The SCC Programmer's Guide

revision 0.61

Please read the SCC Documentation Disclaimer on the next page.

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 IN-ABILITY 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.

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

Revision History for Document

0.6	Rewrote the Power Management section to describe new version of the Power Management API.
0.61	Rewrote the section on Building SCC Linux. The sources already exist on MCPC and do not need to be downloaded, and their directory structure does not contain rck.

Table of Contents

List of Tables

List of Figures

1 Introduction

The Single-chip Cloud Computer (the SCC) is a research chip created by Intel Labs to study many-core CPUs, their architectures, and the techniques used to program them. It has 24 dual-core tiles arranged in a 6x4 mesh. Each core is a P54C core and hence supports Intel architecture. For an overview of the SCC platform, refer to the *SCC Platform Overview*.

The SCC platform is a circuit board that contains the SCC chip, memory, and a system interface. The SCC usage model will evolve over time, but currently the SCC platform has two programming models.

The first model called the OS model runs a version of opensource Linux on each core. You load your application on one or more cores. The second model is called baremetal; the cores do not have an operating system. Your application runs directly on the cores without any operating system support. At this stage in SCC's lifetime, the OS model is the most mature. Section 2 The OS Model describes the OS model.

Section 3 SCC Architecture and Performance Considerations defines some terminology used when configuring and programming the SCC. It also discusses some performance considerations. Key SCC features are a large address space and a large number of IA cores that support a message-passing programming paradigm. A unique feature of the SCC is its ability to adjust the voltage and frequency of the tiles, both at startup and dynamically during operation. Refer to Section 5 Power Management.

With the current usage model, you connect a PC called the Management Console PC (MCPC) to the system interface on the SCC platform. The MCPC runs some version of an opensource Linux. Intel does not recommend a particular distribution, but Section 4 The Management Console describes what Intel has used and tested internally.

You then use Intel-provided software that runs on the MCPC to configure the SCC platform, compile your application, and then load your application on the SCC cores. Your application's I/O is through the MCPC. Section 4 The Management Console describes the software that runs on the Management Console.

However, it is important to note that this is an initial usage model. Intel Labs has designed the SCC with flexibility and potential in mind. How you actually use the SCC platform is determined by your own imagination and experience.

When you compile programs that run on the MCPC, you typically use gcc/g++. When you compile for the SCC platform, you should use an older version of icc/icpc. This Intel compiler is an older version because the SCC cores have the P54C architecture, the Pentium® architecture before the introduction of the streaming SIMD extensions (SSE) and out-of-order execution.

This document assumes that you are an experienced parallel programmer. Most likely you already have a parallel application that you are interested in porting to the SCC. Section 9 Porting an Application discusses some of the issues you may face when porting to the SCC platform, issues such as the lack of shared caching and the older P54C architecture.

This document also shows how to run some very simple "from-scratch" programs on the SCC platform. These programs range from a simple "hello-world" to short programs that read and write SCC configuration registers. Section 6 Some Simple SCC Programs describes

how to write, compile, and run such simple programs.

Intel also provides RCCE. RCCE (pronounced “rocky”) is a many-core communication environment for SCC application programmers. With RCCE, you can write message-passing application programs for either the OS model or baremetal. The RCCE package also contains a number of sample applications, ranging from simple (such as two cores just exchanging messages) to complex (such as high performance linpack). Section 8 Running RCCE Applications describes how to run some of these applications.

Section 7 Building RCCE describes how to build RCCE. You can build RCCE as an emulator or as a library intended for SCC hardware. When you use RCCE as an emulator, you can create RCCE applications that run on any standard Linux computer not connected to the SCC platform. The RCCE emulator is built on top of OpenMP.

The RCCE library is useful and highly performing in and of itself, but Intel also provides the complete source code for RCCE. You can look at how RCCE does what you want to do and write your own versions.

2 The OS Model

The MCPC contains an Intel-provided Linux image that runs on the SCC cores. You can build and use your own Linux image modeled after the one that Intel provides.

SCC Linux has been designed to run on the SCC cores. It will evolve as the SCC platform develops. For example, currently, I/O calls in a core program are redirected to a memory-mapped interface, and the output then appears at the Management Console.

You run the sccGui to configure the SCC platform. Configuring means training the system interface and the DDR3 memory on the SCC platform, setting values in SCC configuration registers, and loading Linux on one or more cores.

You can also perform these configuration actions from a command line on the MCPC, but the sccGui is currently more featured and mature.

3 SCC Architecture and Performance Considerations

Each of the 24 tiles has two cores. Each core has L1 and L2 caches. Each core has a 16KB LI instruction cache and a 16KB L1 data cache. The L1 caches are on the core. Each core also has a 256KB L2 cache. The L2 caches are on the tile.

Each tile also has a message passing buffer (MPB). This message passing buffer is 16KB of SRAM, local to the tile. This memory is shared among all the cores on the chip. Each tile has its own local MPB. Conventionally, it assigns 8KB to one of its cores and 8KB to the other. Although assigned to a particular core, the MPB is accessible to all cores. This convention is configurable. The total MPB for all the tiles is 384KB.

When a message-passing program sends a message from one core to another, internally it is moving data from the L1 cache of the sending core to its MPB and then to the L1 cache of the receiving core. The MPB allows L1 cache lines to move between cores without having to use the off-chip memory.

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. The maximum off-chip DRAM is 64GB. A core has a 32-bit address space and hence can address 4GB. A core accesses a *core address* which must be translated into a *system address*.

Each core also has a lookup table (LUT). This LUT translates the core address into a system address. A core also accesses its configuration registers via memory-mapped I/O, and the LUT translates a core address into the addresses needed for memory-mapped I/O.

System memory is divided into memory private to a core and memory shared by all the cores. Where this division occurs is determined by the core's LUT and the core's own pagetables. When you load SCC Linux on the cores, the LUTs are filled with default values. Refer to the *SCC EAS* for a listing of these default values. You can modify the values in the LUT dynamically after loading SCC Linux.

Message-passing data are typed as message-passing buffer type (MPBT). Data typed as MPBT bypass the L2 cache. If you write to data that are already resident in the cache, the cache line may (if L1 is configured as write-through) or may not (if L1 is configured as write-back) be moved to memory.

It is important to note that there is no cache coherence protocol among the cores. This means that when a core reads MPBT data, it gets the values in the L1 cache even when the data are stale. Cores may not get the latest MPBT data unless they invalidate the MPBDT data in their own LI cache.

To solve this problem, the SCC has a new instruction called CL1INVMB. This instruction invalidates all lines in the L1 cache that contain message buffer data, that is, data typed as MPBT.

A mesh interface unit (MIU) on each tile catches a cache miss and then using the LUT to decode the core address into a system address. If the data are typed MPBT, this is an L1 cache miss.

The LUT translates a 32-bit core address into a 46-bit system address. The most significant bit (bit 45) of the system address is the bypass bit. When the bypass bit is set, the memory access goes directly to the local MPB. The bypass bit speeds access to the local MPB. When the bypass bit is not set, the access is to the MPB (either local or non-local), a configuration register, or the off-chip DRAM.

The next eight bits (bits 44 through 37) determine the tile, which might be the same tile whose core is requesting access. The next three bits (bits 36 through 34) determine whether the access is to a non-local MPB, a configuration register, a memory controller, or the system interface. When the access is to off-chip DRAM, the lower 34 bits (bits 33 through 0) go to the memory controller on the specified tile. Each memory controller uses those 34 bits to address up to 16GB.

Messages employ XY routing. Messages go from the sending core to the receiving core first along the X direction and then along the Y direction. When this rule is followed, note that when a message goes back and forth between cores, the "back" path is different from the "forth" path.

3.1 Latencies

Table 1 lists some approximate latencies experienced when a core reads a 32-byte cache line. The table shows both core cycles and mesh cycles. A core and the mesh may run at the same frequency, but because of the SCC's power management capability, these frequencies may also be different. It is even possible that individual cores may run at different frequencies. In the table, core cycles refers to the cycles of the core making the request.

Latency Table	Approximate latency to read a cache line (output from the core to input back to core)
L2 access	18 core cycles
Local MPB access with bypass	15 core cycles
Local MPB access no bypass	45 core cycles + 8 mesh cycles
Remote MPB access	45 core cycles + 4*n*2 mesh cycles
DDR3 access	40 core cycles + 4*n*2 mesh cycles + 30 on-die memory controller (400MHz)+ 16 cycles(400MHz off-die DDR3 latency)
	n=number of hops to the MPB or the memory controller (0<n<10)

Table 1 Core Latency Table

3.2 processorID, tileID, and coreID

There are three IDs associated with a core: the processorID, the tileID, and the coreID. In Table 2 The SCC Mesh Showing tileIDs, processorIDs, and (x,y) Coordinates, the number in the lower left of each tile is the tileID. The (x,y) coordinates are in the lower right.

You can get this information by reading the TileID register. The lower 11 bits of this register are valid. Bits 10:07 contain the Y value; bits 05:03 contain the X value. Bits 02:00 contain the subID of the requesting agent. If the requesting agent is a core, its subID is the coreID.

If instead of running a program on the core to access the TileID register, you are using the sccGui, the coreID is always 101b. When you are using the sccGui, the requesting agent is the system interface whose subID is 101b.

Note that a core's tileID is not the same as the value of a core's TileID register. The tileID is an 8-bit value; the TileID register has 11 valid bits.

Because a core's tileID is an 8-bit hex value, it is not continuous. Its value is just the upper eight bits of the TileID register. If you have a core's (x,y) coordinates, get the tileID as 0x*yx*. Numerically, this is tileID = 16*y + x.

Calculate the processorID as ((x + (6 * y)) * 2) + coreID. The processorID goes from 0 through 47.

