RCCE: a Small Library for Many-Core Communication

Software Version 1.0-release, May 3, 2010

Document Version 0.7

Tim Mattson (IL) and Rob van der Wijngaart (SSG)

Abstract: SCC is a many-core research chip developed by Intel Labs. It contains a mesh of tiles, two processor cores per tile, off-chip private memory per core, shared off-chip memory, and a shared on-chip message passing buffer. The cores support a general x86 instruction set (P54C), hence we have access to compilers and a stable execution environment for the cores used on the chip. That means we can support full scale application programming on SCC. The programming environment for SCC described in this report is named RCCE: a Small Library for Many-Core Communication. This is a simple message passing environment built on top of a basic one-sided communication system.

In this document, we define the RCCE API and provide notes and assumptions used to support its implementation on the SCC chip. We also describe our functional emulator built on top of OpenMP. This emulator lets us develop and test code for SCC without requiring access to actual hardware.

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.

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 the Power Management section to describe new version of the Power Management API. Added Appendix B to describe the interaction of RCCE software with SCC hardware,

Table of Contents

1 Introduction 7
2 Overview of the SCC Architecture and RCCE 7
2.1 Memory 8
2.1.1 Memory Organization 8
2.1.2 Memory Size 8
2.1.3 Cache Behavior 8
2.1.4 Working with MPB Memory 9
2.2 Programming Model 10
3 Basic RCCE API 11
3.1 Core Utilities 11
3.2 Communication 11
3.3 Synchronization 11
3.4 Core Utilities: Description 12
3.4.1 Return values, Error Codes, and Types 12
3.4.2 int RCCE_init(int *argc, char ***argv) 12
3.4.3 int RCCE_finalize(void) 12
3.4.4 int RCCE_num_ues(void) 12
3.4.5 int RCCE_ue(void) 12
3.4.6 int RCCE_debug_set(int dbg_enable) 13
3.4.7 int RCCE_debug_unset(int dbg_disable) 13
3.4.8 int RCCE_error_string(int err_no, char *error_string, int *string_length) 13
3.4.9 double RCCE_wtime(void) 14
3.4.10 int RCCE_comm_size(RCCE_COMM comm, int * size) 14
3.4.11 int RCCE_comm_rank(RCCE_COMM comm, int *rank) 14
3.4.12 int RCCE_comm_split(int (*color)(int, void *), void *aux, RCCE_COMM *comm) 15
3.5 Communication: Description 16
3.5.1 int RCCE_send(char *privbuf, size_t size, int ID) 16
3.5.2 int RCCE_recv(char *privbuf, size_t size, int ID) 16
3.5.3 int RCCE_recv_test(char *privbuf, size_t size, int ID, int *test) 16
3.5.4 int RCCE_reduce(char *inbuf, char *outbuf, int number, int datatype, int operation, int root, RCCE_COMM comm) 17
3.5.5 int RCCE_allreduce(char *inbuf, char *outbuf, int number, int datatype, int operation, RCCE_COMM comm) 17

3.5.6 int RCCE_bcast(char *buf, size_t number, int root, RCCE_COMM comm) 18

3.6 Synchronization: Description 18

3.6.1 int RCCE_barrier(RCCE_COMM *comm) 18

3.6.2 Int RCCE_fence(void) 18

4 Gory RCCE API 19

4.1 Core Utilities 19

4.2 Memory Management 19

4.3 Communication 19

4.4 Synchronization 20

4.5 Auxiliary Functions (based on elementary functions defined above) 20

4.6 Power management 20

4.7 Memory management: Description (MPB address space) 21

4.7.1 volatile char * RCCE_malloc (size_t size) 21

4.7.2 void RCCE_free (volatile char *ptr) 21

4.7.3 int RCCE_ flag_alloc (RCCE_FLAG *flag) 21

4.7.4 int RCCE_ flag_free (RCCE_FLAG *flag) 21

4.8 Communication: Description 21

4.8.1 int RCCE_put(volatile char * target, volatile char *src, int num_bytes, int ID) 22

4.8.2 int RCCE_get(volatile char * target, volatile char *src, int num_bytes, int ID) 22

4.8.3 int RCCE_flag_write(RCCE_FLAG *flag, RCCE_FLAG_STATUS status, int ID) .. 22

