BIOS MCSDK 2.0

PCIE Boot Example

Applies to patch release based on 02.00.05.17
Publication Date: January 10, 2012
Version 1.3

Texas Instruments, Incorporated
20450 Century Boulevard
Germantown, MD 20874 USA

Document License

This work is licensed under the Creative Commons Attribution-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/3.0/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Contributors to this document

Copyright (C) 2011 Texas Instruments Incorporated - http://www.ti.com

© Copyright 2011 Texas Instruments, Inc.
All Rights Reserved

Contents

PCIE Boot Example

1 Overview

The PCIE boot example is created to help customer quickly boot DSP through PCIE. The boot example includes:

- A HelloWorld boot example from all cores, which has two CCS projects to build the DDR initialization boot image and HelloWorld boot image.
- A simple POST boot example from core 0 in addition to HelloWorld boot example.
- An EDMA, interrupt boot example shows how to use interrupt between Linux PC and DSP; and fast transfer of a large amount of data between PCIE memory space and DSP memory using EDMA.
- Linux host PCIE loader code to map between PC memory and DSP memory. It loads the boot example into DSP via PCIE link for boot demo purpose.

2 Revision History

Revision	Details
1.3	Add support of big endian boot for 6670/6678; support 32-bit/64-bit Linux; support interrupt between host and DSP; support EDMA over PCIE with throughput measurement
1.2	Add support of 6670
1.1	Add a new PCIE boot demo for “HelloWorld”
1.0	Initial Version

3 References

[1] KeyStone Architecture Peripheral Component Interconnect Express (PCIe) User Guide

(Rev. A, http://www.ti.com/litv/pdf/sprugs6a)

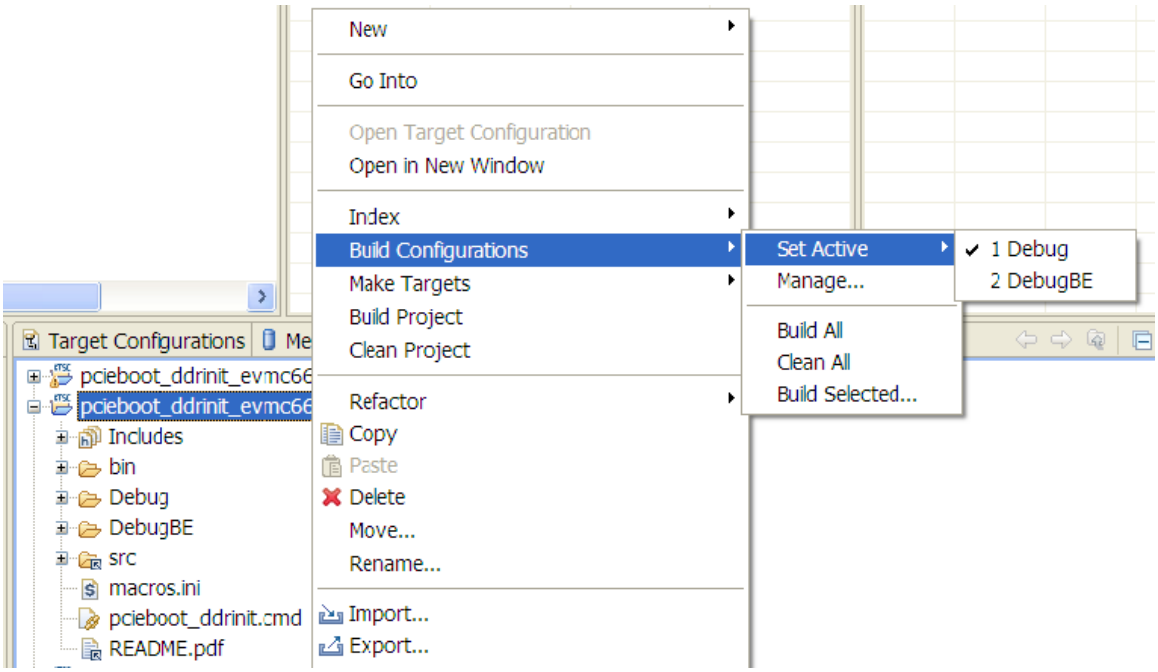
[2] PCIe Use Cases for KeyStone Devices, http://www.ti.com/lit/an/sprabk8/sprabk8.pdf

4 DDR Init Boot Image

The DDR Init project uses the BIOS MCSDK Platform Library to initialize the DDR.