37 36 0x30 (0,3)	39 38 0x31 (1,3)	41 40 0x32 (2,3)	43 42 0x33 (3,3)	45 44 0x34 (4,3)	47 46 0x35 (5,3)
25 24 0x20 (0,2)	27 26 0x21 (1,2)	29 28 0x22 (2,2)	31 30 0x23 (3,2)	33 32 0x24 (4,2)	35 34 0x25 (5,2)
13 12 0x10 (0,1)	15 14 0x11 (1,1)	17 16 0x12 (2,1)	19 18 0x13 (3,1)	21 20 0x14 (4,1)	23 22 0x15 (5,1)
1 0 0x00 (0,0)	3 2 0x01 (1,0)	5 4 0x02 (2,0)	7 6 0x03 (3,0)	9 8 0x04 (4,0)	11 10 0x05 (5,0)

Table 2 The SCC Mesh Showing tileIDs, processorIDs, and (x,y) Coordinates

4 The Management Console

The Management Console is a PC that 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. Users typically VNC into the Management Console from their own workstation.

The management console (MCPC) runs some version of Linux. You can use any distribution you want. Intel Labs provides a number of software tools that run on the MCPC. Intel Labs uses and has tested its tools on Ubuntu 10.4.

You should ensure that your MCPC has **`pssh`** and Python. You can use **`pssh`** to load programs on the cores or execute commands on the cores. The script **`rccerun`** that loads RCCE applications on the cores uses **`pssh`** internally.

4.1 How to Get Copies of the Latest MCPC Tools

<TBD>
How to get the latest (a public SVN for source? a tar file?). How to install it (probably no more than just checking it out or untarring). How to update it (should you periodically check out or download again). Caution about modifying (checking out a new revision rev may overwrite local modifications).

Because the source is available, is there any mechanism for Intel Labs to accept or evaluate changes?

How do users get binaries for sccKit and install them.
</TBD>

4.2 Installing sccKit

The sccKit is distributed as three tar files: **`sccKit_base.tar.bz2`**, **`sccKit_libs.tar.bz2`**,

and **sccKit_1.1.0.tar.bz2**.

- **sccKit_base.tar.bz2** contains the basic sccKit components and will change infrequently.
- **sccKit_libs.tar.bz2** contains some needed Qt libraries used with sccGui.
- **sccKit_1.1.0.tar.bz2** contains the actual sccKit release.

You can install the sccKit anywhere you want, but typically it is installed in **/opt/sccKit**. When you untar these tar files in **/opt/sccKit**, the directory **1.1.0** appears. The directory name indicates the sccKit release number. The convention is to create a symbolic link to **1.1.0** called **current**. This convention allows you to easily switch between sccKit releases by redirecting the link.

Check that the file **systemSettings.ini** in sccKit's top-level directory contains the correct IP address of the SCC BMC. This address is printed on the top of the BMC. The port is always 5010.

To access the sccKit binaries, put **/opt/sccKit/current/bin** in your path. You also need to set the environment variable **LD_LIBRARY_PATH** to /**opt/sccKit/lib** so that you can access the shared Qt libraries. The script **setup** in sccKit's top-level directory sets your path and **LD_LIBRARY_PATH** for either the bash or tcsh shells.

4.3 Using sccKit

The sccKit has both a GUI and a command line. The capabilities of the sccGui overlap significantly but not completely with those available from the command-line.

For example, you can re-initialize the SCC platform by issuing the command, **sccBMC -i**. You can also bring up sccGui, click on the BMC tab, and choose (Re-)initialize platform.

To invoke the SCC GUI, issue the command, **sccGui&**. Figure 1 shows the initial SCC GUI screen.

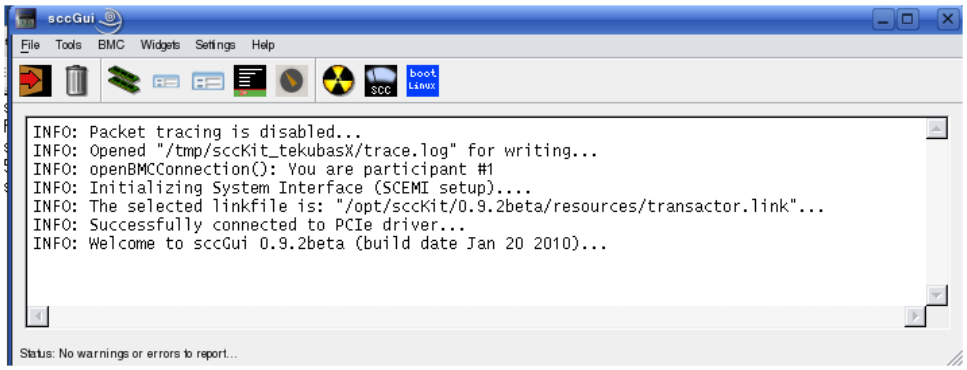

Figure 1 The SCC GUI

Table 3 lists the SCC functions available from the command line. Each command has online help. Issue the command with the **-h** option to get help information.

SccKit Command	Description
sccBMC	Initialize the SCC platform. Send commands to the BMC.
sccBoot	Boot Linux on one or more cores.
sccDump	Specify a tileID as in *yx* format and read memory or memory-mapped registers from that tile.
sccKonsole	Start up a console on one or more cores.
sccReset	Reset one or more cores. You can reset/release, reset, or release one or more cores.
sccPerf	Display the SCC performance meter.
sccMerge	An advanced command used by sccBoot.

Table 3 sccKit Commands

You can bring up SCC Linux on the cores with either **`sccGui`** or the command line. To load SCC Linux with the SCC GUI, just click on the blue Boot Linux button. Another window comes up that allows you to choose the cores on which you want to load SCC Linux. The default location of the SCC Linux image is **`/opt/sccKit/current/resources`**. You can change this default location by selecting Settings➔Linux boot settings➔Choose Linux image, as shown in Figure 2.

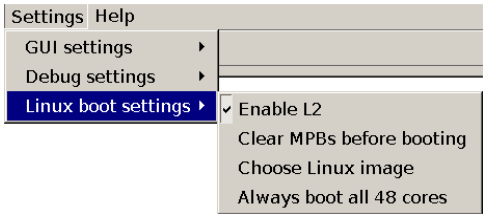

Figure 2 Changing the SCC Location with SCC GUI

When the SCC Linux boot is successful, the blue Boot Linux button changes to a green Linux okay button.

You can also boot SCC Linux from the command line with **`sccBoot`**. Without arguments, you just get the help message. Load SCC Linux with the **`-l`** option. Use the **`-g`** option to specify an SCC Linux image in a non-default location. For example, the command below loads **`mylinux.obj`** on cores 0 through 7.

```
sccBoot -l 0..7 -g /home/username/mylinux.obj
```

With the **`-s`** option, you can check that the cores have successfully booted.

```
username@mrllab1002:~$ sccBoot -s
INFO: Welcome to sccBoot 1.1.0 (build date Apr 29 2010)...
Status: The following cores can be reached with ping (booted): 8 cores
(PIDs = 0x00, 0x01, 0x02, 0x03, 0x0c, 0x0d, 0x0e and 0x0f)...
username@mrllab1002:~$
```

You can also ping the SCC cores. The cores are called `rck`*pid* where pid is the core's pid and goes from 0 to 47. The pid here is the same as the processorID shown in Table 2.

```
username@mrllab1002:~$ ping rck01
PING rck01.in.rck.net (192.168.0.2) 56(84) bytes of data.
64 bytes from rck01.in.rck.net (192.168.0.2): icmp_seq=1 ttl=64
time=0.240 ms
```

With the SCC GUI, you can look at the performance meter. Bring up the performance meter by selecting Widgets➔SCC Performance Meter as shown in Figure 3. You can also start up the SCC performance meter from the command line with **`sccPerf`**. Note, however, that sccPerf must connect to an X server. You cannot run it from an ssh window; either work directly on the MCPC or VNC into the MCPC (the more common method).

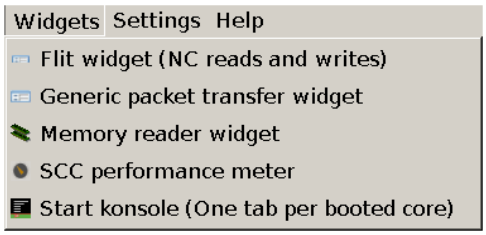

Figure 3 Selecting the SCC Performance Meter

You can tell that SCC Linux is running on a core if the core has a green arrow. Figure 4 shows the SCC performance meter with eight green arrows because SCC Linux is running on eight cores in the lower left 2x2 tile array.

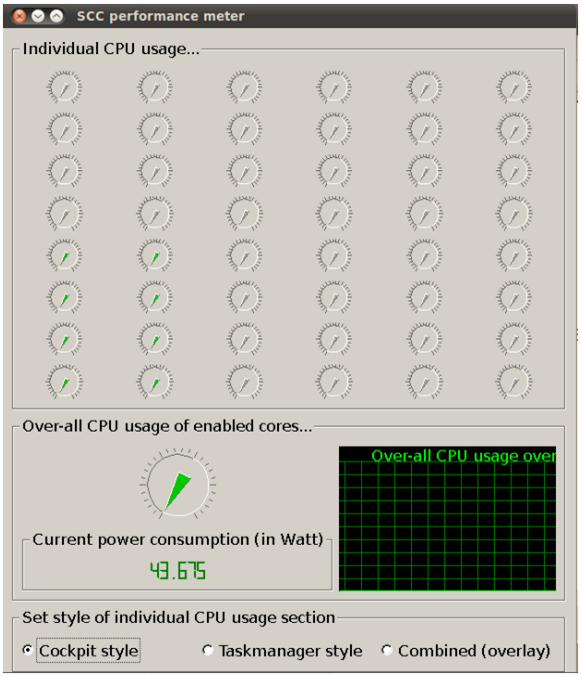

Figure 4 The SCC Performance Meter

When you click on the green Linux okay button, you can restart SCC Linux on the cores currently running SCC Linux. The "Boot Linux on selected cores …" window comes up with the cores currently running SCC Linux selected. You can choose at this point to reboot the cores that are running SCC Linux or to add additional cores.