4.8.4 int RCCE_flag_read(RCCE_FLAG *flag, RCCE_FLAG_STATUS *status, int ID).. 22

4.9 Synchronization: Description 23

4.9.1 int RCCE_wait_until (RCCE_FLAG flag, RCCE_FLAG_STATUS status): 23

4.10 Auxiliary Functions: Description 23

4.10.1 volatile char *RCCE_malloc_request(size_t request, size_t *result) 23

4.10.2 int RCCE_send(char *privbuf, volatile char *combuf, size_t combuf_size, RCCE_FLAG *ready, RCCE_FLAG *sent, size_t size, int ID) 23

4.10.3 int RCCE_recv(char *privbuf, volatile char *combuf, size_t combuf_size, RCCE_FLAG *ready, RCCE_FLAG *sent, size_t size, int ID) 24

4.10.4 int RCCE_recv_test(char *privbuf, volatile char *combuf, size_t chunk, RCCE_FLAG *ready, RCCE_FLAG *sent, size_t size, int ID, int *test) 24

5 Power Management API 25

5.1 Power Management 25

5.2 Power Management: Description 25

5.2.1 int RCCE_power_domain(void) 27

5.2.2 int RCCE_power_domain_master(void) ... 27
5.2.3 int RCCE_power_domain_size(void) ... 27
5.2.4 int RCCE_iset_power(int Fdiv, RCCE_REQUEST *request, int * Fdiv_new, int * Vlevel_new) ... 27
5.2.5 int RCCE_wait_power(RCCE_REQUEST *request) ... 28
5.2.6 int RCCE_set_frequency_divider(int Fdiv, int Fdiv_new) ... 28
6 The RCCE Emulator ... 29
6.1 Key Elements ... 29
6.2 Caution about Global Data ... 29
7 Appendix A: Glossary ... 30
8 Appendix B: RCCE and SCC Hardware ... 31
8.1 Tile ID ... 31
8.2 Power Domains ... 31
8.3 Changing the Voltage ... 32
8.4 Changing the Frequency ... 32

List of Tables

Table 1 Tile Frequency Settings for Router Clock of 800MHz ... 33
Table 2 Tile Frequency Settings for Router Clock of 1.6GHz ... 33

List of Figures

Figure 1 SCC processor logical layout showing core IDs, (x,y) tile coordinates, and power domains (0 through 5) ... 26
Figure 2 SCC and RCCE IDs ... 32

1 Introduction

Intel is building a series of research chips to study many-core CPUs, their architecture, and the techniques used to program them. The first of these chips was the now famous “80-core research chip”. The second chip is called the single-chip cloud computer (SCC).

The 80-core chip and SCC have much in common. Both are research chips and hence are not included on any product roadmaps. They both use a mesh for the on-die network. In both cases, the cores do not interact through a cache-coherent shared address space; so the native programming models depend on message passing or some other scheme that makes cache coherence explicit.

The two chips differ, however, in that the cores used in SCC are general purpose x86 processors. The 80-core chip used a limited, non-IA instruction set, no compiler, and no OS. SCC, on the other hand, has a full IA core (P54C), an operating system (for example, Linux), and multiple compilers. Consequently, while the 80 core chip supported only simple application kernels, SCC supports full application programming.

RCCE (pronounced “rocky”) is the message passing 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 emulator was critical before SCC hardware was available and is still useful for software development.

This document provides an overview of the SCC architecture and RCCE. It begins by discussing the basic interface that most programmers will use and continues with a discussion of the gory interface used by expert programmers interested in detailed control over the chip. It then discusses the power management API built into RCCE and closes with a description of the RCCE emulator. Finally, it includes a glossary as Appendix A: Glossary.

2 Overview of the SCC Architecture and RCCE

SCC is a (mostly) distributed memory, tiled, many-core processor. Each tile has two cores, a single router shared by the cores, and a region of shared memory used as a communication buffer. The router connects to an on-die mesh network. The cores are second generation Pentium® cores (P54C) and as expected with the P54C architecture, they include level 1 (L1) instruction and data caches (16KB each) and a unified level 2 (L2) cache (256KB).

