any other utility. The screen shot was taken using cat:



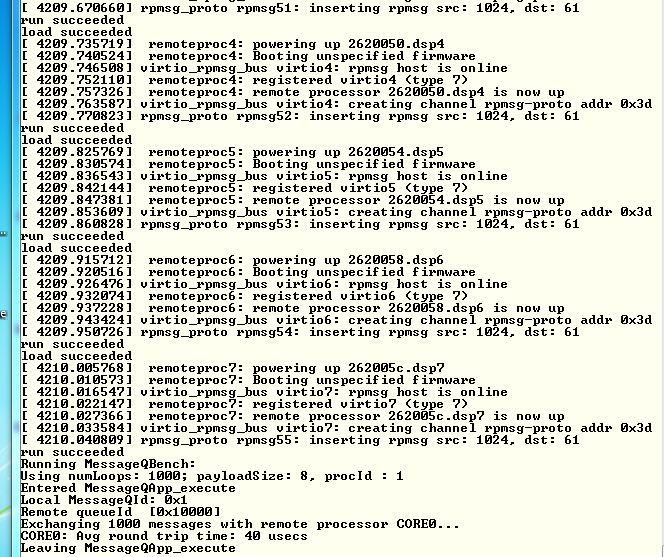
Notice that the DSP code is loaded from directory /lib/firmware and the name of the execution is message\_single.xe66. The Linux code is loads from directory /usr/bin and the executable name is MessageQBench. In the next task you will build these two executable files in your local directory and move them to your private filesystem

### Step 2 – Run the file demo\_ipc.sh

In the terminal, move to directory /usr/share/matrix-gui-2.0/apps/demo\_ipc (cd /usr/share/matrix-gui-2.0/apps/demo\_ipc ) and run the sh file ./demo\_ipc.sh as in the following screen shots:



After the run the terminal looks like the following:



## Task 3: Rebuild the Executable

### Purpose

In this task the student will re-build the DSP and ARM executable to prepare for modifying the code for a new project

### Step 1 – Get a private copy of IPC from the release

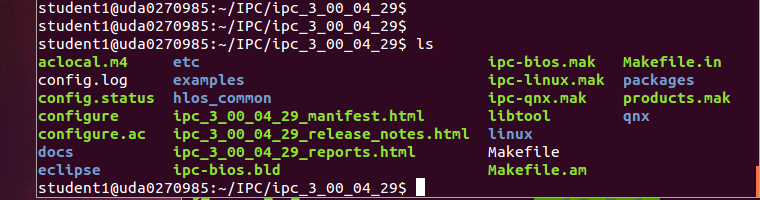
In your home directory /home/studentN create a new directory named IPC and move there

* sudo mkdir IPC
* cd IPC

Copy the ipc directory from the latest release into the new created private directory. Currently the latest release is release 3\_15

* sudo cp -R /opt/ti/MCSDK\_3\_15/ipc\_3\_00\_04\_29 / .
* cd ipc\_3\_00\_04\_29
* ls

The directory should look like the following:



### Step 2 – Build the ARM executable

Detailed instructions how to install and build the Linux version of IPC are in the file IPC\_Install\_Guide\_Linux.pdf that is part of the release in directory \MCSDK\_3\_15\ipc\_3\_00\_04\_29\docs. The install part of the IPC is already in the release, so we will start with the build procedure.

First a set of environment variables should be set in the file products.mak in the directory ipc\_3\_00\_04\_29 (or any later version of the IPC). The following is a list of environment variables that need to be updated:

TOOLCHAIN\_INSTALL\_DIR

TOOLCHAIN\_LONGNAME

PLATFORM

XDC\_INSTALL\_DIR

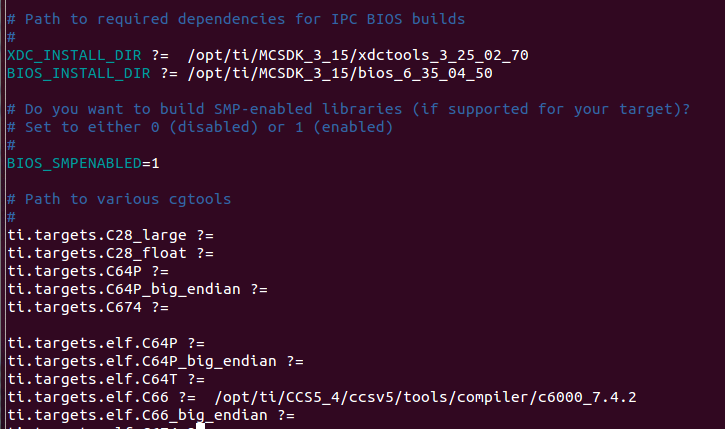
BIOS\_INSTALL\_DIR

ti.targets

The user guide from above explains what needs to be set in the variables that are mentioned above. The following is an example of the definition. You may have to modify the variables based on structure of your system