4.1 Procedure to build ddrinit

- Import the project from tools\boot_loader\examples\pcie\pcieboot_ddrinit\evmc66xxl in CCSv5. The project is set to little endian by default. Please double check that "`Debug`" is checked. If one wants to build the project with big endian, please check "`DebugBE`".



- Clean and re-build the project
- The pcieboot_ddrinit_evm66xxl.map and pcieboot_ddrinit_evm66xxl.out will be generated under tools\boot_loader\examples\pcie\pcieboot_ddrinit\evmc66xxl\bin. Note the local L2 memory used by .out file can't be used by user applications, please check the .map file for details, the magic address (0x0087FFFC for TMS320C6678; 0x008FFFFC for TMS320C6670) can't be used as well.
- Edit pcieboot_ddrinit_elf2HBin.bat (Windows user) or pcieboot_ddrinit_elf2HBin.sh (Linux user) under tools\boot_loader\examples\pcie\pcieboot_ddrinit\evmc66xxl\bin for

the correct endianness, by default "ENDIAN=little" is set. Then run the batch file/shell script, it does the following file conversion:

- Uses Code Gen utility hex6x.exe utility to convert the ELF format .out file to a ASCII hex format boot table file
- Uses Bttbl2Hfile.exe to convert the boot table file to a header text file.
- Uses hfile2array.exe to convert the header text file to a header file with array of the image data
- Moves the converted header file to tools\boot_loader\examples\pcie\linux_host_loader\LE or BE folder depending on endianness

5 HelloWorld Boot Image

The HelloWorld project uses the BIOS MCSDK Platform Library to initialize the UART, it will print the "Hello World" and booting information for all the DSP cores to the UART once it runs.

5.1 Procedure to build HelloWorld

- Import the project from tools\boot_loader\examples\pcie\pcieboot_helloworld\evmc66xxl in CCSv5. The project is set to little endian by default. Again please double check that "Debug" is checked. If one wants to build the project with big endian, please check "DebugBE".
- Clean and re-build the project
- The pcieboot_helloworld_evm66xxl.map and pcieboot_helloworld_evm66xxl.out will be generated under tools\boot_loader\examples\pcie\pcieboot_helloworld\evmc66xxl\bin. Note the DDR memory (common code to all cores) used can't be used by user applications; also some local L2 used by individual cores (.stack, .bss, …) can't be used by user applications. Please check the .map file for details.
- Edit helloworld_elf2HBin.bat (Windows user) or helloworld_elf2HBin.sh (Linux user) under tools\boot_loader\examples\pcie\pcieboot_helloworld\evmc66xxl\bin for the correct endianness, by default "ENDIAN=little" is set. Then run the batch file/shell script, it does the same file conversion as pcieboot_ddrinit_elf2HBin.bat does.

6 POST Boot Image

The existing POST is used as another PCIE boot example. The POST uses the BIOS MCSDK Platform Library to do a board test and results can be displayed via UART. Note POST image is built with little endian only in the package.

6.1 Procedure to prepare POST boot image

- Run pcieboot_post_elf2HBin.bat (Windows user) or pcieboot_post_elf2HBin.sh (Linux user) under tools\boot_loader\examples\pcie\pcieboot_post\evmc66xxl\bin, the batch file/shell script first copies post_evm66xxl.out from tools\post\evmc66xxl\bin to tools\boot_loader\examples\pcie\pcieboot_post\evmc66xxl\bin folder, then it does the same file conversion as pcieboot_ddrinit_elf2HBin.bat does.

7 EDMA-Interrupt Boot Image

The Interrupt project uses the BIOS MCSDK Platform Library to initialize the UART, and CSL to support interrupt. It demonstrates how to move data between Linux PC memory and DSP memory using EDMA, via PCIE link, with throughput measured. It also shows how to use interrupt between Linux PC and DSP.

7.1 Procedure to build EDMA-Interrupt

- Import the project from tools\boot_loader\examples\pcie\pcieboot_interrupt\evmc66xxl in CCSv5. The project is set to little endian by default. Again please double check that "`Debug`" is checked. If one wants to build the project with big endian, please check "`DebugBE`".

- Clean and re-build the project

- The pcieboot_interrupt_evm66xxl.map and pcieboot_interrupt_evm66xxl.out will be generated under tools\boot_loader\examples\pcie\pcieboot_interrupt\evmc66xxl\bin. Note the DDR memory from 0x80000000 to 0x80400000 (4MB) is used for EDMA transfer testing. Also the local L2 used by .out can't be used by user applications. Please check the .map file for details.