2.1 Memory

2.1.1 Memory Organization

SCC memory consists of off-chip DRAM and on-chip SRAM. RCCE programs use both. When you write RCCE applications, your program directly accesses the off-chip DRAM while the functions inside RCCE internally use the on-chip SRAM (called the message-passing buffer or MPB). As a programmer, you can also access this on-chip SRAM, but there are specific rules that you must follow. These rules arise because of the specifics of how data in the message passing buffer are cached. See Section 2.1.3 Cache Behavior.

The off-chip DRAM consists of memory that is private to each core and memory that is shared by all the cores. Where this division occurs is configurable. Each core has an associated lookup table (one per core, or LUT0 and LUT1 on each tile). These LUTs are configured with default values at boot time, but you can modify their settings. Their default is to give each core as much private memory as possible. The memory that's left over is shared by all cores and is currently not yet used by RCCE. As a programmer, you still have access to it, but you must manage the coherence between cores yourself.

The SCC chip has four on-chip memory controllers that support DDR3 memory off chip. 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.

2.1.2 Memory Size

Each of the four memory controllers can support from 4GB to 16GB of DRAM, resulting in a total off-chip DRAM of 16GB to 64GB, which is addressable by the SCC *system address*. A core addresses up to 4GB with a 32-bit address called the *core address*. A core's LUT translates the core address into the system address.

The MPB is shared memory and in principle directly addressable by any core in the SCC chip. Each tile has 16KB of SRAM allocated to the MPB. Hence, the MPB provides 384KB (24 * 16KB) of on-die SRAM memory. RCCE, however, does not use the MPB as a flat address space. Instead, RCCE logically partitions the MPB into 8 KB message buffers assigned to each core.

2.1.3 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 cores, the SCC system avoids snooping, snarfing, or any other type of inter-core cache protocol overhead.

The relationship between a core's shared memory (and this includes both the shared off-chip DRAM and the MPB) and is described in the following subsection.

2.1.3.1 Cacheable Shared Memory

Shared memory (off-chip DRAM or MPB) data can be assessed through a core's cache, but not in the way commonly associated with a typical x86 processor.

Data from shared memory are cached in L1 but may bypass L2. The programmer can declare pages in the shared memory space as write-back or write-through, and these data will normally be

cached in L2. However, if the data are typed as Message Passing Buffer Type (MPBT), the data will not be cached in L2. MPBT data bypass L2 and go directly to L1.

As mentioned previously, the SCC does not provide an automatic mechanism to maintain cache coherence among cores. You must manage coherence of cache data between cores explicitly. The SCC provides two tools for this purpose.

One is a special tag for cache lines that marks the data as MPBT or Message Passing Buffer Type. MPBT data are moved between the core's L1 cache and shared memory with the granularity of 32-byte cache lines. When this move occurs is internal to the operation of RCCE, and most users need not be concerned with the details. Essentially RCCE uses a new SCC instruction called CL1INVMB. This instruction marks all MPBT-typed data as invalid L1 lines so that a later access of the data forces an update of L1. The RCCE library handles these features of MPBT internally.

2.1.3.2 Non-Cacheable Shared Memory and RCCE

In addition to the MPBT data mapped onto the L1 core caches, you can configure shared memory in the SCC system that is not of type MPBT. In this case, data move between the registers of a core and shared memory (that is, it bypasses the caches entirely) with a granularity of 1, 2, 4 or 8 bytes. For these memory operations, due to the restrictions of the P54C architecture, only one read or write may be active at one time to an address.

This feature is under development within RCCE . We are exploring use of this memory within RCCE. Non-cacheable shared memory would be mapped onto off-chip DRAM and exposed through a special `malloc()` called `shmalloc()`.

2.1.4 Working with MPB Memory

There are various approaches for working with MPB memory. RCCE adopts the *shared name space* or symmetric memory model. In this model, all the variables of a given name are assigned together across all nodes. This model lets a programmer reference variables stored within the MPB name and the core ID.

An implication of the shared name-space model is that certain RCCE routines must be encountered jointly by all UEs. RCCE calls these *collective* operations. For example, memory management routines such as `RCCE_malloc()` are collective operations in the shared name space model.

"Encountered jointly" does not necessarily mean at the same time. It means that when a RCCE thread or process (called a UE for unit of execution) calls a collective routine, it calls it in the same order with respect to other collective RCCE routines.