The platform is tci6638 (Hawing). The TOOLCHAIN\_INSTALL\_DIR is the location where the Linaro tools were installed on the server. The KERNEL\_INSTALL\_DIR is where the kernel sources were installed using the git repository. Instructions how to install the kernel sources using the git repository are given in Lab 2 task 2 of the KeyStone II Multicore Workshop Arm-Based Lab manual.



XDC)INSTALL\_DIR is part of the MCSDK release and the location points to where MCSDK was installed. The same is true for the BIOS\_INSTALL\_DIR. The ti.targets.elf.c66 is the location of the code generating tools for this platform. The tools location is part of the CCS release. The CCS was installed in /opt/ti /CCS5\_4 directory.

In order to manipulate files in the ipc directory you have to change the permission. Since this is a private copy on a local network, you can give full permission

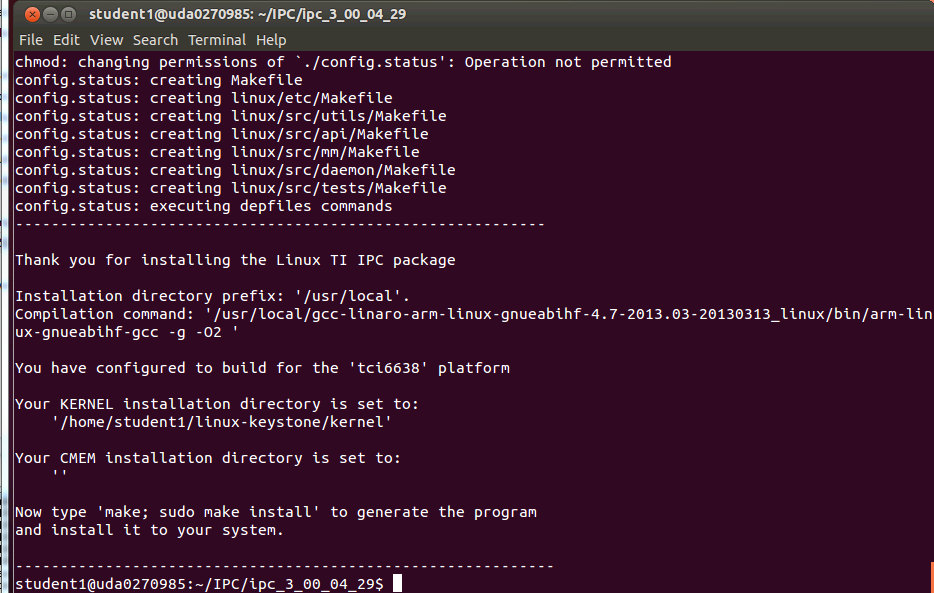
* cd ..
* sudo chmod 777 –R ipc\_3\_00\_04\_29
* cd ipc\_3\_00\_04\_29

The first step is cleaning the older build:

* sudo make clean

Next step is to run the make utility with the linux makefile:

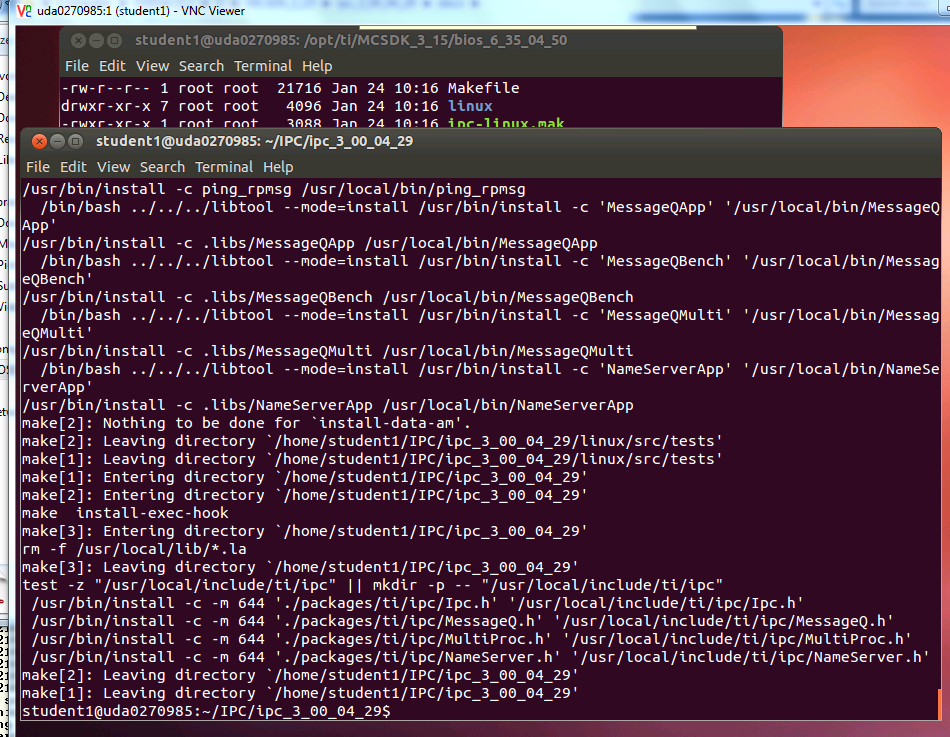
* make –f ipc-linux.mak config



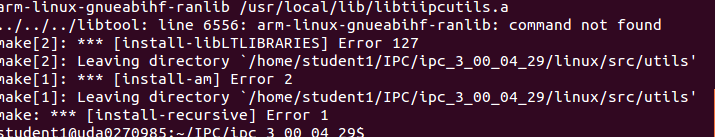
The screen prompts you to run the next make

* make ; sudo make install

Follow the printing in the window. It may ask you for a password for the sudo part. The following screen shot shows the end of the build. In case of error, see the note after the screen shot.



Note – some of the libraries may not be rebuilt, but the output files are built. The screen shot may be like the following:



The build process builds several files. The file MessageQBench in directory linux/src/tests/ is a temporary wrapper script file that shows how the build is done. The file MessageQBench (yes, the same name) in directory linux/src/tests/.libs is the executable. After the build move this file to the file system location

* sudo cp linux/tests/.libs/MessageQBench /opt/filesys/studentN/mcsdk\_x\_xx/usr/bin/.

Verify that the executable in the /opt/filesys/studentN/mcsdk\_x\_XX/lib/usr/bin directory is the one that you built by doing ls –ltr MessageQBench and verify the data and time

### Step 3 – Build the DSP executable

Detailed instructions how to install and build the DSP-BIOS version of IPC are in the file IPC\_Install\_Guide\_BIOS.pdf that is part of the release in directory \MCSDK\_3\_15\ipc\_3\_00\_04\_29\docs. The install part of the IPC is already in the release, so we will start with the build procedure.

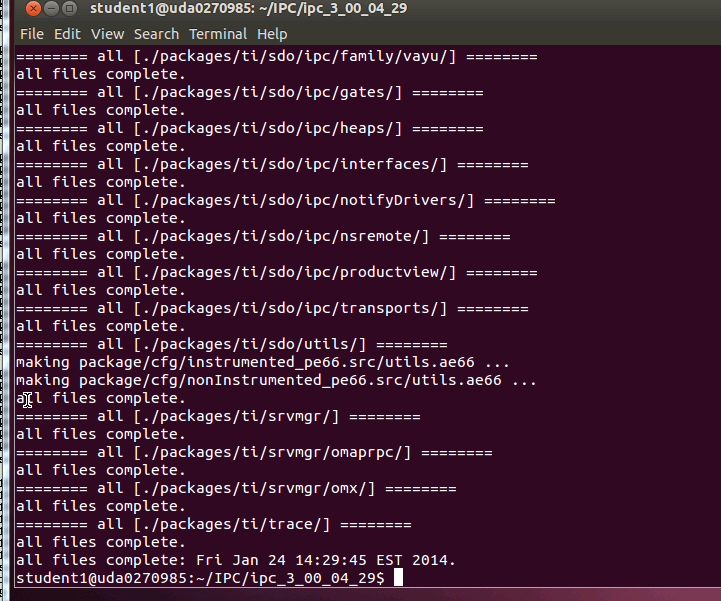
The environment variables in the file products.mak were already set in the previous step.