- Edit interrupt_elf2HBin.bat (Windows user) or interrupt_elf2HBin.sh (Linux user) under tools\boot_loader\examples\pcie\pcieboot_interrupt\evmc66xxl\bin for the correct endianness, by default "`ENDIAN=little`" is set. Then run the batch file/shell script, it does the same file conversion as pcieboot_ddrinit_elf2HBin.bat does.

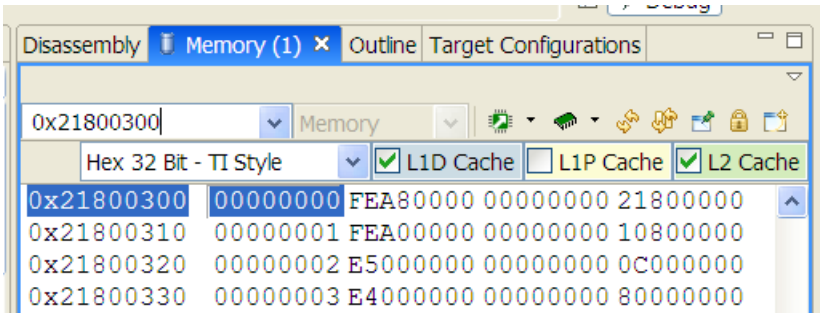
8 PCIE Linux Host Loader Code

The PCIE Linux host loader code has several functions:

- Do a memory mapping between PC memory and DSP memory (4 blocks of memories are requested by DSP via PCIE registers BAR0, BAR1, BAR2 and BAR3 masks:
    - For 6678: 4K, 512K, 4M and 16M respectively for PCIE application registers, local L2, shared L2 and DDR3;
    - For 6670: 4K, 1M, 2M and 16M respectively for PCIE application registers, local L2, shared L2 and DDR3.

The BAR masks are configured inside PCIE initialization code when selects PCIE boot mode on EVM.

- Configure the PCIE inbound address translation through the accessing of application registers as below example for `IB_BARn`, `IB_STARTn_LO`, `IB_STARTn_HI` and `IB_OFFSETn` (n = 0, 1, 2, 3).



- Provide DSP memory read/write API:
  - `Uint32 ReadDSPMemory(Uint32 coreNum, Uint32 DSPMemAddr, Uint32 *buffer, Uint32 length)`
  - `Uint32 WriteDSPMemory(Uint32 coreNum, Uint32 DSPMemAddr, Uint32 *buffer, Uint32 length)`
- Parse the boot example header array file for boot entry address, section size and start address of sections and load the boot data into DSP memory via API.
- Write the boot entry address into the magic address on core 0 via API.
- Provide DSP memory read/write API via EDMA for bulk data transfer:
  - `void HAL_readDMA(uint32_t srcAddr, uint32_t dstAddr, uint32_t size, uint32_t flag)`
  - `void HAL_writeDMA(uint32_t srcAddr, uint32_t dstAddr, uint32_t size, uint32_t flag)`

8.1 Procedure to build and run Linux host loader

- Create a folder (e.g. pcie_test) in a Linux machine. Copy pciedemo.c, Makefile, pcieDdrInit_66xx.h, pcieBootCode_66xx.h, pcieInterrupt_66xx.h and post_66xx.h from tools\boot_loader\examples\pcie\linux_host_loader to the folder.
- Type "`make`", a pciedemo.ko file should be created
- By default, this will build the "HelloWorld" demo on little endian 6678, which is controlled by the following Marcos in pciedemo.c:

```
#define BIG_ENDIAN          0
#define HELLO_WORLD_DEMO    1
#define POST_DEMO           0
```

```
#define EDMA_INTC_DEMO      0
#define EVMC6678L           1
#define EVMC6670L           0
```

One must select the endianness, demo program and target type by toggling between 0 and 1 accordingly. Then, type "`make clean`" and type "`make`" to rebuild the pciedemo.ko. Note, "HelloWorld" and EDMA_INTC demos can be run on both endianness. POST demo can be run on little endian only.

- To insert the module into kernel, type "`sudo insmod pciedemo.ko`"; to view the kernel message, type "`dmesg`"; to remove the module from kernel, type "`sudo rmmod pciedemo.ko`"

8.2 How HelloWorld boot example works

The Linux host first pushes the DDR init boot image data to L2 memory of core 0, then writes the boot entry address of the DDR init boot image to the magic address on core 0, both via PCIE. When the EVM is in PCIE boot mode, the IBL code running on the DSP core 0 polls the entry address and jumps to that address and starts to boot (initialize the DDR). After DDR is properly initialized, the DDR init code clears the magic address and keeps on polling it.