If you deselect the cores in the lower left 2x2 array and select those in its adjacent 2x2 array, you boot SCC Linux on eight more cores. SCC Linux continues to run on the original eight cores, and so now you have SCC Linux running on 16 cores.

```
tekubasx@mrllab1002:~$ sccBoot -s
INFO: Welcome to sccBoot 1.1.0 (build date Apr 29 2010)...
Status: The following cores can be reached with ping (booted): 16
cores (PIDs = 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x0c,
0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12 and 0x13)...
```

Note that booting on additional cores does not affect the cores that already have SCC Linux running. To stop SCC Linux from running on a core, reset that core.. To do that, select Tools➔System reset as shown in Figure 5. By default, you reset all the cores. Select cores to reset with Tools➔Change selected reset(s).

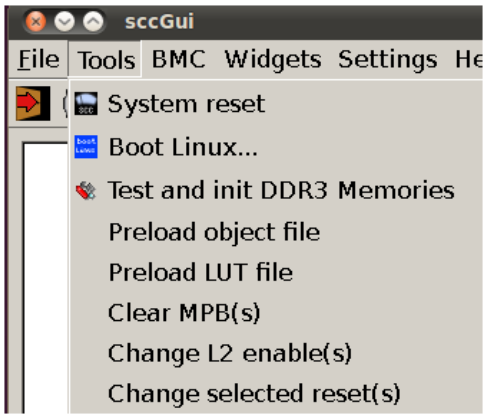

Figure 5 Selecting System Reset

With the command line, you can choose to reset individual cores. You can see what cores are in reset with the **-s** option. For example, the command below shows 32 cores in reset. This is because we have SCC Linux running on the two 2x2 arrays in the lower row; SCC Linux is running on 16 cores.

```
username@mrllab1002:~$ sccReset -s
INFO: Welcome to sccReset 1.1.0 (build date Apr 29 2010)...
Status: The following resets are active (pulled): 32 cores (PIDs =
0x08, 0x09, x0a, 0x0b, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b,
0x1c, 0x1d, 0x1e, 0x1f, 0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26,
0x27, 0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x2d, 0x2e and 0x2f)...
```

To reset the cores in the lower 2x2 array, issue two **sccReset** commands, one for the range 0..3 and the other for the range 12..15.. The **-p** resets the core and holds the reset. The pid numbers can be in decimal or hex, but when in hex, they should be preceded with 0x.

```
username@mrllab1002:~$ sccReset -p 0..3
INFO: Welcome to sccReset 1.1.0 (build date Apr 29 2010)...
INFO: Resets have been pulled: 4 cores (PIDs = 0x00, 0x01, 0x02 and
0x03)...
username@mrllab1002:~$ sccReset -p 0xc..0x0f
INFO: Welcome to sccReset 1.1.0 (build date Apr 29 2010)...
INFO: Resets have been pulled: 4 cores (PIDs = 0x0c, 0x0d, 0x0e and
0x0f)...
```

```
username@mrllab1002:~$
```

The **sccReset** command also gives you the option of releasing the reset with the **-r** option. The example below releases the reset on cores 0 though 3.

```
tekubasx@mrllab1002:~$ sccReset -r 0..3
INFO: Welcome to sccReset 1.1.0 (build date Apr 29 2010)...
INFO: Resets have been released: 4 cores (PIDs = 0x00, 0x01, 0x02 and
0x03)...
tekubasx@mrllab1002:~$
```

When you release the reset on a core, SCC Linux then boots again. Notice that now SCC Linux runs on 12 cores.

```
tekubasx@mrllab1002:~$ sccBoot -s
INFO: Welcome to sccBoot 1.1.0 (build date Apr 29 2010)...
Status: The following cores can be reached with ping (booted): 12
cores (PIDs = 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x10,
0x11, 0x12 and 0x13)...
tekubasx@mrllab1002:~$
```

With sccKonsole, you can open up an ssh connection to one or more cores. To start a console on cores whose pids are 0,1,2,3, issue

```
sccKonsole 0..3
```

Each tabbed konsole is a separate connection. If you want input from one konsole to be recognized by another, select Edit➔Copy Input To➔All Tabs in Current Window. Figure 6 shows the four tabbed windows created with sccKonsole 0..3. The konsole **rck01** was configured to send its input to all konsoles; that's why the red !. The konsole **rck01** is selected in and shows an **ls -a** command that was issued in **rck01**.

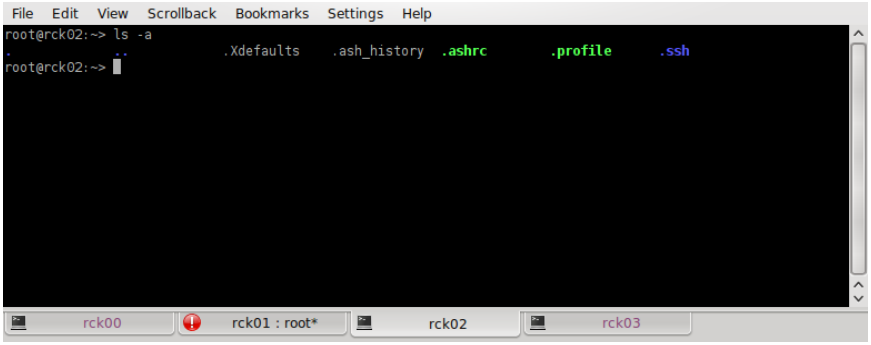

Figure 6 Result of sccKonsole 0..3

You can also create tabbed konsoles with the SCC GUI. Bring up tabbed konsoles by selecting Widgets➔Start konsole (one tab per booted core) as shown in Figure 3. With the SCC GUI, you always create one tab per booted core. Use the command line if you want to select what cores get a konsole. As with **sccPerf**, **sccKonsole** requires connection to an X server.

The command **sccDump** is useful when debugging. With **sccDump** you can read memory (off-chip DRAM), the message-passing buffer, the configuration registers, and the lookup tables. It corresponds to the memory reader widget in the SCC GUI. Select the memory reader

widget with Widgets➔Memory reader widget as shown in Figure 3.

For example, to display the configuration registers and lookup tables for the tile in the upper left corner and save the output to a file called **sccDump_c_0x31.txt**, enter

```
sccDump –c 0x30 –f sccDump_c_0x30.txt
```

sccDump identifies the tile with its tileID. Recall that the tileID is an 8-bit number with the y coordinate in the upper four bits and the x coordinate in the lower four bits; the tileID is not continuous over the mesh. Instead of 0x30, you could have specified the decimal number 38.

The default LUT configuration==========

```
============================================================
Dumping CRB registers of Tile 0x30
============================================================
GLCFG0   = 0x00348df8
GLCFG1   = 0x00348df8
L2CFG0   = 0x000006cf
L2CFG1   = 0x000006cf
SENSOR   = 0x00000000
GCBCFG   = 0x00a8e2f0
MYTILEID = 0x00000185
LOCK0    = 0x00000001
LOCK1    = 0x00000001
------------------------------------------------------------
Restoring locks: LOCK0 and LOCK1
============================================================
Dumping LUTs of Tile 0x30
Format: Bypass(bin)_Route(hex)_subDestId(dec)_AddrDomain(hex)
============================================================
LUT0, Entry 0x00 (CRB addr = 0x0800): 0_0x20_6(PERIW)_0x078
LUT0, Entry 0x01 (CRB addr = 0x0808): 0_0x20_6(PERIW)_0x079
LUT0, Entry 0x02 (CRB addr = 0x0810): 0_0x20_6(PERIW)_0x07a
LUT0, Entry 0x03 (CRB addr = 0x0818): 0_0x20_6(PERIW)_0x07b
LUT0, Entry 0x04 (CRB addr = 0x0820): 0_0x20_6(PERIW)_0x07c
LUT0, Entry 0x05 (CRB addr = 0x0828): 0_0x20_6(PERIW)_0x07d
LUT0, Entry 0x06 (CRB addr = 0x0830): 0_0x20_6(PERIW)_0x07e
LUT0, Entry 0x07 (CRB addr = 0x0838): 0_0x20_6(PERIW)_0x07f
LUT0, Entry 0x08 (CRB addr = 0x0840): 0_0x20_6(PERIW)_0x080
LUT0, Entry 0x09 (CRB addr = 0x0848): 0_0x20_6(PERIW)_0x081
LUT0, Entry 0x0a (CRB addr = 0x0850): 0_0x20_6(PERIW)_0x082
LUT0, Entry 0x0b (CRB addr = 0x0858): 0_0x20_6(PERIW)_0x083
LUT0, Entry 0x0c (CRB addr = 0x0860): 0_0x20_6(PERIW)_0x084
LUT0, Entry 0x0d (CRB addr = 0x0868): 0_0x20_6(PERIW)_0x085
LUT0, Entry 0x0e (CRB addr = 0x0870): 0_0x20_6(PERIW)_0x086
LUT0, Entry 0x0f (CRB addr = 0x0878): 0_0x20_6(PERIW)_0x087
LUT0, Entry 0x10 (CRB addr = 0x0880): 0_0x20_6(PERIW)_0x088
LUT0, Entry 0x11 (CRB addr = 0x0888): 0_0x20_6(PERIW)_0x089
LUT0, Entry 0x12 (CRB addr = 0x0890): 0_0x20_6(PERIW)_0x08a
LUT0, Entry 0x13 (CRB addr = 0x0898): 0_0x20_6(PERIW)_0x08b
LUT0, Entry 0x14 (CRB addr = 0x08a0): 0_0x64_1(CORE1)_0x141
                                    :
                                    :
LUT0, Entry 0xfe (CRB addr = 0x0ff0): 1_0x88_0(CORE0)_0x0a4
LUT0, Entry 0xff (CRB addr = 0x0ff8): 0_0x20_6(PERIW)_0x0fa
LUT1, Entry 0x00 (CRB addr = 0x1000): 0_0x20_6(PERIW)_0x08c
```

```
LUT1, Entry 0x01 (CRB addr = 0x1008): 0_0x20_6(PERIW)_0x08d
```

<TBD>

Need a short discussion of sccDump. sccMerge is an advanced command. Who would use sccMerge and when would they use it?

