Application Report
SPRAB20–January 2009

Inter-Core Communication on TMS320C6474

Jelena Nikolic-Popovic

ABSTRACT

Inter-core (also called Inter-CPU or Inter-Processor) communication on the C6474 multi-core DSP devices can be accomplished using the on-chip inter-processor communication (IPC) module. The main function of the IPC module is to provide inter-core interrupts. Optionally, flags can be sent along with an interrupt for implementation or more advanced inter-core communication protocols.

The purpose of this application report is to demonstrate the use of the IPC module using a simple application example.

Project collateral and source code discussed in this application report can be downloaded from the following URL: http://www.ti.com/lit/zip/SPRAB20.

Contents

1 Introduction 1
2 IPC Module Structure 2
3 Application Example 2
4 References 6

List of Figures

1 Setting Up BIOS_INSTALL_DIR Environment Variable 3
2 BIOS Configuration File for the IPC Project 4
3 Code Composer Studio Setup for C6474 EVM 5
4 Running the IPC Project in Code Composer Studio 5

1 Introduction

Inter-core communication is the exchange of information among central processing unit (CPU) cores in a multi-core device, such as the C6474 device.

In most applications where the work is partitioned among cores, either in a parallel or in a sequential manner, there is a need for one core to pass control information and/or data buffers to another core. One example would be a WiMax basestation, where one core might be working on transmitter functions, the other core on the basic receiver functions, and the third core on advanced receiver processing such as multiple-input and multiple-output (MIMO). The two receiver cores need to exchange data buffers and possibly control information, whereas, the transmitter and receiver cores are mostly exchanging control information.

The only scenario that does not require such communication is multi-channel processing where each core is assigned a certain number of channels, statically or by an external host processor.

The mechanism of exchanging data buffers typically involves one core writing data to a particular memory area – either via CPU, cache write-backs or direct memory access (DMA), in shared memory such as external double data rate (DDR) or in one of on-chip static random access memory (SRAM) areas, and notifying the other core that the data is available. A sample implementation is given in [2].

C6000 is a trademark of Texas Instruments.
All other trademarks are the property of their respective owners.

For exchanging control information, a notification with a small number of flags is typically sufficient. In both cases, the notification typically occurs via an interrupt sent from one core to another.

In this application report, focus is on the interrupt mechanism among cores using the IPC peripheral that is available on the C6474 device. A brief review of relevant peripheral registers and a description of a simple application that uses the IPC module is also provided.

2 IPC Module Structure

The IPC is a global module, i.e., it is physically located outside the individual CPUs and surrounding mega-modules. It is connected via the CFG bus (see the Configuration Switched Central Resource Block Diagram in [1]).

There are two sets of registers in the IPC module: IPCGRn and IPCARn, n=0,1,2.

IPC Generator Registers (IPCGR) are written to by the source core to generate an interrupt, optionally with flags, to the destination core. There are three registers, one per destination core: IPCGR0/1/2 to generate an interrupt for core 0/1/2, respectively.

Writing to the LSB of IPCGRn generates an interrupt to core n. The event number corresponding to the IPC interrupt is given in the *Interrupts* table in [1]. This information is needed to setup interrupt processing on the destination core, which will be shown in the next section. Note that all cores generate the same interrupt; some of the SRCS flags can be used to determine which core has sent an interrupt. Alternatively, each task on each core can have a flag associated with it so that the receiving core knows which specific task or process generated the interrupt.

There are 28 SRCS bits available for flags; the usage of these bits is completely under software control. For example, a protocol can be defined where each of the remaining two cores might be assigned 14 bits, where each bit or bitfield corresponds to some software-defined flag.

IPC Acknowledge Registers (IPCAR) are used by the destination core to clear SRCS bits. There are three of those registers: IPCAR0/1/2 which are used by cores 0/1/2, respectively.

The source core sets flags by writing 1 to the SRCS bits in the IPCGR and the destination core clears them by writing 1 to the SRCC bits in the IPCAR. The source core cannot clear the SRCS bits.