- Each UE is assigned a distinct range of contiguous addresses in the MPB address space.
- Memory in the MPB is allocated by *collective* calls to `RCCE_malloc()`. This defines a single MPB namespace shared among all the UEs involved in the computation.
- Names from the MPB namespace use identical offsets from the beginning of the MPB address space associated with each core.

As an example, the address returned from `RCCE_malloc()` for the UE of rank ID is (offset + head(ID)) where head(ID) denotes the beginning of the MPB address space associated with the UE of rank ID; offset is the same for all UEs.

Its important to note, however, that if the RCCE programmer avoids the gory interface and uses

only the basic interface, the details of MPB memory operations remain hidden. Because the SCC is a research chip, we do discuss the details of how the MPB memory works. As a researcher, you are likely concerned with such low level details; but, when you are getting started with RCCE, we recommend that you restrict yourself to the basic API.

2.2 Programming Model

Communication between cores occurs by moving data between private memories and MPBs. At the lowest level this suggests a one-sided communication model. RCCE is a minimal programming environment. RCCE has functions that perform the following actions.

- Initialize and shut down the environment.
- Send and receive messages among the cores.
- Synchronize core programs with barriers and fences.
- Manage the power of the cores. The power management capability is optional. You can choose to include this capability when you build RCCE.
- Move data between private memory and the MPBs with simple put/get routines. This advanced interface is exposed with the gory interface. When using the gory interface, the programmer must ensure that the granularity of data movement is the width of an L1 cache line (32 bytes).
- Synchronize core programs using flags, which are implemented based on a known initial state of the MPBs. This advanced interface is exposed with the gory interface.

The SCC processor is capable of supporting a wide range of distributed memory execution models. Initially, we focus on the simplest model described as follows.

- A program executes as one or more Units of Execution, or UEs, mapped one to a core. A UE is an agent that "owns the program counter" and makes progress in a computation; that is, think of a UE as an abstraction that can be implemented as a thread or a process. Once assigned to a core, a UE remains pinned to that core.
- A static SPMD model, all UEs are created together when the program is started. They are assigned an ID which is their sequence number among the collection of UEs (that is, it ranges from 0 to the number of cores minus 1). Because a UE is pinned to a core, the ID uniquely defines a core and a UE.
- No ordering is implied as to when respective UEs begin execution. A correct RCCE program cannot rely on any assumptions about when any UE begins execution.
- Only one RCCE parallel program executes on the chip at a time, utilizing either all or a subset of the cores.

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 who MPB you want to clear.

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.

3 Basic RCCE API

In this section, we define the functions that comprise the basic SCC Communication Environment. The Basic RCCE API is a simplified interface that hides all details of the MPB and the synchronization flags used to manage the MPB. It is a restrictive model that only allows fully synchronized communications (matched send and receive calls).

Recall that RCCE uses a symmetric name space model. Such a model designates a number of functions as *collective*, meaning that they are called jointly by all UEs in the same program order. Except for `RCCE_barrier()`, the collective functions do not imply synchronization. The collective functions are listed below in bold font.

3.1 Core Utilities

`int` **RCCE_init(int *, char***)**
`int` **RCCE_finalize(void)**
`int` RCCE_num_ues(void)
`int` RCCE_ue(void)
`int` RCCE_debug_set(int)
`int` RCCE_debug_unset(int)
`int` RCCE_error_string(int, char *, int *)
`int` RCCE_wtime(void)
`int` RCCE_comm_rank(RCCE_COMM, int *)
`int` RCCE_comm_size(RCCE_COMM, int *)
`int` RCCE_comm_split(int (*)(int, void *), void *, RCCE_COMM *)

3.2 Communication

`int` RCCE_send(char *, size_t, int)
`int` RCCE_recv(char *, size_t, int)
`int` RCCE_recv_test(char *, size_t, int, int *)
`int` **RCCE_reduce(char *, char *, int, int, int, int, RCCE_COMM)**
`int` **RCCE_allreduce(char *, char *, int, int, int, RCCE_COMM)**
`int` **RCCE_bcast(char *, int, int, RCCE_COMM)**
`int` **RCCE_comm_split(int (*color)(int, void*), void *aux, RCCE_COMM *comm)**

3.3 Synchronization

`void` **RCCE_barrier(RCCE_COMM *)**
`void` RCCE_fence(void)

3.4 Core Utilities: Description

3.4.1 Return values, Error Codes, and Types

Return values, error codes, and types are defined in `RCCE.h` and are discussed in the sections relevant to their definition.