Linux host then pushes the HelloWorld boot image data to DDR memory, then writes the boot entry address of the HelloWorld boot image to the magic address on core 0 to boot core 0. Core 0 starts to boot and print the "Hello World" booting information, and then boot all the other cores by writing the address of _c_int00 to the magic address on other cores and sending an IPC interrupt to other cores. The RBL running on other cores will jump to _c_int00 and start to boot, each core will write 0xBABEFACE to its magic address by running a function write_boot_magic_number().

Note that host boot application needs to wait for some time after pushing the DDR init boot image and before pushing the HelloWorld boot image to the DDR, this will ensure DDR is properly initialized.

8.3 How POST boot example works

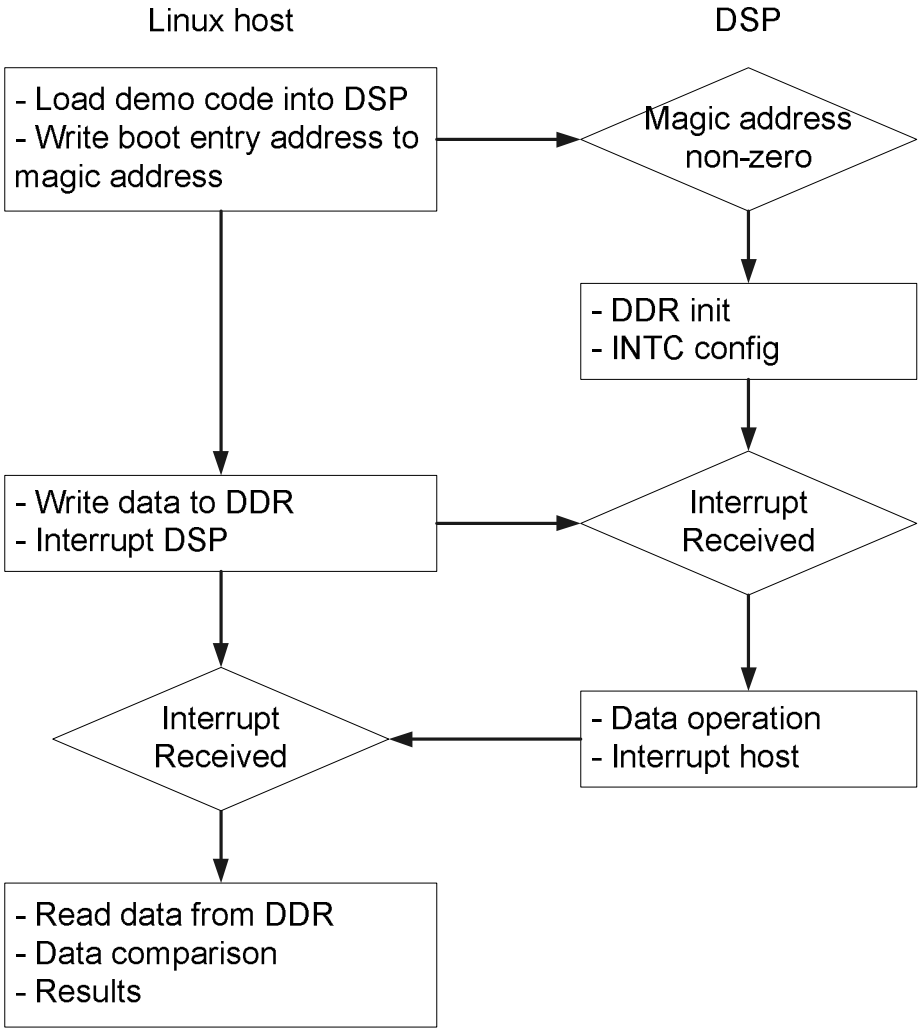
The POST example uses L2 only. The Linux host first pushes the POST boot image data to L2 memory of core 0, then writes the boot entry address of the POST to the magic address on core 0, both via PCIE. The IBL code running on the DSP core 0 polls the entry address and jumps to that address and starts to boot.

8.4 How EDMA-interrupt boot example works

The EDMA-interrupt example uses L2 only. On the host side, host code pushes the boot image data to L2 memory of core 0, then writes the boot entry address to the magic address on core 0, both via PCIE. The IBL code running on the DSP core 0 polls the entry address and jumps to that address and starts to boot. The code initializes the DDR memory, and configures the interrupt using CSL.

Next, host writes 4MB data into DSP's DDR memory via EDMA, and then sends an interrupt to DSP. DSP's ISR does a simple data manipulation upon receiving the interrupt from host and then send an interrupt back to host.

