

The SCC Platform Overview

Revision 0.75

This document is an introduction to the SCC platform. The SCC is Intel's single-chip computer, the latest research chip from Intel Labs. This chip consists of a mesh of 24 tiles with two IA (P54C) cores per tile, off-chip private memory per core, and a shared on-chip communication buffer. This document is intended to present the capabilities and potential that the SCC platform provides for both academic and industrial research.

Please read the SCC Documentation Disclaimer on the next page.

Intel Labs solicits and appreciates feedback. If you have comments about this documentation, please email them to SCC_Technical_Questions@intel.com.

You can also comment in our forum at <http://communities.intel.com/community/marc> .

Please understand that this is a preliminary document and changes often. Some sections are incomplete, and some don't even exist yet. Please do not hesitate to request more information about topics that you feel are too sparsely documented or topics that are not yet present.

IMPORTANT - READ BEFORE COPYING, DOWNLOADING OR USING

Do not use or download this documentation and any associated materials (collectively, “Documentation”) until you have carefully read the following terms and conditions. By downloading or using the Documentation, you agree to the terms below. If you do not agree, do not download or use the Documentation.

USER SUBMISSIONS: You agree that any material, information or other communication, including all data, images, sounds, text, and other things embodied therein, you transmit or post to an Intel website or provide to Intel under this agreement will be considered non-confidential ("Communications"). Intel will have no confidentiality obligations with respect to the Communications. You agree that Intel and its designees will be free to copy, modify, create derivative works, publicly display, disclose, distribute, license and sublicense through multiple tiers of distribution and licensees, incorporate and otherwise use the Communications, including derivative works thereto, for any and all commercial or non-commercial purposes.

THE DOCUMENTATION IS PROVIDED "AS IS" WITHOUT ANY EXPRESS OR IMPLIED WARRANTY OF ANY KIND INCLUDING WARRANTIES OF MERCHANTABILITY, NONINFRINGEMENT, OR FITNESS FOR A PARTICULAR PURPOSE. Intel does not warrant or assume responsibility for the accuracy or completeness of any information, text, graphics, links or other items contained within the Documentation.

IN NO EVENT SHALL INTEL OR ITS SUPPLIERS BE LIABLE FOR ANY DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, LOST PROFITS, BUSINESS INTERRUPTION, OR LOST INFORMATION) ARISING OUT OF THE USE OF OR INABILITY TO USE THE DOCUMENTATION, EVEN IF INTEL HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME JURISDICTIONS PROHIBIT EXCLUSION OR LIMITATION OF LIABILITY FOR IMPLIED WARRANTIES OR CONSEQUENTIAL OR INCIDENTAL DAMAGES, SO THE ABOVE LIMITATION MAY NOT APPLY TO YOU. YOU MAY ALSO HAVE OTHER LEGAL RIGHTS THAT VARY FROM JURISDICTION TO JURISDICTION.

Copyright © 2010, Intel Corporation. All rights reserved.

*Other names and brands may be claimed as the property of others.

Revision History for Document

0.7	Rewrote to show use of the new MCPC.
0.75	Typos; updated links

Table of Contents

1	Introduction	6
2	The Purpose of this Document	6
3	Overview of the SCC Architecture.....	6
4	Overview of the Tile and Cores.....	8
5	Power Management.....	9
6	SCC Memory.....	10
6.1	Organization.....	10
6.2	Address Translation	11
6.3	Cache Behavior	12
6.4	Programmer's View of Memory	13
7	SCC Configuration Registers	14
8	Programming Model.....	17
9	The Management Console.....	17
9.1	Software that runs on the Management Console	18
9.1.1	sccKit.....	18
9.1.2	Compilers	20
10	Programming the SCC.....	21
10.1	RCCE	21
10.1.1	Build the RCCE Emulation Library	21
10.1.2	Run a Simple RCCE Application with the RCCE Emulator	22
10.2	Shared Memory.....	23

List of Figures

Figure 1	SCC Top-Level Architecture	7
Figure 2	Top-Level Tile Architecture.....	8
Figure 3	Voltage and Frequency Domains on the SCC	9
Figure 4	Telnetting to the BMC Server.....	10
Figure 5	Status Command Output.....	10
Figure 6	SCC Address Translation.....	11
Figure 7	Programmer's View of Memory	13
Figure 8	RCCE Applications	17
Figure 9	The sccGui.....	18
Figure 10	Reading the TILEID SCC Register	19
Figure 11	Using sccBoot.....	19
Figure 12	sccKonsole with Four Tabs.....	20
Figure 13	sccKonsole Showing ls Command.....	20
Figure 14	User-Specified Variables in symbols.in.....	22
Figure 15	The RCCE Application Running Successfully.....	23

List of Tables

Table 1	Address Translation subdestID Bits	12
Table 2	Default LUT Entries Determine MPB Addresses	14
Table 3	Default LUT Entries for CRB Base Addresses	15
Table 4	SCC Configuration Registers.....	16

1 Introduction

Intel's Tera-scale Computing Research Program is a worldwide effort to advance computing technology for the next decade and beyond. The program is investigating how to increase the performance and capabilities of current computers. As part of that program, Intel Labs has created a research chip called the Single-chip Cloud Computer (the SCC) to study many-core CPUs, their architectures, and the techniques used to program them.

Intel Labs believes the SCC is an ideal research platform to help accelerate many-core software research. By the middle of 2010, Intel Labs anticipates having dozens of industry and academic research partners conducting advanced software research on the SCC hardware platform.

This document introduces the SCC architecture and discusses the flexibility and potential of the SCC platform for academic and industrial research. Read through the information here. Discover how the SCC platform can enhance your current research or stimulate new research.

- To hear Justin Ratner, Intel's CTO, introduce the SCC platform, go to <http://connectedsocialmedia.com/intel/page/2/>
- For technical information about the SCC platform, go to <http://www.intel.com/info/scc>.

2 The Purpose of this Document

This document introduces the features and architecture of the SCC Platform and gives an overview of how you would program it. For detailed architectural information, refer to the *SCC External Architecture Specification*. For detailed programming information refer to the *SCC Programmer's Guide*. All the SCC publications assume that you are an experienced parallel programmer.

Finally, this document introduces RCCE (pronounced “rocky”), a many-core communication environment for SCC application programmers. For detailed information about RCCE, refer to the *RCCE Specification*.

The intent is that after reading this document, you will understand what it feels like to use the SCC platform, and you'll be motivated to read the *SCC Programmer's Guide*. You will have seen some screenshots of the SCC software that runs on the SCC Management Console, and you will have run a simple RCCE application on your own workstation, using the RCCE emulator.