</TBD>

4.4 MCPC Tools

In addition to the sccKit, MCPC tools consist of C and Fortran cross compilers, a version of the Math Kernel Library (MKL), an **`.ssh2`** directory, and an older version of **`icc`**.

You can use any C compiler you want for programs intended to run on the MCPC. Typical C compilers are **`gcc/g++`** and the latest version of Intel's **`icc/icpc`**. The RCCE emulation library builds and runs with either compiler.

The C and Fortran cross compilers are older versions of Intel's compilers that work on the P54C architecture. The P54C architecture is the Pentium architecture before the introduction of streaming SIMD instructions and out-of-order execution. The MKL is also an older version for use on the P54C architecture.

If you are building an application to run on the cores themselves, you must use the Intel-provided cross compilers. Table 4 lists the versions that work with the SCC platform.

Cross Compiler/Library	Version
`gcc/g++`	3.4.5
`icc/icpc`	8.1.038
`ifort`	8.1.034
`mkl`	8.1.1.004

Table 4 Cross Compiler and Library Versions that Run on the SCC Cores

Ensure that you have an acceptable **`.ssh2`** directory. SCC applications run on the cores as root, and the **`.ssh2`** directory lets that happen.

The sccKit consists of the sccGUI and some command line utilities. With the sccGui, you can perform such actions as booting Linux on one or more cores, reading memory, and reading/writing configuration registers.

4.5 Installing MCPC Tools

<TBD>

Tar files? Default location. There isn't really a using MCPC Tools section. Using them is Sections 6 and greater.

</TBD>

5 Power Management

The SCC platform contains a voltage regulator controller (VRC) that allows you to

independently change the voltage of an eight-core voltage island. You can also change the frequency of individual cores. You can do this dynamically from within a program running on the cores.

When you initialize the SCC platform, you can choose the initial frequency settings. Do this by selecting the re-initialize button. Refer to Figure 7 Initialize the SCC Platform. The re-initialize button is the black-and-yellow button that looks like a radiation hazard warning.

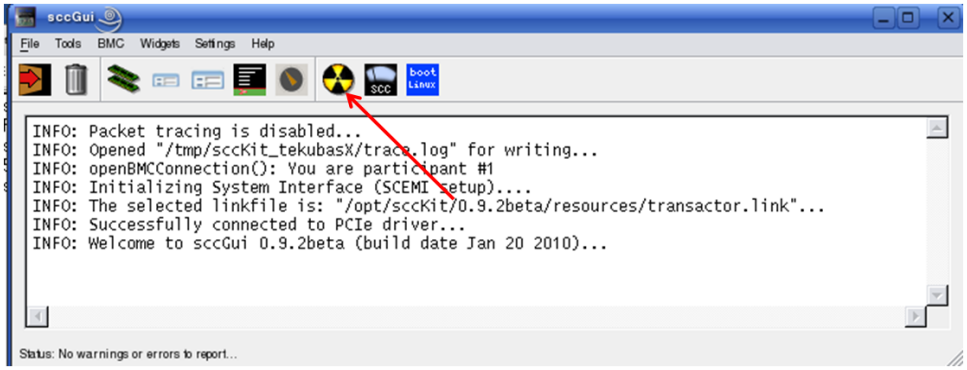

Figure 7 Initialize the SCC Platform

When you click the re-initialize button, you get the screen in Figure 8. With the dropdown box, you can choose from among the following five settings. The numbers refer to clock frequencies measured in MHz. Core is the frequency of the core; router is the frequency of the mesh; and MC is the frequency of the memory. The default is **`Core533_Router800_MC800`**. It specifies that the cores run at 533MHz and that the mesh and the memory run at 800MHz. The voltage for all settings is nominally 1.1v. Choose **`Core533_Router800_MC800`** for normal operation.

Core533_Router800_MC800 (Default)
Core800_Router800_MC800
Core800_Router800_MC1066
Core800_Router1600_MC800
Core800_Router1600_MC1066

To see the actual voltages for the power domains, you can telnet into the BMC from the MCPC, specifying port 5010. Then, issue the status command.

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 just have to know it. It's most likely written on a sticker attached to the BMC.

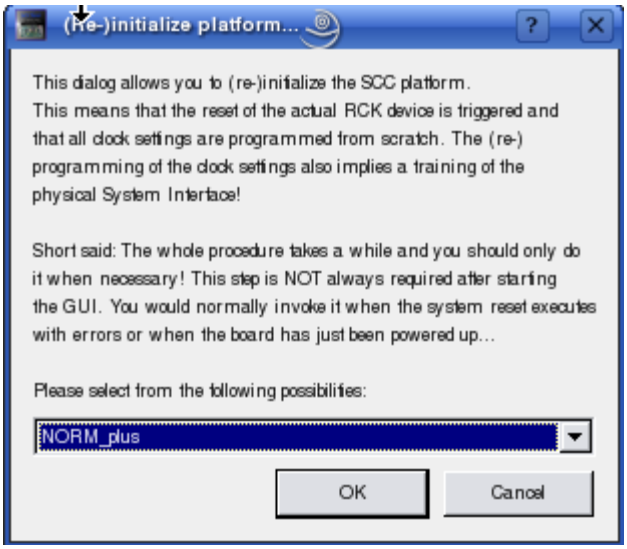

Figure 8 Choosing a Voltage/Frequency Setting

The goal is, of course, to improve performance while minimizing power and not killing the chip. Running the cores at a frequency that is too high for the existing voltage may cause the system to crash.

Dynamic power is proportional to frequency * voltage squared. Here are some guidelines.

- If your program is computation CPU bound, its performance is proportional to the frequency.
- Only reduce the frequency for programs that are not computation bound; that is, programs that are accessing I/O or memory.
- Try to minimize both frequency and voltage. If you cannot, minimize the frequency.
- Remember to take into account the latency when changing the voltage. This latency is on the order of milliseconds. The latency for changing the frequency is much smaller, about 20 cycles.

SCC cores are divided into seven voltage domains. Six of those are 2x2 arrays of tiles, as shown in Figure 9. The seventh is the entire mesh. Currently, the RCCE API does not provide the ability to access the "entire-mesh" domain. When you change the voltage, you choose a voltage domain and change the voltage for all the cores in that domain.

The SCC has 24 frequency domains, one for each core. You can change the frequency of each individual core. However, the power management calls in the RCCE API do not currently support the control of all frequency domains independently. Instead RCCE only allows stepping of the frequency on the cores within a voltage domain. Effectively for RCCE, frequency and voltage domains coincide and are jointly called *power domains*.

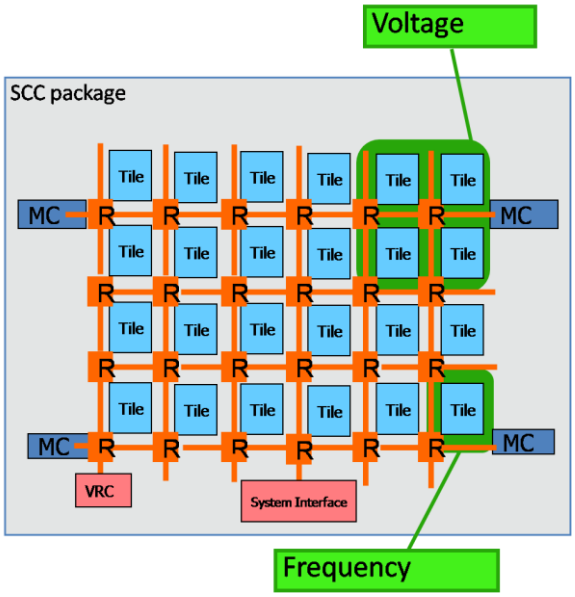

Figure 9 Voltage and Frequency Domains

One usage possibility is initially to set up areas of voltage and frequency values and then move tasks to the area where it has the best performance while minimizing power consumption. Because tasks share no state, they can easily be mapped to different cores.

You can control the voltage of a voltage domain by writing the VRC control register. You can change the frequency by writing the GCU (Global Clock Unit) register. The best way, however, is to use the power management API that RCCE provides. See Section 8.7 Power Management for information about using the RCCE power management functions.

6 Some Simple SCC Programs

This section shows how to run some simple C programs on the SCC platform. These programs are not RCCE programs, so you should not load them with the **`rccerun`** script that comes with RCCE. **`rccerun`** makes some assumptions that do not hold for arbitrary SCC programs. Rather you should use **`pssh`** (the parallel ssh), which is available at http://freshmeat.net/projects/pssh/

This section shows two examples. The first is just a "hello world." Its purpose is to show how you would create the **`pssh`** hosts file and then compile and load the application. The second example is also very simple, but more realistic. It shows how you would read and write a core configuration register.

6.1 Hello World

This example assumes that you've loaded SCC Linux on at least two cores, namely cores 00 and 01.

Make a **`pssh`** hosts file. Name it anything you want. The format must be as follows

```
rck00 root
rck01 root
rck02 root
```

```
rck03 root
rck04 root
rck05 root
    :
```

Each line has two fields. The first identifies the core as **rck***processorID*, and the second identifies the user on the cores. The user is always root.