Finally, host's ISR reads back the 4MB data from DDR of DSP and reverses the manipulation for data verification, upon receiving the interrupt from DSP. The EDMA write and read throughput is also measured.



It is worth to mention that the EDMA over PCIE is implemented by poking the EDMA registers from the Linux PC via PCIE link. From DSP perspective, this is an outbound transfer as the local device (DSP) initiates the transactions to write to or read from the external device (PC). The `HAL_readDMA()` function moves data from DSP to PC, this is outbound write from DSP's point of view. Similarly, `HAL_writeDMA()` function is outbound read from DSP's point of view.

Also from PCIE specification, legacy interrupts cannot be generated from RC and be passed downstream. The example code is just making use of the facility that RC can access EP side

register to generate a generic interrupt on local (EP) side using one of the event inputs of Interrupt Controller (INTC). There is no real interrupt signal sent over the PCIe link.

As for the EDMA over PCIE throughput, there are several factors to consider:

- An 8b/10b encoding at the physical layer.
- At the transport layer, there is a 24-byte overhead for 32-bit addressing mode (SEQ, TLP header, ECRC, LCRC …) in addition to the data payload, details see [2].
- The maximum payload size of KeyStone PCIE module is 128 bytes for outbound transfer and 256 bytes for inbound transfer. If using EDMA for the PCIE outbound transfer, the data payload in the TLP is equal to the Data Burst Size (DBS) of the EDMA transfer controller (TC) if the DBS is less than or equal to the maximum PCIE payload size. Here DBS is 128 when CC0 and TC0 of EDMA are used.
- PCIE bandwidth. This can be checked with "`sudo lspci -vvv`" command on Linux host. For example, below shows a 2.5Gx2 = 5.0G/s bandwidth.

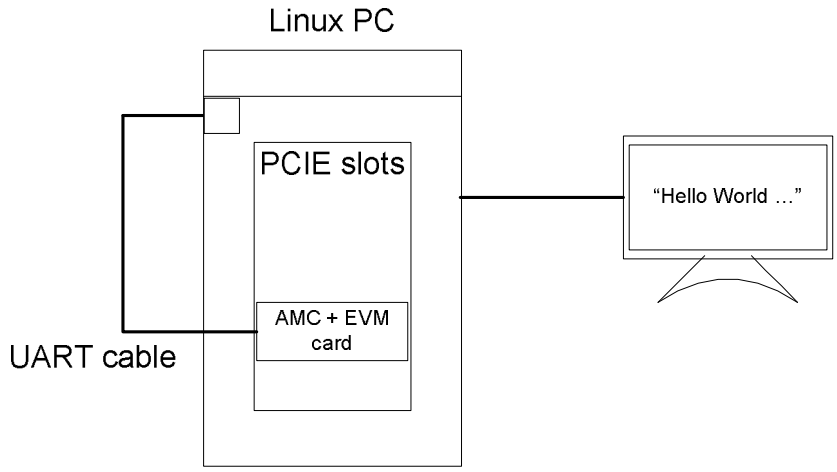
```
      LnkSta:     Speed 2.5GT/s, Width x2, TrErr- Train- SlotClk+
DLActive- BWMgmt- ABWMgmt-
```

Therefore, the theoretical throughout is: 5.0 G/s x 8/10 x 128/(128+24) = 3368 Gb/s = 421 MB/s. The throughout test result is saved in the kernel log, and can be checked by "`dmesg`".

Note: It is IBL (in local L2) that monitors magic address and boots the DDR init (in local L2) or POST (in local L2) or EDMA-interrupt (in local L2) in those demos. If one wants to load his/her own boot demo code, then it shouldn't overlap with the IBL code. As a guideline, the IBL uses memory from 0x00800000 to 0x0081BDFF. To check the exact memory usage, you can re-build the IBL by following the instructions in tools\boot_loader\ibl\doc\build_instructions.txt and check the resulting ibl_c66x_init.map file. In addition, following local L2 is reserved by RBL and shouldn't be used: for 6678 ROM PG 1.0, 0x00872DC0 – 0x0087FFFF; for 6670 ROM PG 1.0, 0x008F2DC0 – 0x008FFFFF.

9 Test Setup and Results

An AMC to PCIE adaptor card, a TMS320C66xxL EVM card and a Linux PC are required to do the test, as the diagram shown below:



The test is verified on both TMS320C6670L and TMS320C6678L cards, with both 32-bit and 64-Linux PCs running Ubuntu 10.04. Other Linux OS are expected to work as well.