Return back to the ipc directory

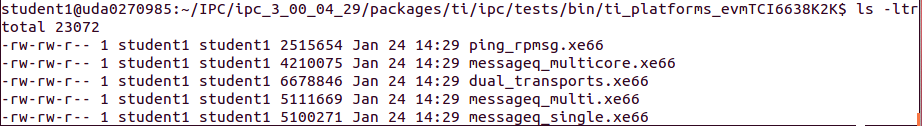
Next step is to run the make utility with the linux makefile:

* cd ~/IPC/ipc\_3\_00\_04\_29 (or a newer version of IPC)
* make –f ipc-bios.mak all

The build will take several minutes. The following is a screen shot when the build is done:



The executables are built in the directory IPC/ipc\_3\_00\_04\_29/packages/ti/ipc/tests/bin/ti\_platforms\_evmTCI6638K2K. There are 5 executables as follows:



The file messageQ\_signal.xe66 should be copied to the file server at location /opt/filesys/studentN/mcsdk\_x\_XX/lib/firmware

* sudo cp packages/ti/ipc/tests/bin/ti\_platforms\_evmTCI6638K2K/messageq\_single.xe66 /opt/filesys/student1/mcsdk\_3\_14/lib/firmware/.

Verify that the executable in the /opt/filesys/studentN/mcsdk\_x\_XX/lib/firmware directory is the one that you built by doing ls –ltr messageQ\_single.xe66 and verify the data and time

## Task 4 – Optional: Modify source code and rebuild the executable

### Purpose

In this task the student will change the ARM code and the DSP code, build and run the executable

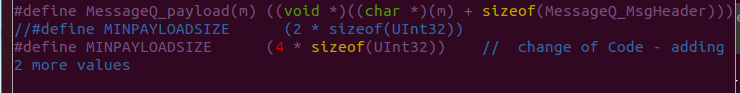
### Step 1 – Modify the ARM code

The demo source file is MessageQBench.c in directory /IPC/ipc\_3\_00\_04\_29/linux/src/tests. Following the source file, it looks like the ARM allocates a message, sends it to the DSP, and then gets the message back and make sure that the data was not corrupted during the transfer. The changes that are suggested for the ARM side are the following:

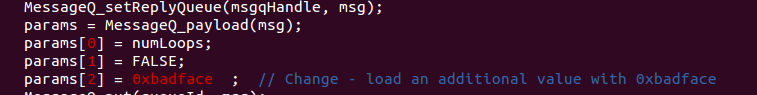
1. Modify a printf statement to show that the code has been modified
2. Adding an additional value (or values) to the message, load the additional value with a known data, and print the additional information from the DSP side

Here are the suggested changes:

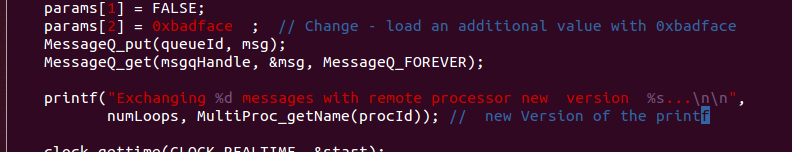
1. Add two more values to the message:



1. Load the additional value with whatever you wish, in the example it is 0xbadface



1. Modify the print statement



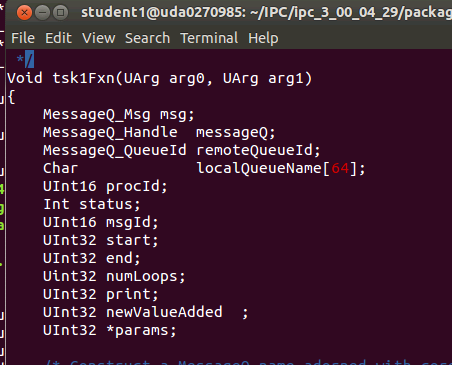
### Step 2 – Build the ARM code

Repeat the steps from the previous task and re-build the ARM executable MessageQBench

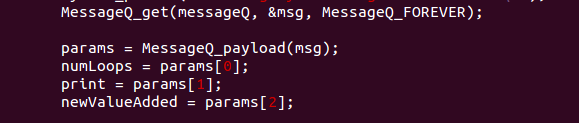
### Step 3 – Modify the DSP code

The DSP source function messageq\_single.c is in directory IPC/ipc\_3\_00\_04\_29/packages/ti/ipc/tests .