The code is in the file **SCChello.c**. It just includes **stdio.h** and has a **main()** that calls **printf()**. Compile it as follows.

```
icpc -DCOPPERRIDGE -static -mcpu=pentium -gcc-version=340 helloSCC.c
```

The macro **COPPERRIDGE** specifies that you are compiling for SCC hardware. **COPPERRIDGE** is a legacy codename that is no longer relevant to SCC hardware, but it persists in the code. The switch **-mcpu=pentium** indicates that you are compiling for the P54C architecture. For this reason, you must also use an older version of **icc** (8.1.038).

```
YourUsername@YourComputer:/shared/YourUsername> which icc
/shared/icc-8.1.038/bin/icc
YourUsername@YourComputer:/shared/YourUsername>
```

icc/icpc does require that you have **gcc/g++** installed. The version of **icc/icpc** that you need for SCC is validated for gcc 3.4.0. You can use a later version of **gcc**. The MCPC that this example ran on has **gcc** 4.1.2; but the **icc/icpc** 8.1 validation stopped at **gcc** 3.4.0. You must use the switch **-gcc-version=340** to ensure that **icc/icpc** does not use any **gcc** features beyond 3.4.0.

Copy the resulting **a.out** to the **/shared** directory on the MCPC. Typically users create a subdirectory under **/shared** and name it with their username. Then, use **pssh** to load **a.out** on SCC cores.

How many cores you load the program on depends on the pssh hosts file and the **pssh -p** switch. In this example the pssh hosts file has only two lines

```
rck00 root
rck01 root
```

and **-p** specifies two cores. The switch **-p** actually specifies the maximum number of cores. So you could have entered **-p 4**, and the program would still load and run, but run on only two cores because the **pssh.hosts** file specifies just two.

```
YourUsername@YourComputer:/shared/YourUsername> pssh -h pssh.hosts -t
-1 -P -p 2 /shared/YourUsername/a.out
rck00: hello
rck00: [1] 08:43:43 [SUCCESS] rck00 22
rck01: hello
rck01: [2] 08:43:43 [SUCCESS] rck01 22
YourUsername@YourComputer:/shared/YourUsername>
```

Note that the **pssh** command line specifies the full pathname for **a.out**, even though **a.out** is in the working directory. If you don't specify the full pathname, you get an error.

```
YourUsername@YourComputer:/shared/YourUsername> pssh -h pssh.hosts -t
-1 -P -p 2 a.out
```

```
rck00: sh: rck01: sh: rck00: rck00: a.out: not found
rck01: a.out: not found
rck01: [1] 08:46:51 [FAILURE] rck00 22 Received error code of 127
[2] 08:46:51 [FAILURE] rck01 22 Received error code of 127
YourUsername@YourComputer:/shared/YourUsername>
```

The **-t** switch specifies the timeout in seconds, and -1 means it never times out. The **-P** switch specifies that the program prints output as it is received.

6.2 Reading and Writing Core Configuration Registers

This example shows how a core program can access core configuration registers. It can access its own configuration registers as well as those of other cores using memory-mapped I/O. The core program performs memory-mapped I/O in the standard Linux way using the **mmap()** function. A good reference for how to perform memory-mapped I/O is *Advanced Programming in the Unix Environment* by W. Richard Stevens and Stephen A. Rago.

The configuration registers for each tile are given a base address in the core's LUT. The RCCE header file **config.h** defines macros for these base addresses. Realize, however, that these are the base addresses that result from the default LUT configuration. Note that there is a special address that a core can use to identify its own base address.

The base address for the configuration registers for the tile at (x=0, y=0) is 0xe0000000. The configuration registers for each tile are offset by 0x01000000 from 0xe0000000 as you travel along the x axis. Following this convention, the base address for the tile at (x=1, y=0) is 0xe1000000, that for the tile at (x=2, y=0) is 0xe2000000, etc. The tile after (x=5, y=0) is (x=0, y=1), etc. Continuing with this method, the base address for the final tile at (x=5, y=3) is 0xf7000000. Table 5.shows the base addresses for the configuration registers for the SCC tiles.

Base Address	Tile (x,y)
F8000000	Base address for Calling Core
F7000000	System Configuration Registers -- Tile (x=5,y=3)
F6000000	System Configuration Registers -- Tile (x=4,y=3)
:	:
E2000000	System Configuration Register s-- Tile (x=2,y=0)
E1000000	System Configuration Registers -- Tile (x=1,y=0)
E0000000	System Configuration Register s-- Tile (x=0,y=0)

Table 5 Base Addresses for Core Configuration Registers

The base address 0xf8000000 is the special one. When a core specifies this base address, it specifies its own base address. A core can reference its own base address in this way to read its own TileID register and obtain its own (x,y) coordinates, tileID, coreID, and processorID. Appendix contains a sample code listing that a core can use to read its TileID register. The

The key points of that program are shown below. For the file descriptor of the file to be mapped, use the device **/dev/rckncm**. Open the device **/dev/rckncm** and get its file descriptor.. Call **mmap()** and specify this file descriptor and map a page. Dereference the returned pointer to get the value of the TileID register.

```
typedef volatile unsigned char* t_vcharp;
int          tileID;
```

```
unsigned int alignedAddr, pageOffset;
t_vcharp     MappedAddr;
           :
if ((NCMDeviceFD=open("/dev/rckncm", O_RDWR|O_SYNC))<0) {
   perror("open"); exit(-1);
}

alignedAddr = 0xf8000000;
pageOffset  = 0x100;

MappedAddr = (t_vcharp) mmap(NULL, PAGE_SIZE, PROT_WRITE|PROT_READ,
   MAP_SHARED, NCMDeviceFD, alignedAddr);
if (MappedAddr == MAP_FAILED) {
   perror("mmap"); exit(-1);
}
tileID = *(int*)(MappedAddr+pageOffset);
             :
```

Compile with **icpc**. Then copy the resulting executable to **/share/YourUsername**.

```
icpc -DCOPPERRIDGE -static -mcpu=pentium -gcc-version=340 readTileID.c
```

As with the "hello-world" example, the file **pssh.hosts** determines what processors the program runs on. For this example, **pssh.hosts** looks as follows.

```
rck10 root
rck11 root
```

This file causes the program to run on processors 10 and 11. Run the program with the **pssh** command.

```
YourUsername@YourComputer:/shared/YourUsername> pssh -h pssh.hosts -t
-1 -P -p 2 a.out
```

The output called tileID prints first in hexadecimal and then in decimal.

```
rck11: tileID = 29 41
rck11: [1] 17:33:16 [SUCCESS] rck11 22
rck10: tileID = 28 40
rck10: [2] 17:33:16 [SUCCESS] rck10 22
```

From the tileID that is returned you can obtain the tile's (x,y) coordinates and the core's coreID by shifting and masking. Then, the core's tileID is 16*y + x. Its processorID is (6*y + 2)*2 + coreID.

The (x,y) coordinates for this example are (5,0). The TileID register has y in bits 10:07, x in bits 06:03 and the coreID (which is 0 or 1) in bits 02:00. To be completely accurate, bits 02:00 contain the subID of the requesting agent; and if the requesting agent is a core, its subID is the coreID. Note, though, that if a core reads the TileID register of another core, it is the core doing the read that is the requesting agent.

If you are using the sccGui to access a TileID register, the requesting agent is the System Interface whose subID is 101b.

The TileID register for processorID 10 is 0x28, and the Tile ID register for processorID 11 is 0x29. Table 6 shows the bit pattern of the TileID register and how the values 0x28 and 0x29

populate its fields

Y	X	CoreID	
1 0987	6543	210	Hex Value
0000	0101	000	0x28
0000	0101	001	0x29

Table 6 Bit Pattern for the TileID Register

7 Building RCCE

Get the latest RCCE source code using whatever method is best for you. This may be checking RCCE out from an Intel-provided SVN repository or downloading a tar file from an Intel website. Figure 10 shows what RCCE contained at a particular snapshot in time.

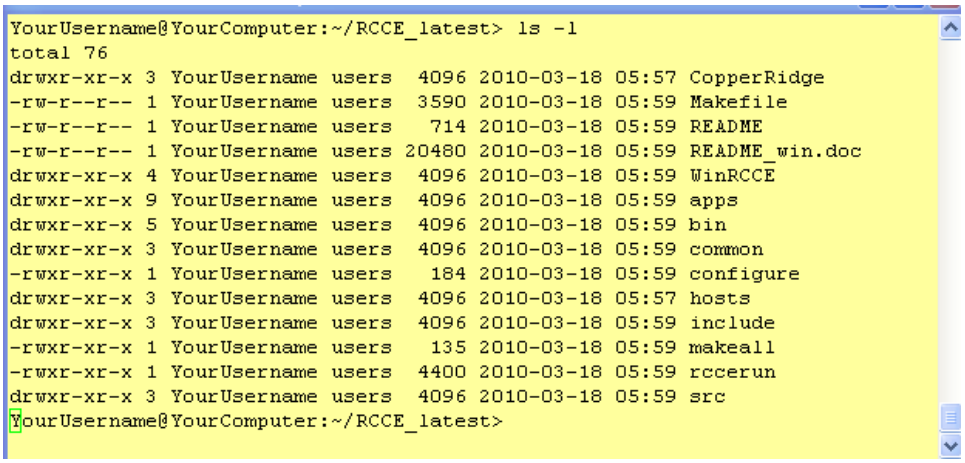

Figure 10 Contents of RCCE

Type **`./configure <PLATFORM>`**. Where **`<PLATFORM>`** is either **`SCC`** or **`emulator`**. Use **`SCC`** when you are building for SCC hardware; use **`emulator`** when you are building the RCCE emulator.

The command is silent. It modifies the **`common/symbols`** file to make it appropriate for your installation.

The **`symbols`** file sets compiler flags and determines whether to build RCCE for SCC hardware or the emulator. The default is to build all the libraries for the emulator.

Do not build the RCCE emulator to run on the Management Console. The Intel compiler on the MCPC is really a cross compiler for the older P54C architecture.