- Before connect the system, please update IBL with the latest from MCSDK by following the step 1 -- Programming "IBL" on the EEPROM at bus address 0x51 in tools\boot_loader\ibl\doc\evmc66xx-instructions.txt. Please make sure using "`swap_data = 0`" in tools\writer\eeprom\evmc66xxl\bin\eepromwriter_input.txt. However, only little endian IBL is pre-built in MCSDK. For big endian test, one needs an extra step to build big endian IBL before step 1, please check the note below.

  Note: How to build/program big endian IBL

  - Please follow the instructions under tools\boot_loader\ibl\doc\build_instructions.txt to build the big endian IBL. Basically, you need to: 1) setup build environment via running a script, 2) build the IBL by "`make evm_c667<0/8>_i2c` **`ENDIAN=big`** `I2C_BUS_ADDR=0x51`"

  - When perform step 1 to write big endian IBL to EEPROM, please set the board as **little endian** (SW3, pin 1 "OFF"). This is because the writing tool eepromwriter_evm667xl.out is built with little endian. Also, make sure using "`swap_data = 0`" in tools\writer\eeprom\evmc66xxl\bin\eepromwriter_input.txt.

- Set EVM card to PCIE boot via following switch setting (For SW3, pin 1: OFF: little endian; ON: big endian)

SW3 (pin1, 2, 3, 4)	SW4 (pin1, 2, 3, 4)	SW5 (pin1, 2, 3, 4)	SW6 (pin1, 2, 3, 4)	SW9 (pin1)
(off/on, on, on, off)	(on, on, on, on)	(on, on, on, off)	(off, on, on, on)	(on)

- Assemble the EVM card into the adaptor card

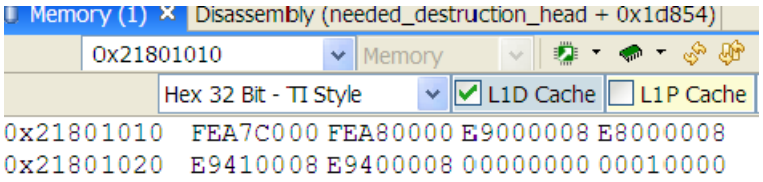
- UART port can be accessed either through Mini-USB connector (USB1) or through 3-pin RS232 Serial port header (COM1). The selection can be made through UART route select connector COM_SEL1 as follows:
    - UART over USB Connector (Default): Shunts installed over COM_SEL1.3-COM_SEL1.1 and COM_SEL1.4-COM_SEL1.2
    - UART over 3-Pin Header LAN1: Shunts installed over COM_SEL1.3-COM_SEL1.5 and COM_SEL1.4-COM_SEL1.6
    - Connect the URAT cable from EVM card to a Linux PC's USB port or serial port based on the desired access method
- Completely shut off the PC power supply (by disconnecting the power cord), insert the AMC adaptor card (with TMS320C66xxL EVM card mounted) into an open PCIE slot in PC's motherboard
- Supply the power to PC, wait for a few seconds and power on the PC.
- Make sure the PCIE device is correctly enumerated by PC by checking below, note DEVICE_ID field is changed from 0x8888 to 0xb005 which is programmed in IBL.
    - Either enter PC's BIOS setting when PC is booting up, a new PCIE device should be populated in the PCIE slot where card is inserted, shown as a "Multimedia device".
    - Or, type "`lspci -n`" under Linux command shell after Linux OS is loaded, a TI device (VENDOR_ID: 0x104c) should be in the list:

```
local-ubuntu:~$ lspci -n
00:00.0 0600: 8086:2774
00:1b.0 0403: 8086:27d8 (rev 01)
….
00:1f.3 0c05: 8086:27da (rev 01)
01:00.0 0480: 104c:b005 (rev 01)
03:00.0 0200: 14e4:1677 (rev 01)
```

      Similarly, one can type "`lspci`",

```
local-ubuntu:~$ lspci
….
00:1f.3 SMBus: Intel Corporation N10/ICH 7 Family SMBus Controller (rev
01)
01:00.0 Multimedia controller: Texas Instruments Device b005 (rev 01)
....
```

    - The PCIE BARn (n = 0, 1, 2, … , 5) registers are written by Linux PC after enumeration, they should be non-zero. Optionally, if a JTAG emulator is available, one can verify this by looking at address starting from 0x21801010 for 6 32-bit word. Below is an example.