3 Overview of the SCC Architecture

The SCC is the second generation processor design that resulted from Intel's Tera-Scale research. The first was Intel's Teraflops Research Chip; it had 80 non-IA cores. The second is the SCC; it has 24 tiles and two cores per tile. The SCC core is a full IA P54C core and hence can support the compilers and OS technology required for full application

programming.

[Figure 1 SCC Top-Level Architecture](#) shows a stylized view of the SCC chip. You can see the 24 tiles arranged in a $X \times Y = 6 \times 4$ array. There is a router associated with each tile. Four memory controllers go to off-die (but on-board) DDR3 memory.

The SCC board communicates with its Management Console (MCPC) over a PCIe bus. The Management Console is a 64-bit PC running some version of Linux. Intel Labs provides software that runs on the Management Console to manage the SCC chip. Key features of this software are the ability to load a Linux image on each core or a subset of cores, to read and modify SCC configuration registers, and to load programs on the SCC cores. You can also write programs that run on the Management Console and configure the SCC platform.

Users don't have to run a Linux image on the cores. Running Linux on the cores is the most common configuration, but certainly not the only one. Some users may be interested in one of the research operating systems being developed for many-core systems.

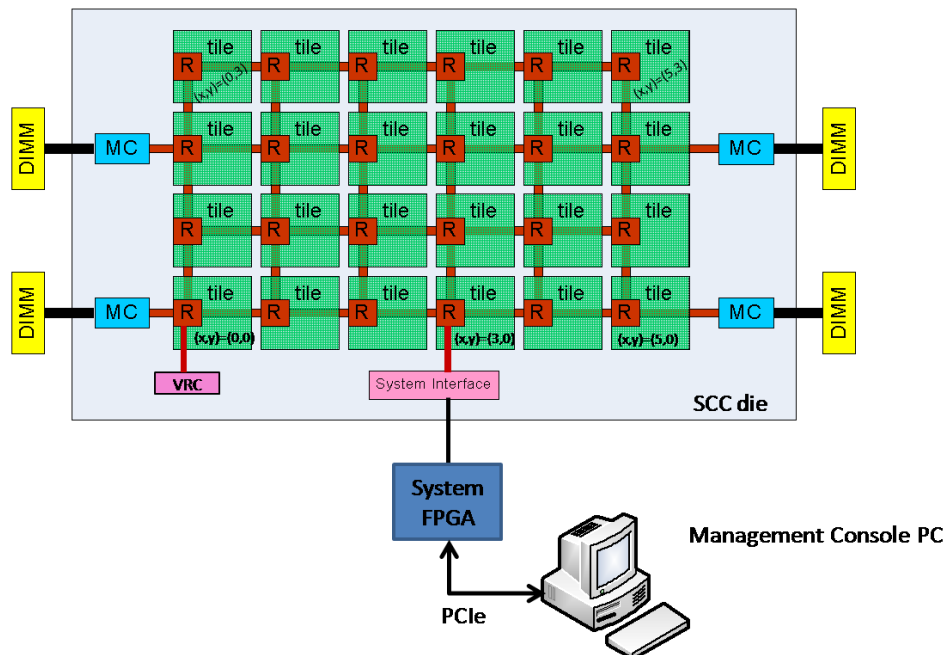


Figure 1 SCC Top-Level Architecture

4 Overview of the Tile and Cores

The SCC has 24 tiles, 48 cores, two cores to a tile. Each core has L1 and L2 caches. The L1 caches are on the core; the L2 caches are on the tile next to the core. Each core has a 16KB L1 instruction cache and a 16KB L1 data cache. Each core's L2 cache is 256KB. [Figure 2 Top-Level Tile Architecture](#) shows an overview of an individual tile.

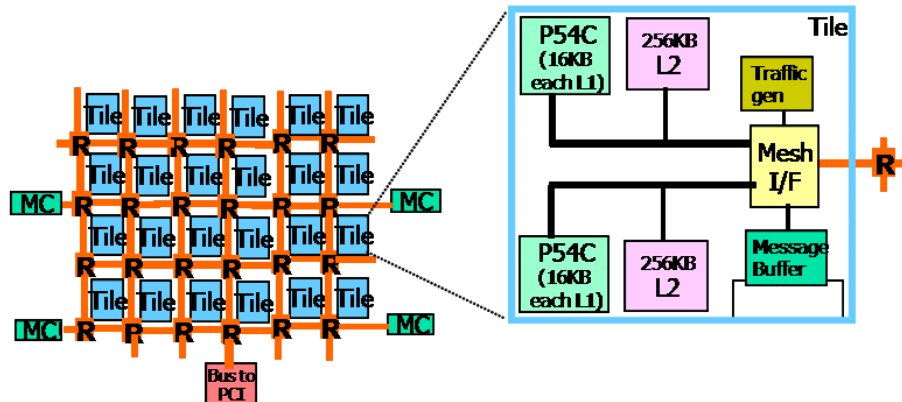


Figure 2 Top-Level Tile Architecture

Each tile also has a message passing buffer (MPB). This message passing buffer is 16KB. Its memory is shared among all the cores on the chip. With 24 tiles, the SCC provides 384KB of message passing buffer. When a message-passing program sends a message from one core to another, internally it is passing the data through the message passing buffers on the chip.

The Traffic Generator is a unit used to test the performance capabilities of the mesh by injecting and checking traffic patterns and is not used in normal operation.

The Mesh Interface Unit (MIU) connects the tile to the mesh. It packetizes data out to the mesh and de-packetizes data in from the mesh. It has buffers that queue data for flit organization. It controls the flow of data on the mesh with a credit-based protocol. It uses a round-robin scheme to arbitrate between the two cores on the tile.

From a programmer's point of view, think of the MIU as catching a cache miss and decoding the core address into a system address. A core uses a 32-bit address called the *core address* to address its 4GB memory space. The SCC platform can have up to 64GB of memory, which is addressable by the *system address*. The MIU uses a lookup table (LUT) to translate the core address into a system address. There is a LUT for each core.

The resulting address request gets placed in one of the following three queues.

- A queue to send the request to the router, then to the memory controller, and finally out to DDR3 memory.
- A queue to send the request to the core's MPB.

- A queue to access local configuration registers, which also reside in the MIU.

5 Power Management

An SCC power controller (called the VRC for voltage regulator controller) provides the capability of changing the voltage and frequency of the cores as well as other parts of the die. This capability supports research on power-aware applications.

The SCC has seven voltage domains and 24 frequency domains. A program running on a core can change the voltage for all the members of a voltage domain; it can also change the frequency for all the members of a frequency domain.