7.1 Building the RCCE Emulator

To build all the libraries for the emulator, use the **`makeall`** command . This means that if

you are on your Linux desktop and you just type **./makeall**, you will build the following libraries under **bin/OMP**.

```
YourUsername@YourUsername-desktop:~/RCCE_latest$ ls bin/OMP
libRCCE_bigflags_gory_nopwrmgmt.a     libRCCE_smallflags_gory_nopwrmgmt.a
libRCCE_bigflags_gory_pwrmgmt.a       libRCCE_smallflags_gory_pwrmgmt.a
libRCCE_bigflags_nongory_nopwrmgmt.a  libRCCE_smallflags_nongory_nopwrmgmt.a
libRCCE_bigflags_nongory_pwrmgmt.a    libRCCE_smallflags_nongory_pwrmgmt.a
YourUsername@YourUsername-desktop:~/RCCE_latest$
```

Figure 11 List of RCCE Emulator Libraries

There are eight libraries. You have a library with bigflags and a library with smallflags. Each of those can have either a gory or a nongory interface, and each interface can then either include or not include the power management calls.

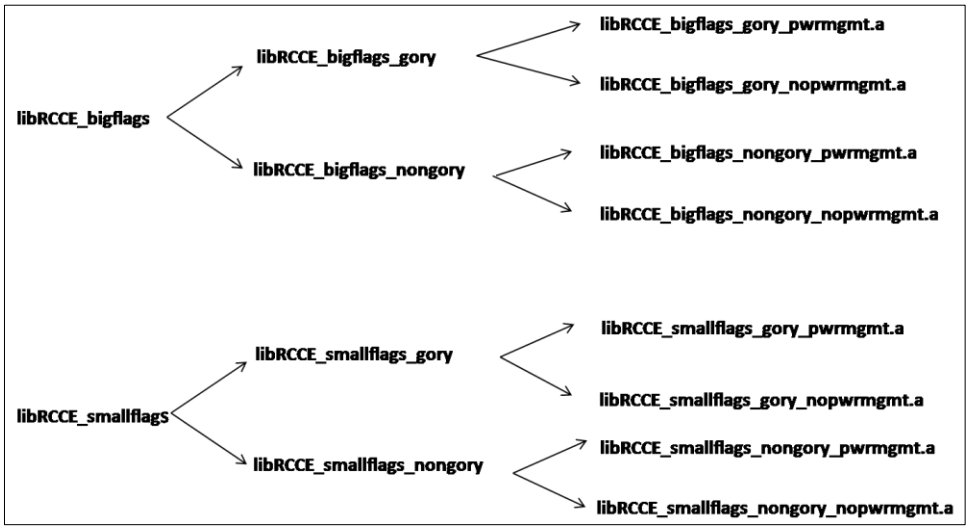

Figure 12 RCCE Emulation Libraries

Note that RCCE routines may use some C99 syntax, and so you may need to modify the **CCOMPILE** line in **common/symbols** to include the switch **-std=c99** so that these routines can build. This is the **CCOMPILE** definition when **OMP_EMULATOR** is 1, which is its default.

```
ifeq ($(OMP_EMULATOR),0)
  CCOMPILE=/shared/icc-8.1.038/bin/icpc
  SUBDIR=RCK
  PLATFORMFLAGS=-DCOPPERRIDGE -static -mcpu=pentium -gcc-version=340 -I../include
else
  CCOMPILE=icc -std=c99
  SUBDIR=OMP
  PLATFORMFLAGS=-openmp
endif
```

Figure 13 Adding a Flag to icc in common/symbols

Refer to Section 7.3 RCCE Build Options for information about the difference between bigflags and smallflags, the difference between the gory and nongory interfaces, and whether to include the power management API.

To build a subset of the RCCE emulation libraries, use the **make** command and choose the appropriate option.

```
make [OMP_EMULATOR=(1,0)][PWRMGMT=(0,1)]
```

```
[API=(nongory,gory)][SINGLEBITFLAGS=(0,1)]
```

Here the [ ] indicate an optional parameter and the embedded ( ) indicate the choices for that optional parameter. The parameter is optional in the sense that you don't have to specify it, but when you do not specify it, it has a default value. The default value is the first of the parameter choices. So if you just type **make**, you are choosing to build the RCCE emulation library without the power management API, without the gory interface, and without single bit flags (bigflags).

Type **make** and you get the library **libRCCE_bigflags_nongory_nopwrmgmt.a** in **bin/OMP**.

7.2 Building RCCE for SCC Hardware

Build the RCCE library for SCC hardware on the MCPC. You don't have to build it on the MCPC, but if you do, you are assured of access to the correct cross-compiler. You are building for the SCC cores, not the MCPC.

Use **make**, not **makeall**. If you issue **makeall**, you will default to building the RCCE emulator and most likely fail because the Intel compiler in your path is that for the SCC's P54C architecture, not the MCPC itself.

Type **make OMP_EMULATOR=0**. And then choose from the remaining options.

```
make OMP_EMULATOR=0 [PWRMGMT=(0,1)]
[API=(nongory,gory)][SINGLEBITFLAGS=(0,1)]
```

For example,

```
make OMP_EMULATOR=0 PWRMGMT=1 API=nongory
```

creates the RCCE library for SCC hardware with the power management API, the nongory (basic) interface, and not singlebitflags. This library is created in the directory **bin/SCC**.

```
YourUsername@YourComputer:~/RCCE_latest> ls bin/RCK
libRCCE_bigflags_nongory_pwrmgmt.a
YourUsername@YourComputer:~/RCCE_latest>
```

Figure 14 RCCE Library for SCC Hardware

Refer to Section 7.3 RCCE Build Options for information about the difference between bigflags and singlebitflags, the difference between the gory and nongory interfaces, and whether to include the power management API.

7.3 RCCE Build Options

The gory and nongory interfaces are aptly named. If you want to write RCCE applications and get down into the nitty gritty of how message passing is implemented, use the gory interface (**API=gory** on the make line). Otherwise choose, the nongory interface (**API=nongory** on the make line).

Flags are used to coordinate interaction between units of execution. You can choose flags to have low latency and be somewhat wasteful of memory or flags that have a higher latency

and consume less memory. The memory referred to here is message passing buffer memory.

The first choice (lower latency, higher memory use) occurs when you specify **SINGLEBITFLAGS=0** on the **make** command line. With bigflags, each flag takes up an entire 32-byte cache line. The second choice (higher latency, lower memory use) occurs when you specify **SINGLEBITFLAGS=1** on the **make** command line. With singlebitflags, flags are stored as a single bit. A cache line contains many flags.

You can choose to include the power management routines in the RCCE library by adding **PWRMGMT=1** on the **make** command line. The default is not to include the power management routines; that is, **PWRMGMT=0.** Having the power management routines in the library does not affect the performance of the RCCE library, but sometimes you want the library to include only what you plan on using.

8 Running RCCE Applications

The SCC Platform Overview showed how to run an RCCE example using the RCCE emulator. The example was pingpong, which just sends messages back and forth between two cores.

The RCCE distribution contains several examples in its apps directory. The point of entry for all units of execution (UEs) into a RCCE applications is **RCCE_APP()**. Recall that a UE refers to a process, thread or other agent of execution that moves the program counter forward. For RCCE applications, the number of UEs is the number of cores (when running on SCC hardware) or the number of simulated cores (when running under the RCCE emulator).

The call **RCCE_APP()** is really intended for the RCCE emulator, which runs on top of OpenMP. When you are programming for actual SCC hardware, you can replace **RCCE_APP()** with **main()**. You do not have to perform this replacement because the file **RCCE.h** does it for you as shown below. When you are running the RCCE emulator, **_OPENMP** is defined.

```
// little trick to allow the application to be called "RCCE_APP" under
// OpenMP, and "main" otherwise
#ifndef _OPENMP
  #define RCCE_APP main
#endif
```

To start a RCCE application, use **rccerun**. The options for **rccerun** are as follows.

```
rccerun -nue nbrUEs [-f hostfile] [-clock GHz] executable [parameters]
```

You must specify the number of UEs. Most of the time you must specify a hostfile. The hostfile is a file that lists the processorIDs of the cores that rccerun will load the application on. It contains one line for each SCC core as follows.

```
00
01
02
03
:
```

When **rccerun** loads your RCCE application, it chooses the number of cores specified by **-nue** starting with the processorID on the first line.

If you are at the root of your RCCE installation, you need not specify a hostfile. **rccerun** will use the default hostfile in hosts called **rc.hosts**. Otherwise, specify the pathname (which can be relative) of the hostfile you want to use.

-clock specifies the frequency that RCCE uses for timing measurements. It does not change the actual frequency of the SCC tiles. If you specify a frequency with **-clock** on the **rccerun** command line that is different from the actual tile frequency, the timing results returned by RCCE examples will be incorrect.

rccerun either uses the RCCE emulator to run your application or loads you application on SCC hardware, depending on what you set **<PLATFORM>** to when you ran **configure**.

Specify the name of your RCCE executable. If this RCCE executable is not in your current working directory, give its pathname, which can be relative.

8.1 Characteristics of RCCE Programs

From a programmer's point of view, RCCE applications are message passing applications. Because of the lack of cache coherence among the cores, the programming model is most naturally and most efficiently based on the ability to send messages between the cores.

Recall that SCC has an on-chip message-passing buffer (MPB). Each of the 24 tiles has 16KB of MPB, totaling 384KB for the chip itself. Each of the 48 cores is assigned 8KB of this MPB, but each core has access to the entire MPB. A core can send a message to another core by moving data from its own L1 cache to the MPB. Then, the receiving core can take the data from the MPB and move it into its own L1 cache.

Note that messages are passed from one core to another without a core having to use any off-chip memory. Note also that messages bypass a core's L2 cache. The sending core *puts* the message from its L1 cache into the sending core's MPB. Then, the receiving core *gets* the message from the sending core's MPB.

However, referring to the MPB as sender's MPB and receiver's MPB is somewhat artificial because the MPB address space is accessible by all cores. RCCE manages the MPB by assigning 8KB regions to each core, but any core can write anywhere in the MPB.