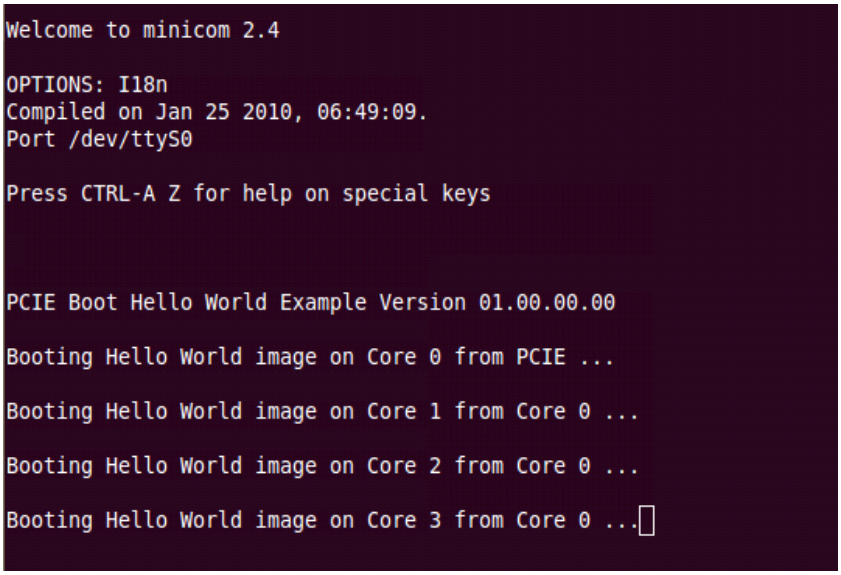


- Prepare pciedemo.ko in the Linux PC, please refer to section 8.1
- On the Linux PC open a new terminal window to run minicom. First run "`sudo minicom -s`" to set the correct configuration: 115200bps, 8-N-1, Hardware flow control: OFF, Software flow control: OFF, and select the correct Serial Device. Save then run "`sudo minicom`" to monitor the port.
- Type "`sudo insmod pciedemo.ko`",
    - For the HelloWorld demo, one should see the following printed on the minicom

      6678 example:

```
Compiled on Jan 25 2010, 06:49:09.
Port /dev/ttyS0

Press CTRL-A Z for help on special keys



PCIE Boot Hello World Example Version 01.00.00.00

Booting Hello World image on Core 0 from PCIE ...

Booting Hello World image on Core 1 from Core 0 ...

Booting Hello World image on Core 2 from Core 0 ...

Booting Hello World image on Core 3 from Core 0 ...

Booting Hello World image on Core 4 from Core 0 ...

Booting Hello World image on Core 5 from Core 0 ...

Booting Hello World image on Core 6 from Core 0 ...

Booting Hello World image on Core 7 from Core 0 ...
```

      6670 example:

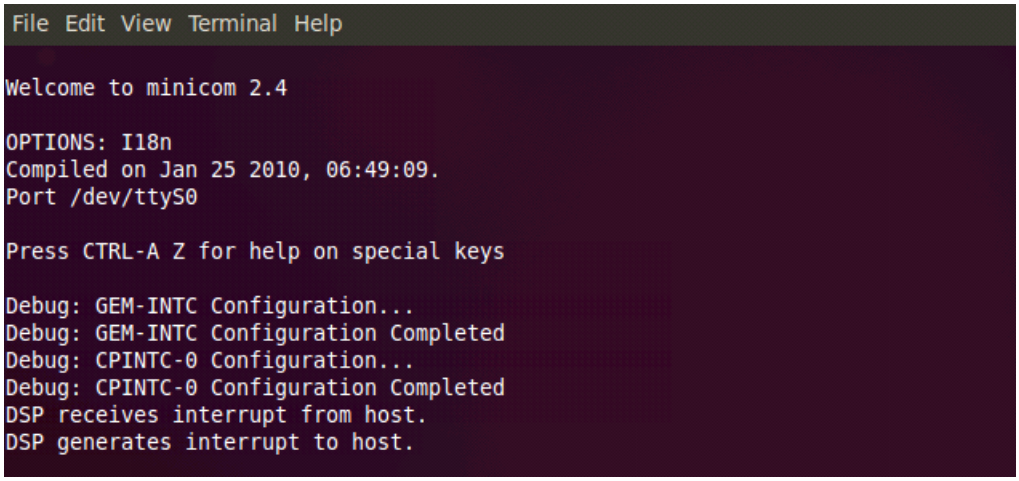


Optionally, if a JTAG emulator is available, one can verify that the PC registers for cores other than core 0 should be inside DDR; and magic address for cores other than core 0 should be written with 0xBABEFACE.