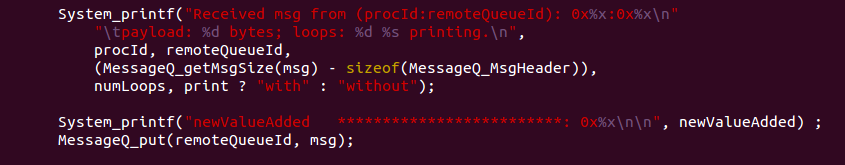
1. Add a new variable newValueAdded to the tsk1Fxn



1. Load the newValueAdded with the last value in the receive message from the ARM



1. Print the value of the newValueAdded



### Step 4 – Build the DSP code

Follow the instructions from the previous task

### Step 5 – Run the file demo\_ipc.sh

Follow the instructions of task 2

1. Observe the printing on the terminal to see the changes that you did in the printf of the ARM code
2. Look at the ARM trace buffer of DSP core 0 to see if the added value is printed. Look at

* Cat /debug/remoteproc/remoteproc0/trace0

The value in newValueAdded should be printed in the trace (in addition to other printings)

# Lab 10: Build U-boot, Boot Monitor and Kernel

Linux distribution enables users to download all source code and build the Linux components. This optional task gives instructions how to do so.

TI uses the Arago distribution on a public server to store all source code. The user can download all the sources and make files to build all images in the release. The Ubuntu server must be connected to the network to facilitate downloading of the source code. If the server is behind a firewall, proxies must be set. These proxies are defined already in the /usr/local/training/studentScript.sh script.

# U-BOOT source code extraction and build instructions

Before getting the source, the user must be in his home directory. Cloning the source code will generate sub-directories.

*cd ~/* back into the home directory /home/studentN

*git* clone git://git.ti.com/keystone-linux/u-boot.git u-boot-keystone . This instruction will clone the repository into your local directory ~/u-boot-keystone

*cd u-boot-keystone*  change directory to where the sources are

*git reset --hard label* Where label is the label that is attached to the current release. To find the head label the user should look at the release notes. For MCSDK 3\_14 the label is ***K2\_UBOOT\_2013-01\_13.09\_01*** so the reset instruction for the current release will be the following

*git reset –hard K2\_UBOOT\_2013-01\_13.09\_01*

Different versions of the U-BOOT can be built, depends how the code is loaded. U-BOOT can be loaded from code composer studio, or from boot methods and the format can either be u-boot.bin, u-boot.img, u-boot-spl.bin or u-boot-spi.gph (The different formats are discussed in the boot loader presentation)

option a: if using CCS to load (u-boot.bin)

*make tci6638\_evm\_config*

*make*

option b: if using the two stage SPI NOR boot

*make tci6638\_evm\_config*

*make spl/u-boot-spl.bin*

*make tci6638\_evm\_config*

*make u-boot.img*

*make tci6638\_evm\_config*

*make u-boot-spi.gph*

## Boot Monitor (skern.bin) source code extraction and build instructions

Before getting the source, the user must be in his home directory. Cloning the source code will generate sub-directories.

*cd ~/* back into the home directory /home/studentN

*git clone git://git.ti.com/keystone-linux/boot-monitor.git*

This instruction will clone the repository into your local directory ~/boot-monitor

*cd boot-monitor* change directory to where the sources are

*git reset --hard label* Where label is the label that is attached to the current release. To find the head label the user should look at the release notes. For MCSDK 3\_14 the label is K2\_BM\_13.08 so the reset instruction for the current release will be the following

*git reset --hard K2\_BM\_13.11*

*make clean*

*make*

## Linux Kernel and device tree source code extraction and build instructions

Before getting the source, the user must be in his home directory. Cloning the source code will generate sub-directories.

*cd ~/* back into the home directory /home/studentN

*git clone git://git.ti.com/keystone-linux/linux.git linux-keystone* This instruction will clone the repository into your local directory ~/boot-monitor

*cd linux-keystone* change directory to where the sources are

*git reset --hard label* Where label is the label that is attached to the current release. To find the head label the user should look at the release notes. For MCSDK 3\_14 the label is *K2\_LINUX\_03.08.04\_13.09\_01* so the reset instruction for the current release will be the following

*git reset --hard K2\_LINUX\_03.10.10\_13.11\_01*

*make keystone2\_defconfig*

*make uImage*

*make keystone-sim.dtb*

*make tci6638-evm.dtb*

The built files:

vmlinux is in /linux-keystone folder:

uImage