There are six voltage domains comprising four tiles of two cores each in a 2x2 array. The seventh voltage domain is the entire set of tiles. There are 24 frequency domains, one for each tile. [Figure 3](#) illustrates the voltage and frequency domains on the SCC.

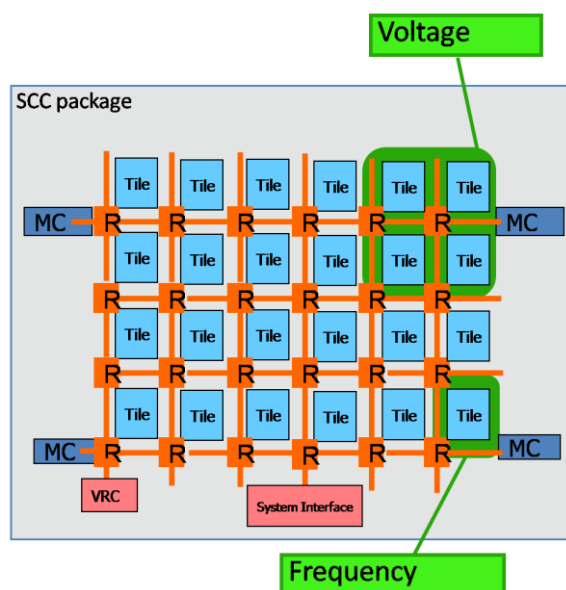


Figure 3 Voltage and Frequency Domains on the SCC

Each core also contains a digital temperature sensor whose output is written to a configuration register. These registers allow you to monitor the cores as you change their voltage and frequency.

You can obtain voltage and power information about the SCC board from the Management Console. From a prompt on the Management Console, type the command

```
telnet <name of your SCC Platform> 5010
```

The name of your SCC platform is assigned to the platform when you receive it. There is no command you can use to get that name. You can also use the IP address of board management controller (BMC) located on the SCC board. Both the name and IP address should be written on a sticker attached to the BMC. If the sccKit is installed, the BMC's IP

address is also available in `/opt/current/systemSettings.ini`. [Figure 4 Telnetting to the BMC Server](#) shows the logon message.

```
=====
Software      1.06 Oct 27 2009 20:30:50
CPLD          1.09
POWR1220     0x80000004 (master), 0x00000004 (slave)
ADC Vref     2.5V
DDR3 modules 0 1 2 3 4 5 6 7
DDR2 module  yes
=====

Welcome to the BMC server of amrrckrb1!
You are participant #1
]>
```

Figure 4 Telnetting to the BMC Server

Then, issue the **status** command. [Figure 5](#) shows the output.

```
mrrlab1001.jf.intel.com - PuTTY

Welcome to the BMC server of amrrckrb1!
You are participant #1
]>status
Power Status: 0x07FFX, ON
Rail 1V0:      1.014 V, 2.666 A
Rail 1V1VCCA:  1.112 V, 2.424 A
Rail 1V1VCCCT: 1.108 V, 3.961 A
Rail 1V5:      1.522 V, 10.693 A
Rail 1V65:     1.720 V
Rail 1V65ADJ:  1.650 V
Rail 1V8:      1.826 V, 2.256 A
Rail 3V3:      3.320 V, 3.197 A
Rail 3V3RCK:   3.312 V, 10.990 A

OPVR VCC0: 1.1004 V (It takes seconds to update all readings!)
OPVR VCC1: 0.9527 V
OPVR VCC2: 1.0998 V
OPVR VCC3: 1.1007 V
OPVR VCC4: 1.1015 V
OPVR VCC5: 1.1006 V
OPVR VCC6: 1.1002 V
OPVR VCC7: 1.1001 V

Temperature:
FPGA:      29 °C (127 °C == no sensor)
RCK:       22 °C  --
FPGA status: 0xD7
Lane Good LED is on
Link width is x4
Link speed negotiated to 2.5 GT/s
DDR 2 phy init is done.
LO: normal operation
CPLD status: 0x43
PLL is locked.
PLL lock lost is cleared.
]>
```

Figure 5 Status Command Output

6 SCC Memory

6.1 Organization

This section provides more detail on the memory organization of the SCC platform.

SCC memory consists of off-chip DDR3 DRAM and on-chip SRAM.

The off-chip DRAM is accessed through four on-die memory controllers. This off-chip DRAM is divided into memory private to each core and memory shared by all cores. Where this division occurs is configurable and determined by values in the lookup table (LUT) associated with each core and by the core's own pagetables. Their default is to give each core as much private memory as possible. The memory that's left over is shared among the cores.

These LUTs get assigned default values at boot time. You can change what those default values are so that the next time you boot, the LUTs will have different values. How you do this is an advanced procedure that is described in the *SCC Programmer's Reference*. The procedure involves booting first with the default values, then modifying files produced in a `tmp` directory, and finally running a configuration tool to produce a `dat` file. Then, the next time you boot the cores, you specify that new `dat` file as the source for the LUT entries.

You can also dynamically change LUT values after boot. You can do this by running a program on the corresponding core or through a program running on the Management Console.

The tiles are organized into four regions, each of which maps to a particular memory controller. When a core accesses its private off-chip memory, it goes through the memory controller assigned to its region. Each memory controller also supports shared memory, memory shared among all 48 cores; and so when accessing its off-chip shared memory, a core may go through any of the four memory controllers.

The on-chip SRAM is the message-passing buffer. Access to this buffer is typically internal to a message passing programming model. For example, RCCE, the small library for many-core communication that is provided with SCC, provides message-passing calls tuned to the needs of many-core chips. Access to the message-passing buffer occurs internally within RCCE.

As a programmer, you can also explicitly access the MPB as well as the shared off-chip DRAM, but you must take into account the way a core's caches interact with the MPB. Refer to [Section 6.3 Cache Behavior](#).

6.2 Address Translation

[Figure 6 SCC Address Translation](#) illustrates how the SCC translates a core address into a system address. Note that a core address starts off at 32 bits. The top 8 bits go to the LUT, which puts out 22 bits. The lower 24 bits pass through.

Of those 22 bits coming from the LUT, the lower 10 bits are prepended to the 24 bits that passed through, resulting in a 34-bit address. This is the address sent to the memory controller. Each memory controller can address up to 16GB, hence the 34 bits.