Detailed register descriptions are provided in the *Inter-DSP Interrupt Registers* section in [1].

3 Application Example

The usage of the IPC module is demonstrated using a simple application. The Code Composer Studio project file, ipc.pjt and associated source files are included with this application report.

3.1 Build Environment

Before using the project, a Code Composer Studio variable CSLforIPC needs to be defined. This variable is used to create the search path for the chip support library (CSL) include files and needs to point to the correct version of the CSL. Note that only a CSL include file is needed and not the actual library file.

The Code Composer Studio variable CSLforIPC should be defined in `c:/CCStudio_v3.3/cc/bin/macro.ini`. If the file does not exist yet, create it as follows:

```
CSLforIPC = C:\CSL\csl_c6488_03_00_06_05\csl_c6474
```

DSP/BIOS version 5.32.04 or higher is needed to support the evm6474 platform file.

DSP/BIOS versions can be downloaded from https://www-a.ti.com/downloads/sds_support/targetcontent/bios/index.html.

Multiple versions can be installed; however, the active version is selected inside Code Composer Studio via Help → About → Component Manager → Target Content → TMS320C6400.

An environment variable BIOS_INSTALL_DIR is used to point to the active version of BIOS. The environment variable can be edited via Start → Control Panel → System → Advanced → Environment variables.

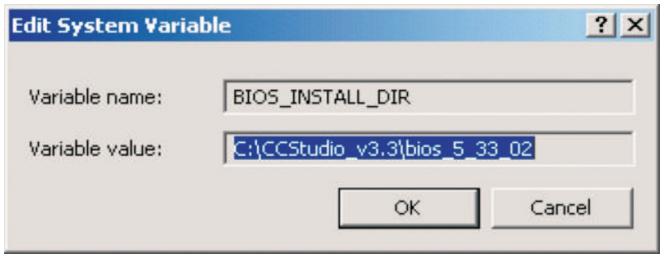

Figure 1. Setting Up BIOS_INSTALL_DIR Environment Variable

The project was tested with C6000™ DSP Code Generation Tool version 6.0.18 and BIOS version 5.33.02.

3.2 How the Application Works

The same application runs on all three cores. Each core is sending messages via IPC registers to the other two cores. The message is sent periodically based on the on-chip timer events. When the core that generates the message (the source core) writes to the IPCGR register that corresponds to the receiving core (the destination core), the destination core receives an interrupt. This corresponding interrupt service routine (ISR), that runs on the destination core, determines which core was the source of the message and posts a corresponding software semaphore; there are three software semaphores in total, one per core. The posting of the software semaphore causes a context switch to a task which is pending on the semaphore. There are three tasks and each processes messages from a particular core. Note that on any given core only two tasks are active since, in this example, a core does not send a message to itself.

the usage of SRCS bits for all three IPCGR registers is defined as follows:

- SRCS0-7 and SRCS24 are used by core 0
- SRCS8-15 and SRC25 are used by core 1
- SRCS16-23 and SRC26 are used by core 2

The high-order bits (SRCS24-26) are used by the destination core to determine from which core the interrupt originated. The low-order bits (SRCS0-23) are used for flags. In this simple application, all 8 bits are used together as an 8-bit counter such that it is easy to verify that the receiving core has received all the interrupts.

The application is using DSP/BIOS and does not require any other libraries. The BIOS configuration for the project is shown in Figure 1. The following BIOS objects are configured:

- HWI_INT4 is the object that corresponds to the inter-core interrupt, with the interrupt selection number 76 (see [1] for the mapping of events to interrupt selection numbers).
- PRDipcsend is a periodic function that is responsible for sending messages. It is triggered once every 5000 ticks or 5sec. Note that tick corresponds to 1msec and is generated by Timer 5.
- TSK0, TSK1, and TSK2 are processing messages from cores 0,1, and 2, respectively. The corresponding software functions run forever and pend on SEM0, SEM1, and SEM2, respectively.
- SEM0, SEM1, and SEM2 objects are software semaphores used to signal receipt of a message from core 0, 1, and 2, respectively. The semaphores are posted in the interrupt service routine for the IPC event. Tasks TSK0, TSK1, and TSK2 pend on the semaphores and are running whenever a new message from the corresponding core is received.
- trace object in the LOG manager is used to display LOG_printf() messages in DSP/BIOS message log window.