- For the POST demo, one should see the following printed on the minicom for a 6678 example.

```
TMDXEVM6678L POST Version 01.00.00.04
----------------------------------------
SOC Information

FPGA Version: 000B
Board Serial Number: 0DCE4230
EFUSE MAC ID is: 90 D7 EB 0D 12 56
SA is disabled on this board.
PLL Reset Type Status Register: 0x00000001
Platform init return code: 0x00000000
Additional Information:
   (0x02350014) :0BEF0000
   (0x02350624) :000215FF
   (0x02350678) :00831F24
   (0x0235063C) :00081800
   (0x02350640) :00091800
   (0x02350644) :000A1800
   (0x02350648) :000B1800
   (0x0235064C) :000C1800
   (0x02350650) :000D1800
   (0x02350654) :000E1800
   (0x02350658) :000F1800
   (0x0235065C) :00000009
   (0x02350660) :00831FEC
   (0x02350668) :00832000
   (0x02350670) :00832014
   (0x02620008) :0500F007
   (0x0262000c) :0401412E
   (0x02620010) :00000000
   (0x02620014) :651E0020
   (0x02620018) :0009E02F
   (0x02620180) :0602F000
----------------------------------------

Power On Self Test

POST running in progress ...
POST I2C EEPROM read test started!
POST I2C EEPROM read test passed!
POST SPI NOR read test started!
POST SPI NOR read test passed!
POST EMIF16 NAND read test started!
POST EMIF16 NAND read test passed!
POST EMAC loopback  test started!
POST EMAC loopback  test passed!
POST external memory test started!
POST external memory test passed!
POST done successfully!

POST result: PASS
```

- For the EDMA-interrupt demo, one should see the following printed on the minicom



- To view the kernel log, one can type “dmesg”
  - Hello World:

```
[  159.915074] Finding the device....
[  159.915087] Found TI device
[  159.915089] TI device: vendor=0x104c, dev=0xb005, irq=0x0000000b
[  159.915090] Reading the BAR areas....
[  159.915633] Enabling the device....
[  159.915688] pci 0000:04:00.0: PCI INT A -> GSI 16 (level, low) -> IRQ 16
[  159.915693] pci 0000:04:00.0: setting latency timer to 64
[  159.915702] Access PCIE application register ....
[  159.915706] Registering the irq 11 ...
[  159.915718] Boot entry address is 0x1082cc00
[  159.918251] Total 4 sections, 0xd748 bytes of data written to core 0
[  159.976877] Boot entry address is 0x8000cd60
[  159.979045] Total 4 sections, 0xda04 bytes of data written to core 9
```

  - POST:

```
[   96.779446] Finding the device....
[   96.779463] Found TI device
[   96.779464] TI device: vendor=0x104c, dev=0xb005, irq=0x0000000b
[   96.779465] Reading the BAR areas....
[   96.780067] Enabling the device....
[   96.780080] pci 0000:04:00.0: PCI INT A -> GSI 16 (level, low) -> IRQ 16
[   96.780085] pci 0000:04:00.0: setting latency timer to 64
[   96.780094] Access PCIE application register ....
[   96.780098] Registering the irq 11 ...
[   96.780109] Boot entry address is 0x  83a560
[   96.782119] Total 3 sections, 0xb190 bytes of data written to core 0
```

  - EDMA-interrupt:

```
[   86.781006] Finding the device....
[   86.781020] Found TI device
[   86.781021] TI device: vendor=0x104c, dev=0xb005, irq=0x0000000b
[   86.781022] Reading the BAR areas....
[   86.781537] Enabling the device....
[   86.781550] pci 0000:04:00.0: PCI INT A -> GSI 16 (level, low) -> IRQ 16
[   86.781555] pci 0000:04:00.0: setting latency timer to 64
[   86.781565] Access PCIE application register ....
[   86.781568] Registering the irq 11 ...
[   86.781583] Allocating consistent memory ...
```

```
[   86.788306] Boot entry address is 0x  82e300
[   86.791065] Total 5 sections, 0xf358 bytes of data written to core 0
[   88.846147] Write DMA to DSP ...
[   88.858308] Generating interrupt to DSP ...
[   88.982125] Interrupt 11 received from DSP
[   88.982126] Read DMA from DSP ...
[   88.997980] DMA test passed!
[   89.870917] DMA write throughput is: 328.38 MB/s
[   89.870918] DMA read throughput is: 341.64 MB/s
[   89.870919] Freeing consistent memory ...
```