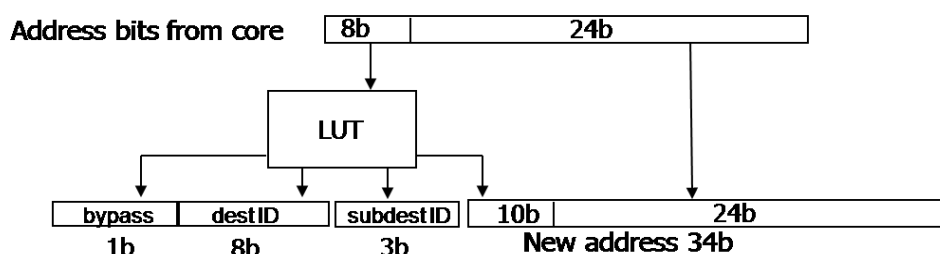


Figure 6 SCC Address Translation

The other bits coming out of the LUT are a 1-bit bypass, an 8-bit `destID`, and a 3-bit `subdestID`. When the bypass bit is set, the address is local to the tile's MPB. The `destID` is ignored.

Note, however, that the bypass bit should no longer be used. For details about this issue, refer to bug 46 in the SCC Bugzilla database, <http://marcbug.scc-dc.com/bugzilla3>.

The 8-bit destID identifies the tile containing the router. The destID is in (y,x) format, 4 bits for y and 4 bits for x. For example, the destID for the top-right memory control in [Figure 1 SCC Top-Level Architecture](#) is (y=2,x=5) or 0010 0101. Note that the actual bit pattern for destID is (y,x) while in text we usually refer to the format as (x,y).

The 3-bit subdestID specifies the port on the router. [Table 1](#) shows the values that subdestID can take. The router ports are identified as N,E,W, or S for north, east, west, and south. In [Figure 1 SCC Top-Level Architecture](#), N is straight up. E is to the right, etc.

The four DDR3 memory controllers are on either the east (E-port) or west (W-port) of their corresponding router. No memory controller is on the north (N-port) of a router.

In addition to a memory controller, the router can address the VRC (which is on the south port of the router at (0,0), the system interface (which is on the south port of the router at (y=0,x=3), the SCC configuration registers, and the MPB.

The two subdestIDs 0x0 and 0x1 identify the destination core on a tile. A core is a destination if a core is writing into the memory space of another core.

Sub-Destination	subdestID	Comment
Core0	0x0	Core0 is the destination
Core1	0x1	Core1 is the destination
CRB	0x2	Configuration Register
MPB	0x3	Message Passing Buffer (exists on every tile)
E_port	0x4	(y=0,x=5) and (y=2,x=5) select DDR3 MC
S_port	0x5	(y=0,x=3) selects the System Interface (SIF) (0,0) selects the VRC
W_port	0x6	(0,0) and (y=2,x=0) select DDR3 MC
N_port	0x7	Nothing is on this port of any edge router

Table 1 Address Translation subdestID Bits

6.3 Cache Behavior

A core's private off-chip DRAM is cached through L1 and L2 according to the normal rules associated with the P54C processor. Because there is no cache coherence among the cores, the SCC system avoids snooping, snarfing, or any other type of cache protocol overhead. The SCC chip does not provide any mechanism to maintain cache coherence among the cores. A programmer must explicitly manage the coherence of cache data among cores.

If you are a RCCE programmer, you need not be concerned with cache issues. The RCCE library takes care of those issues for you.

If you want to manage your own caches, the SCC provides you with two features that will

assist you. The SCC provides a special tag (called MPBT for Message Passing Buffer Type) that identifies the cache lines for data in a shared memory region. The SCC also provides a new instruction called **CL1 INVMB** that marks all data of type MPBT as invalid L1 lines. Accessing data in an invalid L1 cache line forces an update of the data in L1 with the data in the shared memory.

6.4 Programmer's View of Memory

[Figure 7](#) shows a programmer's view of SCC memory. Note the 48 CPUs (CPU_0 ..., CPU_47) and the off-chip private DRAM as well as the off-chip shared DRAM. The default is for the amount of shared off-chip DRAM to be much smaller than the total of all the private DRAM.

The figure also shows the L1 and L2 caches for each of the cores. Data in shared memory may or may not bypass L2 and go directly to L1. Data typed as MPBT bypass L2 and go directly to L1. Legacy software by default is not marked as MPBT data and hence does not bypass L2. Recall that the SCC processor provides a new memory type called the Message Passing Buffer Type (MPBT). A reserved bit in the P54C's pagetable entry marks data as MPBT.

Finally, note the test-and-set register on each core. This register is used to lock the core and perform atomic operations.

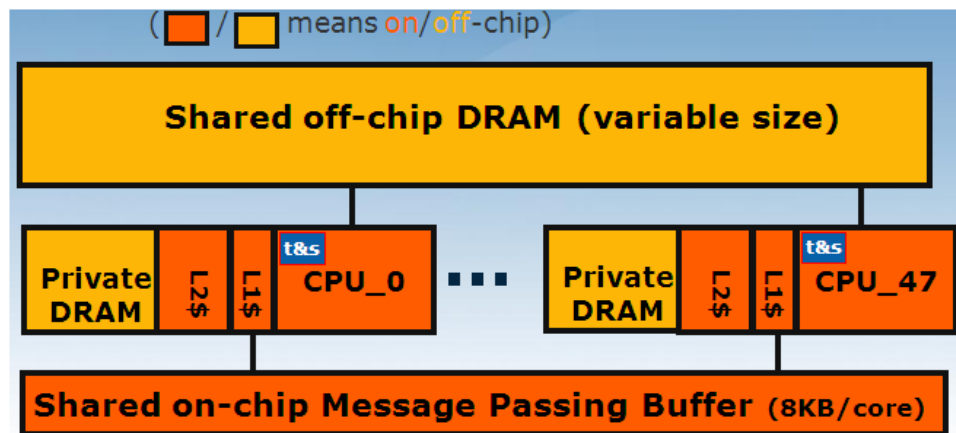


Figure 7 Programmer's View of Memory

Typically programmers will not directly access the message passing buffer, but that capability does exist. The way you would do this to write to the core addresses assigned to the MPB. The LUT assigned to the core the program is running on determines what those addresses are. Default LUT entries are loaded at boot time. As an example, [Table 2 Default LUT Entries Determine MPB Addresses](#) shows the default LUT entries 192 through 216 that determine the base addresses assigned to the MPB.

There are 256 LUT entries. The core has a 4GB address space. Hence, each LUT entry maps to 16MB of address space. The MPB on a tile is 16KB, not 16MB, so obviously only the lower 16KB assigned to that LUT entry are used. By convention, the lower 8KB are assigned to Core 0, and the next 8KB are assigned to Core 1. Although this memory exists locally on each tile, all cores can access the 384KB of MPB space.