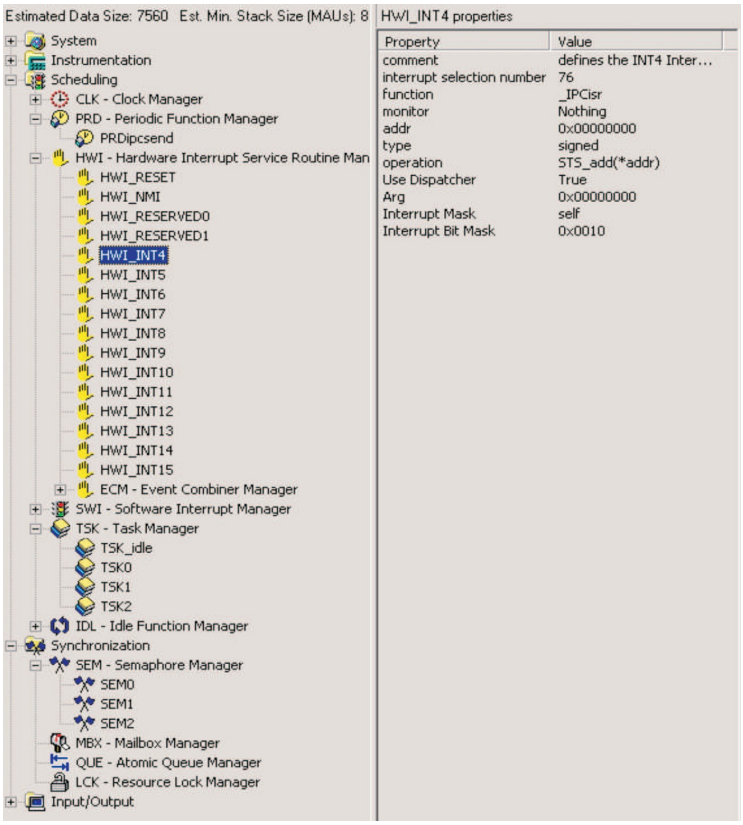

Figure 2. BIOS Configuration File for the IPC Project

3.3 How to Run the Application on the Evaluation Module (EVM)

The steps to run the application on the EVM are shown below.

1. Configure Code Composer Studio setup for the C6474 operation. It should look like Figure 2.
2. Start Code Composer Studio.
3. Open C64PLUS_F1A, _F1B and _F1C from the Parallel Debug Manager (PDM). These sessions correspond to cores 0, 1, and 2 on Device 1.
4. Do the following in each session,
   a. Project → Open … and navigate to ipc_c6474.pjt.
   b. Debug → Connect.
   c. Project → Build (F7).
   d. File → Load … navigate to IPC.out.

   Note: For this step to be performed automatically after the previous step, set Option → Customize → Program/Project/IO → *Load Program after Build*.

   e. Open DSP/BIOS → Message Log.

5. Choose Debug → Run from the PDM. All three cores will start running at the same time. After 5-10 seconds, the Message Log should start displaying messages similar to Figure 3.