As described above, RCCE is based on one-sided put and get primitives. RCCE also provides two-sided synchronous message passing calls. Internally, these high-level message passing calls are implemented as one-sided primitives with flags to control access to the MPB.

8.2 Two Important Cautions When Using RCCE

The first cautions against assuming that the initial state of the message passing buffer and test-and-set registers are clean. The second cautions against assuming that the synchronizing calls with empty messages actually synchronize.

8.2.1 Initial State of the Message Passing Buffers and Test-and-Set Registers

There is no guarantee that the MPBs are in a clean state, at the beginning of a RCCE execution. You can explicitly wipe the MPBs by executing **`mpb -c`** on the cores. Run it on each core whose MPB you want to clear. You can also use the SCC GUI and select Tools➔Clear MPB(s).

Similarly, there is no guarantee that the test-and-set registers are in a clean state. You can reset the test-and-set registers by executing **`mpb -cl`** on the cores. Run it on each core whose test-and-set register you want to clear.

If the application dies and needs to be killed, the state of all registers and on-chip memory is indeterminate. However, even in a correctly executing code, it is possible to leave debris in the MPBs and the test-and-set registers.

8.2.2 Empty Messages do not Synchronize

Note that if the synchronizing calls, **`RCCE_send()`** and **`RCCE_recv(,`** have empty messages, they are not synchronizing. If the message size is zero, the send and receive calls are effectively no-ops and hence senders and receivers do not need to be matched.

This is different from MPI where even an empty message is not really empty. In MPI, the payload is accompanied by a header, allowing MPI programmers to use an empty message for synchronization. RCCE communication calls only have payload. When the message is null, there is no need to communicate and no synchronization occurs. With RCCE, do not use **`RCCE_send()`** and **`RCCE_recv()`** with an empty message for synchronization

8.3 RCCE has Basic and Gory Interfaces and Power Management

The RCCE libraries are distributed as C source code. Recall from Building RCCE that you can build the RCCE library with either the basic or the gory interface. The gory interface exposes the RCCE one-sided primitives. The gory interface is not a strict superset of the basic interface. It contains some routines with the same name as those in the basic interface but with a different set of parameters. These routines are called auxiliary routines, and they are made up from some of the elementary gory routines.

Table 7 lists the RCCE calls belonging to the basic and gory interfaces. This table does not contain the entire set of RCCE calls. RCCE also contains some power management routines, which are shown in Table 8 RCCE Power Management Routines

The remainder of this section describes the RCCE routines in more detail.

Basic	Gory
Core	**Core**
RCCE_init()	RCCE_int()
RCCE_finalize()	RCCE_finalize()
RCCE_ue()	RCCE_ue()
RCCE_debug_set()	RCCE_debug_set()
RCCE_debug_unset()	RCCE_debug_unset()
RCCE_error_string()	RCCE_error_string()
RCCE_wtime()	
RCCE_comm_rank()	
RCCE_comm_size()	
Communication	**Communication**
RCCE_send()	Auxiliary RCCE_send()
RCCE_recv()	Auxiliary RCCE_recv()
RCCE_recv_test()	Auxiliary RCCE_recv_test()
RCCE_reduce()	RCCE_reduce()
RCCE_allreduce()	RCCE_allreduce()
RCCE_bcast()	RCCE_bcast()
RCCE_comm_rank()	RCCE_comm_rank()
RCCE_comm_size()	RCCE_comm_size()
RCCE_comm_split()	RCCE_comm_split()
	RCCE_put()
	RCCE_get()
	RCCE_flag_write()
	RCCE_flag_read()
Synchronization	**Synchronization**
RCCE_barrier()	RCCE_barrier()
RCCE_fence()	RCCE_fence()
	RCCE_wait_until()
Memory Management	**Memory Management**
	RCCE_malloc()
	Auxiliary RCCE_malloc_request()
	RCCE_free()
	RCCE_flag_alloc()
	RCCE_flag_free()

Table 7 RCCE Calls Belonging to the Basic and Gory Interfaces

RCCE Power Management
RCCE_power_domain()
RCCE_power_domain_master()
RCCE_power_domain_size()
RCCE_iset_power()
RCCE_wait_power()
RCCE_set_frequency()

Table 8 RCCE Power Management Routines

8.4 The STENCIL Example

The next two sections illustrate the use of the RCCE basic and gory interfaces. The RCCE release comes with a number of examples in the **`apps`** directory. The STENCIL example is in **`apps/STENCIL`**.

<TBD>
Provide an overview of what the STENCIL example actually does … what problem is it intended to solve?
</TBD>

RCCE provides two versions of the STENCIL example. One (**`RCCE_stencil_synch.c`**) uses the basic interface, and the other (**`RCCE_stencil.c`**) uses the gory interface.

In STENCIL, a matrix is partitioned among the cores. The first and last rows of the array are fixed. The first row is fixed at 1.0, and the last row is fixed at 2.0. That's how STENCIL prints out the matrix, but internally, a core refers to other cores at its right and left. So you can also think of the matrix as having its first column fixed at 1.0, and its last column fixed at 2.0.

`RCCE_num_ues()` returns the number of cores (n) participating in the calculation. A core's ID (also called its sequence number or its rank) goes from 0 to **`RCCE_num_ues()`** -1 (n-1).

The matrix never exists completely on one core. In STENCIL, the matrix **`a`** is declared as a one-dimensional array of dimension **`NX*NY`**. There are **`NX`** columns on each core. For each core, the row starts at **`a[offset]`**. For core 0, **`offset`** is 0; for core n-1, **`offset`** is **`NX*(NY-1)`**.

Consider the example when **`NX`**=8, **`NY`**=10, and the number of cores is 4. This array is distributed among the four cores as follows.

```
Core 0     9 rows (elements of top row are fixed 1.0)
Core 1     8 rows
Core 2     8 rows
Core 3     9 rows (elements of bottom row are fixed 2.0)
```

All other elements are zero. As the calculation proceeds and the stencil is applied, the 1.0 and 2.0 flow toward the center and approach stabilization.

8.5 RCCE Basic

To build the basic version of STENCIL, enter the directory **apps/STENCIL** and type

```
make stencil_synch
```

To run the resulting executable on four cores and perform 50 iterations, enter

```
rccerun -nue 4 -f ../../hosts/rc.hosts stencil_synch 50
```

The default number of iterations is 10, but you can override this default from the command line as shown above.

The command **rccerun** also has an option that specifies the clock. The default is 1GHz. If the cores were running at 748 MHz, you would add **-clock 0.748** to the **rccerun** invocation. The stencil programs print out some timing information and use the value provided with **-clock**.

<TBD>
Provide a description of use of RCCE_send() and RCCE_recv() and describe how/why they are synchronous.
</TBD>

8.6 RCCE Gory

To build the gory version of STENCIL, enter the directory **apps/STENCIL** and type

```
make API=gory stencil
```

To run the resulting executable on four cores and perform 50 iterations, enter

```
rccerun -nue 4 -f ../../hosts/rc.hosts stencil 50
```

<TBD>
Provide a description of use of RCCE_put(), RCCE_get(), RCCE_flag_write(), RCCE_wait_until(), RCCE_flag_alloc().
</TBD>

8.7 Power Management

Although you can manage SCC power by writing the VRC configuration register directly through memory-mapped I/O, it's highly recommended that you use the RCCE power management calls. When you use these RCCE calls, you don't run the risk of crashing the SCC platform. RCCE provides you with some very significant power management capabilities, but it does not provide you with access to everything that the VRC can do. RCCE is an evolving library, however, and future releases may provide more capability.

To use RCCE's power management functions compile the library with the flag **-PWRMGMT=1** to include the power management routines and **-PWRMGMT=0** to exclude them. RCCE assumes that the 2x2 voltage domains and the frequency domains are identical. RCCE does not recognize the "all-mesh" power domain. As far as RCCE is concerned, there are six 2x2 power domains, labeled 0 to 5.

RCCE assigns a single core in a power domain as a power domain master. This designation is not user configurable. Only the power domain master can communicate with the SCC power management facility. Power management calls issued by other cores are ignored and return immediately with the return value **RCCE_SUCCESS.**

For maximum impact on the power budget, RCCE modifies the frequency in concert with the voltage, through the combined power command, **RCCE_iset_power()**. The input to the function specifies the desired tile frequency divider (an integer ranging from 2 to 16).

The frequency is defined relative to a global reference clock that is set when the SCC platform first starts up. This reference clock can vary, but in almost all cases it is 1.6Ghz. Changing it to a different value is an advanced procedure that is not recommended.

RCCE_iset_power() sets the tile frequency to the reference clock divided by the supplied divisor. For example, if the divider is 4, the tile frequency is then 1.6GHz/4 = 400MHz. **RCCE_iset_power()** then determines the lowest voltage level that is consistent with the input value of the frequency divider and initiates the voltage change. **RCCE_iset_power()** provides output values that define the actual settings for the frequency divider and the voltage level.

The returned voltage level is an integer from 0 to 7. The actual voltage (in volts) is 0.6 + voltage_level * 0.1. This means that the minimum voltage is 0.6v and the maximum voltage is 1.3v. These are nominal voltages.

Because changing the voltage has such a high latency, **RCCE_iset_power()** is not a blocking call. When you issue **RCCE_istep_power()**, you get a request ID. The call does not block, and your program continues. Later you can issue the call **RCCE_wait_power()** and specify a request ID. You then block until the preceding request from **RCCE_iset_power()** is satisfied. When it is satisfied, the request ID is cleared.

RCCE_set_frequency_divider() sets the frequency of the cores in the domain of the calling core without changing the voltage. As with **RCCE_iset_power()**, you supply a frequency divider. **RCCE_set_frequency_divider()** will not let you set the frequency to a value that is too high for the current voltage.

Because changing the frequency has a low latency, **RCCE_set_frequency()** is a blocking call. As with **RCCE_iset_power()**, the call only has an effect when executed by the power domain master. Non-master cores do nothing and just return with **RCCE_SUCCESS**.