LUT Nbr	Core Address Range	Where the LUT Entry Points to
216	D8FFFFFF - D8000000	:
215	D7FFFFFF - D7000000	MPB in Tile (x=5,y=3)
214	D6FFFFFF - D6000000	MPB in Tile (x=4,y=3)
:	:	:
194	C2FFFFFF - C2000000	MPB in Tile (x=2,y=0)
193	C1FFFFFF - C1000000	MPB in Tile (x=1,y=0)
192	C0FFFFFF - C0000000	MPB in Tile (x=0,y=0)

Table 2 Default LUT Entries Determine MPB Addresses

The way you access the message passing buffer is to take the base address of a tile's MPB and add the offset (0x0000 to 0x3fff) for the location you want to access.

The tiles are in a 6x4 = XxY array. X goes from 0 to 5 horizontally, starting in the lower left-hand corner. Y goes from 0 to 3 vertically, also starting in the lower left-hand corner. The base address for (x,y)=(0,0) is 0xc0000000; the base address for (x,y)=(5,3) is 0xd7000000. For example, the MPB base address for the tile at (x,y) = (2,3) is 0xd4000000.

If you are writing a program, you can use the MPB base address symbols, **MPB_xx_yy**. For example, the symbol for the tile at (X,Y) = (2,3) is **MPB_x2_y3**. There is a special symbol called **MPB_OWN** = 0xd8000000. A core on a tile can use this address to refer to its own MPB base address, without having to obtain its own (x,y) coordinates.

Note, however, that the MPB symbols **MPB_xx_yy** and **MPB_OWN** refer to the base addresses as specified in the default LUT tables. If you modify these entries in the tables, you must redefine the MPB symbols.

7 SCC Configuration Registers

By setting values in the SCC configuration registers, you can control the operating mode of the SCC. These registers are initialized with default values.

You can read and write the SCC configuration registers with the sccGui that runs on the Management Console. You can also access these registers from a program that you write that runs on the SCC cores.

The configuration registers reside in the MIU in a control register buffer (CRB). The way you access a core's configuration registers is to take the start address of its tile's CRB and add the offset for the particular register you want to access. [Table 3 Default LUT Entries for CRB Base Addresses](#) shows the default LUT entries 224 through 227 that determine the base addresses assigned to the CRB.

LUT Nbr	Core Address Range	Where the LUT Entry Points to
248	F8FFFFFF - F8000000	:
247	F7FFFFFF - F7000000	System Configuration Register -- Tile (x=5,y=3)
246	F6FFFFFF - F6000000	System Configuration Register -- Tile (x=4,y=3)
:	:	:
226	E2FFFFFF - E2000000	System Configuration Register -- Tile (x=2,y=0)
225	E1FFFFFF - E1000000	System Configuration Register -- Tile (x=1,y=0)
224	E0FFFFFF - E0000000	System Configuration Register -- Tile (x=0,y=0)

Table 3 Default LUT Entries for CRB Base Addresses

The symbol naming convention follows the same rules as those for accessing the MPB. For example, the CRB base address for the tile at (x,y) = (2,3) is 0xf4000000. If you are writing a program, you can use the CRB base address symbols, **CRB_xx_yy**. For example, the symbol for the tile at (x,y) = (2,3) is **CRB_x2_y3**. There is also a special symbol called **CRB_OWN** = 0xf8000000. A core on a tile can use this address to refer to its own base address, without having to obtain its own (x,y) coordinates.

For example, a core can access its own TileID register as **CRB_OWN + 0x0100** or **CRB_OWN + MYTILEID**, where **MYTILEID** is the symbol for the offset for the TileID register..

As with the MPB symbols, the CRB symbols **CRB_xx_yy** and **CRB_OWN** refer to the base addresses as specified in the default LUT tables. If you modify these entries in the tables, you must redefine the MPB symbols.

This section lists the registers and describes their functions. Refer to the *SCC Programmer's Guide* for a description of the fields in these registers and for examples of how you would write code to access them. The purpose of showing them here is just to give you an overview of the information and capabilities they provide.

[Table 4 SCC Configuration Registers](#) lists the individual configuration registers and their offsets.

Each tile has LUT tables, two Test-and-Set registers, a TileID register, a global clock configuration register, two sensor registers, and two L2 cache configuration registers.

There are two lookup tables (LUTs) on a tile, one for each core. There are no symbols for the LUTs. Each LUT has 256 8-byte entries.

- **LUT0** (for core 0) starts at 0x800 (for entry 0) and goes to 0xff8 (for entry 255). Entry 1 is 0x808; entry 2 is 0x810, etc.
- **LUT1** (for core 1) starts at 0x1000 (for entry 0) and goes to 0x17f8 (for entry 255). Entry 1 is 0x1000; entry 2 is 0x1008, etc.

Each tile has two Test-and-Set registers, one for each core. They have the symbols **LOCK0** and **LOCK1**.

Each tile has a tile ID register with the symbol **MYTILEID**. This register contains the (x,y) coordinates of the tile and a core ID. It contains the core ID of the core that read it.

Each tile has a global clock configuration register with symbol **GCBCFG**. This register controls the router and tile clocks. It also allows you to reset an individual core or its L2 cache.

Each tile has two core configuration registers. They have the symbols **CLCFG0** and **GLCFG1**. Each has 26 valid bits. The lower 12 bits (bits 11:00) are read/write. The upper 14 bits (bits 25:12) are read-only and are status bits. The read-only portion of **GLCFG** is also called **GLSTAT**, but **GLSTAT** is not a programming symbol. The prefix **GL** is a remnant from an internal codename and is no longer relevant.

Each tile has two sensor registers. One is a sensor control register. With this register, you can enable and preset a counter. When you enable and preset the sensor control register on a tile,

you are affecting both cores. The other sensor register is a sensor read register. You can read the counter values on each core. It is a 26-bit register; the upper 13 bits are the value of core 0's counter, and the lower 13 bits are the value of core 1's counter.

Each tile has two L2 cache configuration registers, one for each core. They have the symbols **L2CFG0** and **L2CFG1**. An important bit in this register is the one that enables or disables the L2 cache.