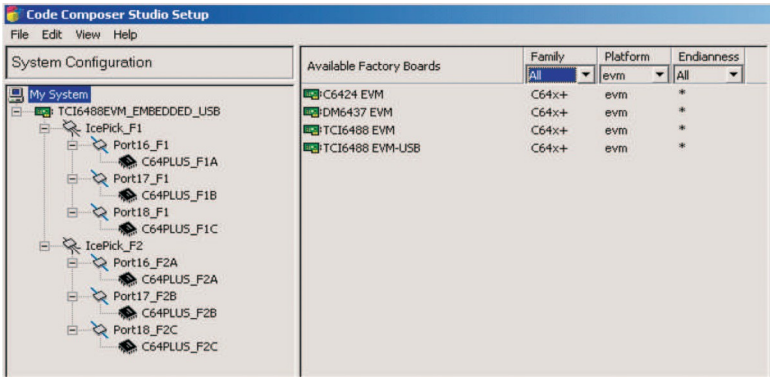

Figure 3. Code Composer Studio Setup for C6474 EVM

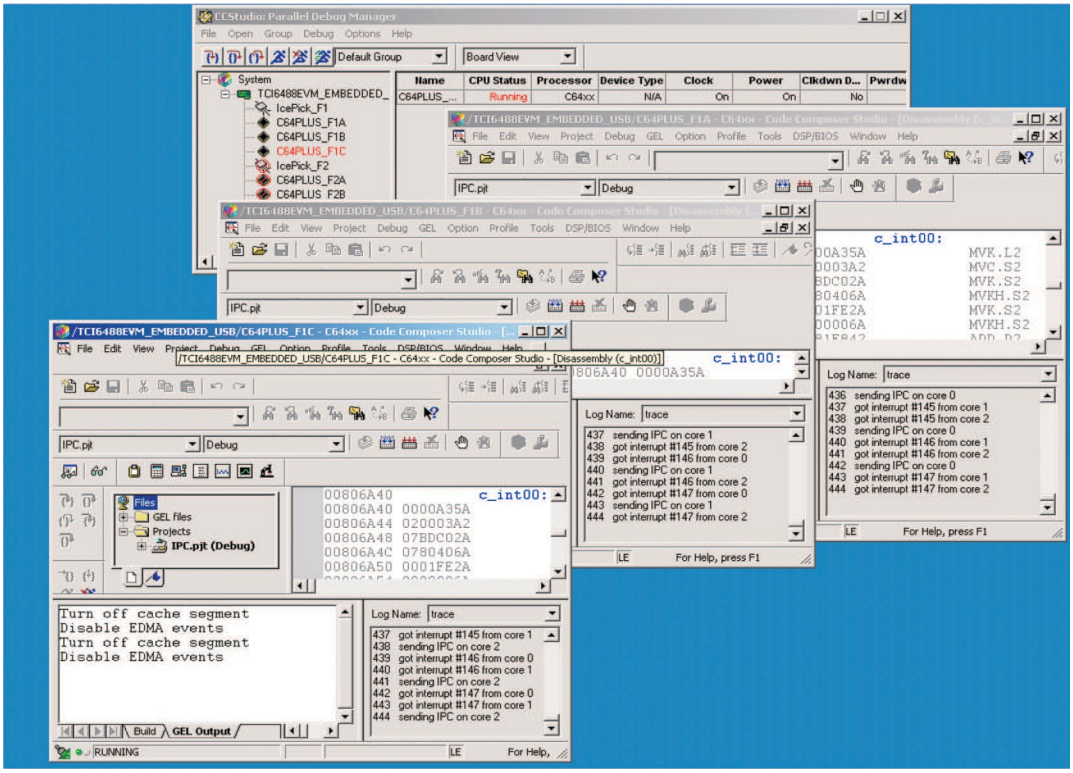

Figure 4. Running the IPC Project in Code Composer Studio

Note that the application will also run in the simulator, but the period of PRDicssend() should be reduced from 5 seconds to 5 msec. Also, the RTDX mode setting under Input/Output in the BIOS configuration file should be changed from *JTAG* to *Simulator*.

The application is built for Little Endian. To change endianess, make the following two changes and rebuild:

1. Change System → Global Settings → General → DSP Endian Mode in the DSP/BIOS configuration file.
2. Change Compiler → Advanced → Endianness in Project → Build Options.

4 References

1. *TMS320C6474 Multicore Digital Signal Processor Data Manual* (SPRS552)
2. Shared Memory Message Queue Transport (SHM_MQT), https://www-a.ti.com/downloads/sds_support/targetcontent/MQT/index.html

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale (www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2019, Texas Instruments Incorporated