8.7.1 Power Domains

RCCE provides three informational calls that deal with power domains.

- **RCCE_power_domain()** returns the number of the power domain that contains the calling core. The power domain number does not change; it is hard linked to the position of the core in the mesh. As shown in Figure 9, there are six power domains, and each contains four tiles in a 2x2 array. The power domain in the lower left is 0. The domain immediately to its right is 1 and then 2. The upper row contains domains 3, 4, and 5, with 5 being the domain in the upper right.
- **RCCE_power_domain_master()** returns the rank of the domain master in the local power domain. RCCE refers to a core's processorID as its sequence number or its

rank. Table 2 shows how the processorIDs are arranged in the mesh. These processorIDs start with 0 and end with 47.

- **`RCCE_power_domain_size()`** returns the number of cores in the local power domain that are participating in the computation. Note, however, that when you change the power for cores in a domain, you change the power for all the cores, even those that are not participating in the computation.

8.7.2 Changing the Power

`RCCE_iset_power()` takes four parameters. The first is an input parameter called **`Fdiv`** that specifies the desired frequency divider. The second is an output parameter that will contain a request ID. The third and fourth parameters are also output parameters that tell you what the frequency divider and voltage level got set to. Most likely this output frequency divider is equal to your input frequency divider.

Table 9 shows the maximum frequency allowed for a particular voltage level. For example, if the voltage is 1.1v, the voltage level is 5, and a frequency above 875MHz may damage the chip. If the voltage is 0.9v, the voltage level is 3, and a frequency above 644MHz may damage the chip.

Voltage Level	Voltage (volts)	Maximum Frequency (MHz)
0	0.6	327
1	0.7	460
2	0.8	598
3	0.9	644
4	1.0	748
5	1.1	875
6	1.2	1024
7	1.3	1198

Table 9 Voltage and Frequency Values

When you start up the SCC platform with the sccGui, you can choose to have the tiles run at either 800MHz or 533MHz. These two values correspond to frequency dividers of 2 and 3 respectively. Table 10 lists the tile frequencies corresponding to all the possible frequency dividers.

With **`RCCE_iset_power()`** you specify a desired frequency divider. The call creates a temporary variable equal to the reference clock divided by that frequency divider. The call then traverses Table 9 starting with voltage level 0 and finds the first voltage level whose maximum frequency is greater that the value of this temporary variable. The call then sets the voltage to that value and sets its output values to that voltage level and frequency divider.

For example, if you choose a frequency divider of 3, the temporary variable equals 533MHz. The call sets the voltage level to 2. If you choose a frequency divider of 2, the temporary variable equals 800MHz. The call then sets the voltage level to 5.

Tile Frequency (MHz)	RCCE Frequency Divider	RCCE Voltage Level
800	2	5
533	3	2
400	4	1
320	5	0
266	6	0
228	7	0
200	8	0
178	9	0
160	10	0
145	11	0
133	12	0
123	13	0
114	14	0
106	15	0
100	16	0

Table 10 Tile Frequencies and RCCE Frequency Dividers

Here is another example. Assume that you issue

```
RCCE_iset_power(4, &request, &newVoltageDiv, &newFreqDiv)
```

Then, the temporary variable equals 400MHz, **`newVoltageDiv`** is 1, and **`newFreqDiv`** is 4. The actual voltage is 0.6v + 1*0.1v = 0.7v. The actual frequency is 1.6GHz/4 = 400MHz.

When you go to a power level that has a larger frequency, RCCE increases the voltage first and then the frequency. When you go to a power level that has a lower frequency, RCCE decreases the frequency first and the voltage.

Only one **`RCCE_istep_power()`** can be "in flight" at the same time for a power domain. If you issue another **`RCCE_istep_power()`** before the first one is satisfied, the second request is denied, and it returns an error code.

8.7.3 Changing the Frequency

Use **`RCCE_set_frequency_divider()`** to change the frequency. Because changing the frequency has low latency, the call is blocking. The call takes two parameters. The first is an input parameter that is the requested frequency divider; the second is an output parameter that is the frequency divider that actually results. Table 10 lists the available frequency dividers and the tile frequencies they correspond to.

A frequency divider less than 2 returns an error. A frequency divider greater than 16 gets set to 16.

9 Porting an Application

<TBD>

9.1 Emulators

<TBD>
Is this part of porting? There are really two emulators. Both run on the MCPC. The qemu emulator and the RCCE emulator.

The other emulator is a message-passing emulator for RCCE that runs on top of OpenMP on the MCPC. Describe using this. There is at least one caution about global data and threadprivate when moving from the message-passing emulator to SCC hw.
</TBD>

9.2 Libraries: when to recompile, when not to

<TBD>
Do users need to recompile libraries? The answer here is most likely “sometimes.” So how do you determine when you need to recompile and when you don’t. If you recompile, we suggest icc . Are there specific compiler configurations we need to caution users about … like what optimization switches to use?

Would you ever statically link? If so, when?
</TBD>

10 Advanced Uses

10.1 Building your own Linux Image

Intel Labs provides a Linux image of SCC Linux that you can load on the cores. This default Linux image is called **linux.obj** and is stored in **/opt/sccKit/current/resources** on the management console (MCPC). It is a hex-encoded ASCII file describing the operating environment for SCC.

You have the ability to create your own Linux image. This section describes how to build the default Linux image. If you want to build a custom Linux image, model your procedure after these steps. One possibility is that you want essentially the default Linux image, but you want to add other utilities or apply a kernel patch.

10.1.1 Obtain rckos and linuxkernel

Obtain the needed sources. The sources are located in /**shared/SCC_Linux_src** on the MCPC. The **SCC_Linux_src** directory contains source in the subdirectories **linuxkernel** and **rckos**. You can copy these sources to a different location maintaining their directory structure, or you can choose to build in place.

10.1.2 Put the Cross Compiler and /sbin in Your Path

To do the build, you need the supplied cross-compiler. You need a cross-compiler because the cores are P54C, unlike the MCPC. The cross-compiler is in the directory **/shared/SCC_Linux_src/i386-unknown-linux-gnu**.

Put the cross-compiler's **bin** in your path. Also, put **/sbin** in your path.

```
export PATH=/sbin:/shared/SCC_Linux_src/i386-unknown-linux-gnu/bin:$PATH
```

10.1.3 Execute the Build Scripts

Enter the directory **rckos** and execute the commands, **mk_clean** and **mk_all**.

```
cd rckos
bin/mk_clean
bin/mk_all
```

The script **mk_all** has two options: **SCC** and **EMU**. SCC is the default. **mk_all SCC** creates the file **rcklinux.obj in rckos**. The file **rcklinux.obj** is an ASCII formatted image file containing the newly built Linux kernel along with some utility programs; it is ready to be booted on the SCC platform.

mk_all EMU creates the file **rcklinux.iso** in **rckos**. If you have **qemu** installed, you can issue the command **bin/run_rck**. This brings up **rcklinux.iso** in the **qemu** emulator.

Note that although the **qemu** emulator only emulates one core, it is still useful. What you can do is build your application outside of the emulator (the emulator has no build environment). Then, you take your application and load it into the emulator and see if it runs. Because the emulator is only emulating one core, you, of course, cannot do any message passing in this application, but you can check that the core has all the appropriate libraries and that they are

the same as those in your build environment.

10.1.4 microcore_2_3_1.iso

Both **`rcklinux.obj`** and **`rcklinux.iso`** begin with **`microcore_2_3_1.iso`**. MicroCore is a no-X-environment **`iso`** based on TinyCore. See http://tinycorelinux.com/ for details

Depending on the option you give **`mk_all`**, you either build **`rcklinux.obj`** (the image to be loaded on the SCC cores) or **`rcklinux.iso`** (the iso that you can bring up in the **`qemu`** emulator).

10.2 Using the sccAPI

<TBD>

</TBD>

10.3 Example of Modifying the Config Registers (LUTs?) for a Specific Need

<TBD>

</TBD>

11 Appendix

11.1 SCChello.c

```
#include <stdio.h>
main() {
   printf("hello\n");
}
```

11.2 readTileID.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>

#define CRB_OWN   0xf8000000
#define MYTILEID  0x100

main() {

typedef volatile unsigned char* t_vcharp;

int result, PAGE_SIZE, NCMDeviceFD;
// NCMDeviceFD is the file descriptor for non-cacheable memory (e.g. config regs).

unsigned int result, tileID, coreID, x_val, y_val;
   coreID_mask=0x00000007, x_mask=0x00000078, y_mask=0x00000780;;

t_vcharp     MappedAddr;
unsigned int alignedAddr, pageOffset, ConfigAddr;

   ConfigAddr = CRB_OWN+MYTILEID;
   PAGE_SIZE  = getpagesize();

   if ((NCMDeviceFD=open("/dev/rckncm", O_RDWR|O_SYNC))<0) {
    perror("open"); exit(-1);
   }

    alignedAddr = ConfigAddr & (~(PAGE_SIZE-1));
    pageOffset  = ConfigAddr - alignedAddr;

    MappedAddr = (t_vcharp) mmap(NULL, PAGE_SIZE, PROT_WRITE|PROT_READ,
       MAP_SHARED, NCMDeviceFD, alignedAddr);

   if (MappedAddr == MAP_FAILED) {
      perror("mmap");exit(-1);
   }

  result = *(unsigned int*)(MappedAddr+pageOffset);
  munmap((void*)MappedAddr, PAGE_SIZE);

  printf("result = %x %d \n",result, result);

   coreID = result & coreID_mask;
   x_val  = (result & x_mask) >> 3;
   y_val  = (result & y_mask) >> 7;
   tileID = y_val*16 + x_val;

   printf("My (x,y) = (%d,%d)\n", x_val, y_val);
```

```
    printf("My tileID = 0x%2d\n",tileID);
    printf("My coreID = %1d\n",coreID);
    printf("My processorID = %2d\n",tileID*2 + coreID);

}
```