Register Name	Desired Operation	Register Offset	Valid data bits
LUT register core 1 (256 8-byte entries) no symbol	Read/write LUT1	10xxxxxxx 0x1000	22
LUT register core 0 (256 8-byte entries) no symbol	Read/write LUT0	01xxxxxxx 0x0800	22
Atomic Flag Core1 LOCK1	Read/write test-and-set Core 1 atomic	001000000000 0x0400	1
Atomic Flag Core0 LOCK0	Read/write test-and-set Core 0 atomic	000100000000 0x200	1
Tile ID register MYTILEID	Read Tile ID	000010000000 0x0100	11
Global Clock Unit (GCU) GCBCFG	Read/write GCU	000001000000 0x0080	26
Sensor Register	Read Thermal Sensor value	0000001001000 0x0048	26
Sensor Register SENSOR	Read/write Thermal Sensor control	0000001000000 0x0040	14
L2 Cache Configuration 0 L2CFG0	Read/write L2 Cache 0	0000000100000 0x0020	14
L2 Cache Configuration 1 L2CFG1	Read/write L2 Cache 1	0000000101000 0x0028	14
Core Configuration 0 GLCFG0	Read/write Core 0 config	0000000010000 0x0010	26 (top 14 read only)
Core Configuration 1 GLCFG1	Read/write Core 1 config	0000000011000 0x0018	26 (top 14 read only)

Table 4 SCC Configuration Registers

8 Programming Model

The principle usage model for the SCC platform is to load an operating system onto the cores and then load your application. The operating system loaded onto the cores is a version of Linux modified specifically for SCC. I/O system calls are routed to the FPGA.

You can also choose to build your application with the RCCE library. With RCCE, you can access the many-core features of the SCC through the RCCE API.

RCCE provides the “greatest common denominator” for SCC applications. You can start your development with the RCCE software emulator that runs on any standard Windows or Linux system. This emulator uses OpenMP to simulate RCCE calls. Then, you can take your RCCE application and run it on an SCC platform whose cores are running the SCC version of Linux. [Figure 8 RCCE Applications](#) illustrates these two ways of running RCCE applications.

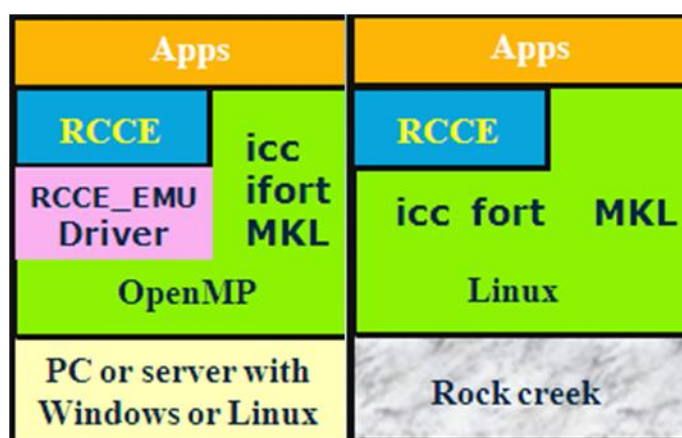


Figure 8 RCCE Applications

9 The Management Console

The Management Console is a 64-bit PC that usually sits in the lab right beside the SCC platform. Users typically VNC into the Management Console from their own workstation.

The Management Console is a generic PC with a PCIe interface running Ubuntu 10.04. It is augmented with some Intel-provided software tools. The Management Console communicates with the SCC platform over a PCIe bus. The PCIe bus connects to a System FPGA interface on the SCC board which connects to a System Interface on the SCC itself. [Figure 1 SCC Top-Level Architecture](#) illustrates the Management Console connecting to the SCC.

The directory `/shared` on the Management Console is NFS-mounted on the cores. You can place your SCC executable in that `/shared` directory and then use `rcceerun` to load and run it on the cores. The utility `rcceerun` was written to load a RCCE application on the cores, but you can also use it to load non-RCCE applications. You could also choose to load your

application directly with **pssh** (<http://freshmeat.net/projects/pssh/>). **rcce**run is a bash shell script that calls **pssh**.

9.1 Software that runs on the Management Console

The Intel-provided software tools that run on the Management Console consist of sccKit, RCCE, a Linux image to load onto the cores, and some tools that will aid you in building your own Linux image for the SCC cores.

You can download the Intel-provided software tools from <http://marcbug.scc-dc.com/svn/repository> . These tools are available for anonymous **svn** checkout.

9.1.1 sccKit

You can put the sccKit software anywhere on the Management Console you like, but typically, it is in the directory `/opt/sccKit/current` where **current** is a link to the latest version stored as a directory in `/opt/sccKit`.

With the sccKit, you can boot Linux on one or more cores, read and write core memory locations and registers, open as ssh connection to one of more cores, and monitor core performance. sccKit has both a GUI interface and a command-line interface. This section gives an overview of using sccKit. It is not intended to completely document sccKit.

[Figure 9 The sccGui](#) shows how the sccGui looks when it starts up. To invoke the sccGui, issue the command **sccGui &**.

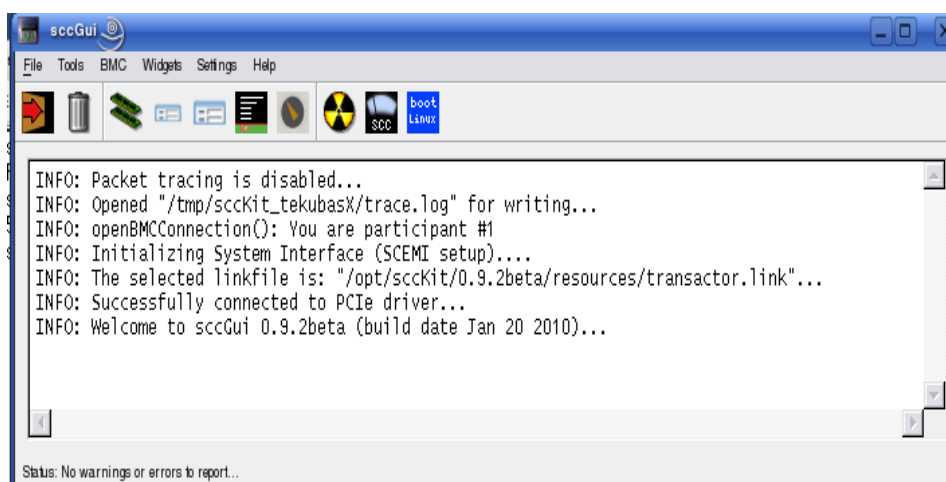


Figure 9 The sccGui

One of the most common uses of the sccGUI will be to boot Linux on the cores. To do that, click on the **Settings** menu item, select **Choose Linux image**, and browse to the location of the Linux image. Then, click on the menu item **Boot Linux**.

Another typical use is to read and write core configuration registers. To do that, click on the **Widget** menu item and select **Flit Widget**. [Figure 10 Reading the TILEID SCC Register](#) shows the screen that results. You have dropdowns that select the tile, the base address (in this example, this is CRB for the control register buffer), and the specific configuration register.

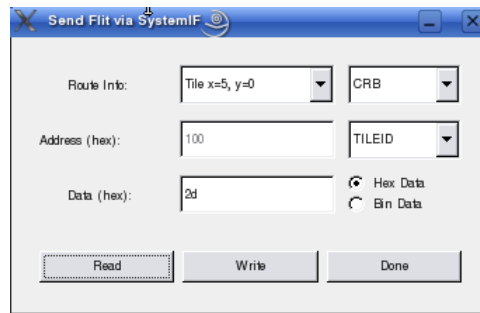


Figure 10 Reading the TILEID SCC Register

[Figure 10](#) shows the value of TILEID for (x,y)=(5,0). The value is 0x2d = 00101101. The value of x is contained in bits 6:3 of TILEID; the value of y is in bits 10:7.

You can also access some of the sccKit features from the command line. You can boot Linux on one or more cores with **sccBoot**. [Figure 11](#) shows using **sccBoot** to probe the status of the cores.

```
YourUsername@YourComputer:/opt/sccKit>|sccBoot -s
tekubasX@mrllab1002:/opt/sccKit> sccBoot -s
INFO: Welcome to sccBoot 1.0.1 (build date Mar 19 2010)...
Status: The following cores can be reached with ping (booted): All cores!
YourUsername@YourComputer:/opt/sccKit>
```

Figure 11 Using sccBoot

Another command-line sccKit feature is **sccKonsole**. With this command, you can open up a console (an **ssh** connection) to one or more cores. With the command-line version, you can specify the number of consoles you want to create. They appear as tabbed shells. When you use the GUI to create consoles, you get 48 tabbed shells.

You can create consoles either with the SccCGui or from the command line. To create consoles for cores 0 through 3, enter the command **sccKonsole 0..3**.

A useful feature of the SCC consoles is the ability to broadcast a command to all opened consoles. Exactly how you do this depends on the particular Linux distribution running on the MCPC. Intel Labs runs Ubuntu 10.04. To have the input of one console sent to all the opened consoles, click on the **Edit** menu item on the window with the tabbed consoles. Then, choose **Copy Input → All Tabs in Current Window**.

[Figure 12](#) shows four tabbed consoles with the second **rck01** selected. [Figure 13](#) shows the execution of an **ls -a** command in **rck01**. The red exclamation point on the **rck01** indicates that its input is broadcast to all the consoles. [Figure 13](#) shows **rck02** selected and showing the result of the invocation from **rck01**.

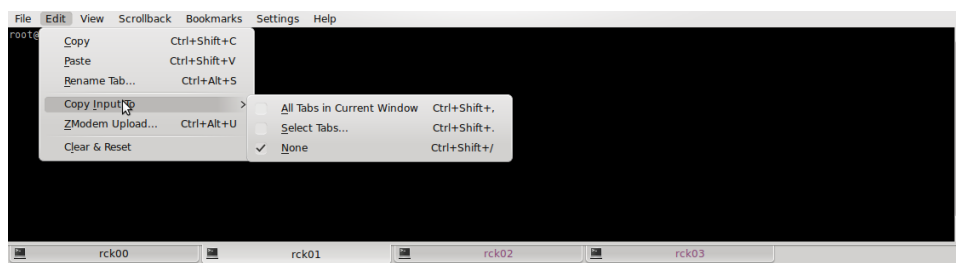


Figure 12 sccKonsole with Four Tabs

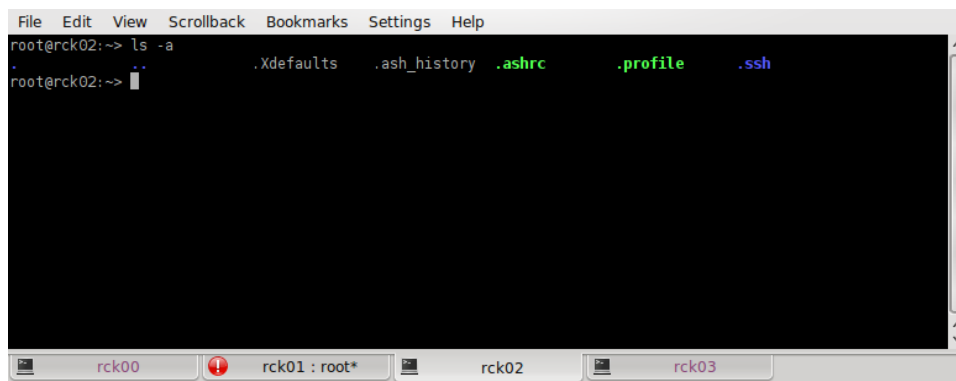


Figure 13 sccKonsole Showing ls Command

The sccKit also includes sccAPI. This is a library that lets you write your own code running on the Management Console that accesses the SCC platform. Intel Labs provides the source code for sccKit so that you can use it as a model when writing your own code. Typical sccAPI calls allow you to perform programmed I/O and DMA access to SCC memory. You can also connect to and send commands to the Board Management Controller. The BMC is an ARM processor that connects to the Management Console via a LAN port.

9.1.2 Compilers

Because the cores are P54C processors, you must use older versions of **icc** (Intel C) and **ifort** (Intel Fortran). You can either build your SCC executable on your own workstation or directly on the Management Console. Once the executable is built, copy it to the **/shared** directory on the Management Console. You need the following compiler versions.

- For **icc**, you need **icc-8.1.038**.
- For **ifort**, you need **ifort-8.1.034**.

If you want to use MKL, Intel's Math Kernel Library, you also need an older version. MKL is not absolutely required, but it significantly improves the performance of RCCE math applications. For example, it improves the performance of the High Performance Computing Linpack Benchmark (HPL) that comes with RCCE.

- For MKL, you need **mkl-8.1.1.004**.

10 Programming the SCC

10.1 RCCE

RCCE (pronounced “rocky”) is a programming model provided with SCC. RCCE is a small library for message passing tuned to the needs of many-core chips such as SCC. RCCE provides

- A basic interface, a higher level interface for the typical application.
- A gory interface, a low level interface for expert programmers.
- A power management API to support SCC research on power-aware applications.

RCCE runs on the SCC chip, as well as on top of a functional emulator that runs on a Linux or Windows platform that supports OpenMP. This document shows how to build the RCCE libraries for emulation and to run a simple example under the emulator.

Refer to the *SCC Programmer’s Guide* to see an example of building the RCCE libraries for use on the SCC Platform itself. The *SCC Programmer’s Guide* also shows several examples of RCCE programs that run on the SCC cores.

RCCE is a versatile library that serves as a research vehicle to understand how message passing APIs map onto many core chips. It is for experienced parallel programmers willing to work close to the hardware.

RCCE, however, is more than a set of message-passing libraries. The RCCE API does use shared memory to send and receive those messages. With the gory interface, you can look at that implementation.

With some additional effort, you can do shared-memory applications by using the shared memory support inside RCCE.

10.1.1 Build the RCCE Emulation Library

You can run the RCCE emulation library on any standard computer running Linux. This section shows how you can build the emulation library.

RCCE is distributed as source code. You may have obtained RCCE source code from an Intel-provided SVN repository or downloaded a tar file. A typical name for this tar file is **RCCEv1.04.tar.bz2**. Note the version number. Your copy may have a different version number.

Put the tar file in some work directory and untar it. This example uses a work directory called SCC. Untarring creates a directory called **RCCEv1.04** with the following contents. Look at the **README** file for more information.

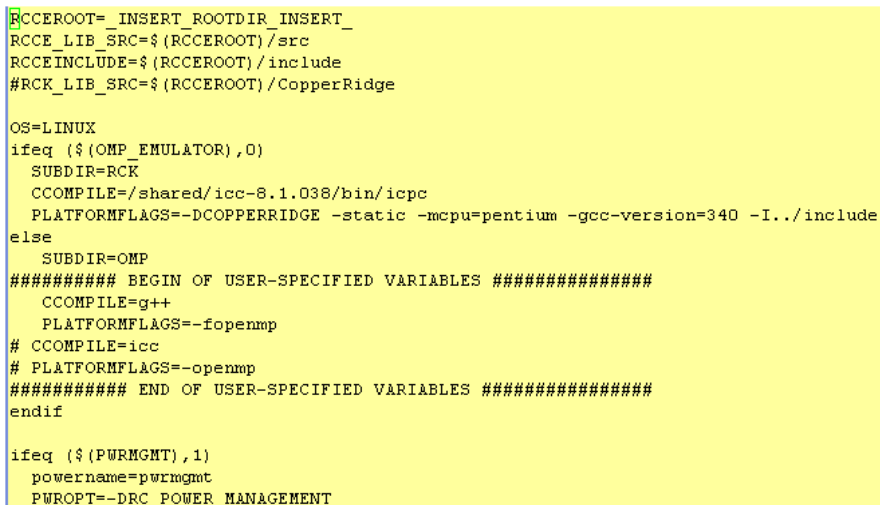
```
tekubasx@tekubasx-desktop:~/SCC/RCCEv1.04$ ls
apps  Change.log  configure  include  Makefile  README
bin   common      hosts      makeall  rcцерun   src
```

Before building the RCCE library or your application, you must decide whether to use the **icc** or **gcc** compiler. The default is to build with **icc**, Intel’s commercial compiler. This

example chooses **gcc**, the opensource Gnu compiler.

To change these defaults, edit the file **common/symbols.in**. [Figure 14 User-Specified Variables in symbols.in](#) shows the appropriate portion of the file **symbols.in**. It is the area delimited by **BEGIN/END OF USER-SPECIFIED VARIABLES**. The example below specifies the use of **g++**, not **icc**. The **g++** running on the Linux used in this example is **g++ 4.4.1**.

In the directory **RCCEv1.04**, type **./configure emulator**. This command is silent. Then, type **./makeall**. This command builds all the emulator versions of the RCCE library.



```

RCCEROOT= INSERT_ROOTDIR INSERT_
RCCE_LIB_SRC=$(RCCEROOT)/src
RCCEINCLUDE=$(RCCEROOT)/include
#RCK_LIB_SRC=$(RCCEROOT)/CopperRidge

OS=LINUX
ifeq ($(OMP_EMULATOR),0)
    SUBDIR=RCK
    CCOMPILE=/shared/icc-8.1.038/bin/icpc
    PLATFORMFLAGS=-DCOPPERRIDGE -static -mcpu=pentium -gcc-version=340 -I../include
else
    SUBDIR=OMP
    ##### BEGIN OF USER-SPECIFIED VARIABLES #####
    CCOMPILE=g++
    PLATFORMFLAGS=-fopenmp
    # CCOMPILE=icc
    # PLATFORMFLAGS=-openmp
    ##### END OF USER-SPECIFIED VARIABLES #####
endif

ifeq ($(PWRMGMT),1)
    powername=pwrmgmt
    PWROPT=-DRC_POWER_MANAGEMENT

```

Figure 14 User-Specified Variables in symbols.in

10.1.2 Run a Simple RCCE Application with the RCCE Emulator

This section shows how, starting with RCCE source code, you can run a simple example. The example is provided as part of the RCCE source code. The simple example is called *pingpong*. It runs on two cores. It will not load on less than two cores or more than two cores.

The program sends a message back and forth between two cores **MAXROUND** times. One core is **ME**; the other core is **!ME** or **YOU**. **ME** sends **MAXROUND** messages and receives **MAXROUND** messages. **YOU** does the same. The program performs this through matching pairs of sends and receives as follows.

```

for (round=0; round <MAXROUND; round++) {
    if (ME) {
        RCCE_send(buffer, BUFSIZE, YOU);
        RCCE_recv(buffer, BUFSIZE, YOU);
    }
    else {
        RCCE_recv(buffer, BUFSIZE, YOU);
        RCCE_send(buffer, BUFSIZE, YOU);
    }
}

```

Now build the *pingpong* application. Go into the directory `apps/PINGPONG`. Because the emulator runs slower than SCC hardware, edit the file `RCCE_pingpong.c` and change `MAXROUND` from 10000000 to a smaller number (say 100). Then, build the application by typing `./make pingpong`.

To run an application, use the `rcцерun` command. Put `rcцерun` in your path. Enter the command

```
rcцерun -emulator -nue 2 -f ../../hosts/rc.hosts pingpong
```

Because you configured with the emulator, the `-emulator` switch is the default and you need not specify it. The `rcцерun` command specifies two cores or units of execution with the switch `-nue 2`. It also specifies a hosts file. This `rc.hosts` file is part of the RCCE deliverable. It is an ASCII file that contains the numbers of all the cores on the tiles, one per line as follows.

```
01
02
03
.
.
.
45
46
47
```

The *pingpong* application returns immediately without running if you specify a number of cores other than two. [Figure 15](#) shows *pingpong* running successfully.

```

YourUsername@YourComputer$ rcцерun -emulator -nue 2 -f ../../hosts/rc.hosts pingpong
My rank is 0, physical core ID is 0
Core 0 passed RCCE_init
UE 0 has checked into barrier
My rank is 1, physical core ID is 1
Core 1 passed RCCE_init
UE 1 has checked into barrier
UE 0 has cleared barrier
UE 1 has cleared barrier
Round trip latency for 32 bytes = 0.022799890
pingpong 2 1.0 00 01
YourUsername@YourComputer$
```

Figure 15 The RCCE Application Running Successfully

10.2 Shared Memory

TBD