For `RCCE_malloc()` a return value of `NULL` (numerical value zero) indicates an error condition. For all other RCCE calls that return an integer, a return value of `RCCE_SUCCESS` implies that no error occurred. Usually, though not necessarily, the value of `RCCE_SUCCESS` is zero.

Note that `RCCE_ue()` and `RCCE_num_ues()` do not take any arguments and cannot fail, provided they occur after `RCCE_init()`.

3.4.2 int RCCE_init(int *argc, char ***argv)

`RCCE_init()` is the RCCE initialization function. It is analogous to `MPI_init()` and provides a place for any code required to set up the environment for RCCE programs. `RCCE_init()` is a collective routine that must be encountered jointly by all UEs. It must be the first RCCE statement in the program. It must also come before any statements that read or write the `argc` and/or `argv` variables.

`argc`	Pointer to the number of application arguments on the command line.
`argv`	Pointer to a pointer to an array of strings (application command line arguments).

3.4.3 int RCCE_finalize(void)

`RCCE_finalize()` is analogous to the `MPI_finalize()`routine and provides a place for any code needed to cleanly shut down the RCCE environment. No RCCE statements may follow `RCCE_finalize()`.

3.4.4 int RCCE_num_ues(void)

`RCCE_num_ues()` returns the number of units of execution (`n`) participating in the computation.

3.4.5 int RCCE_ue(void)

`RCCE_ue()` returns the sequence number (`rank`) of a calling unit of execution (0 to (`n`-1)).

3.4.6 int RCCE_debug_set(int dbg_enable)

`RCCE_debug_set()` enables runtime debug messages for RCCE library calls. Depending on the value of the input parameter `dbg_enable`, error messages concerning synchronization, communication, power management, or all of the above will be printed. The default is to ignore all error messages.

`dbg_enable` Enables runtime debug messages for RCCE library calls. The parameter `dbg_enable` can take on the following values.

`RCCE_DEBUG_SYNCH`	Print error messages concerning synchronization.
`RCCE_DEBUG_COMM`	Print error messages concerning communication.
`RCCE_DEBUG_RPC`	Print error messages concerning power management.
`RCCE_DEBUG_ALL`	Print all error messages.

3.4.7 int RCCE_debug_unset(int dbg_disable)

`RCCE_debug_unset()` disables runtime debug messages for RCCE library calls. The default is to ignore all error messages.

`dbg_disable` Disables runtime debug messages for RCCE library calls. The parameter `dbg_disable` can take on the following values.

`RCCE_DEBUG_SYNCH`	Ignore error messages concerning synchronization.
`RCCE_DEBUG_COMM`	Ignore error messages concerning communication.
`RCCE_DEBUG_RPC`	Ignore error messages concerning power management.
`RCCE_DEBUG_ALL`	Ignore all error messages. This is the default.

3.4.8 int RCCE_error_string(int err_no, char *error_string, int *string_length)

For a given numerical error code, `RCCE_error_string()` returns a descriptive string and the length of that string.

`err_no`	Given numerical error code.
`error_string`	Descriptive error string.
`string_length`	Length of the descriptive string.

3.4.9 double RCCE_wtime(void)

`RCCE_wtime()` returns the number of seconds since some specific time in the past. By calling this function upon entry and exit of a block of code, you can find the elapsed wall clock time within that block. This is closely modeled after the wall clock timers in MPI and OpenMP. There is no guarantee that return values of `RCCE_wtime()` for different UEs are related. When power management routines are called, the accuracy of the timer is not guaranteed.

3.4.10 int RCCE_comm_size(RCCE_COMM comm, int * size)

`RCCE_comm_size()` stores the number of UEs within communicator `comm` in the integer `size`.

`comm`	The communicator whose size (number of UEs) the call returns.
`size`	The number of UEs in the communicator `comm`.

3.4.11 int RCCE_comm_rank(RCCE_COMM comm, int *rank)

`RCCE_comm_rank()` stores the sequence number of the calling UE within communicator `comm` in the integer `rank`.

`comm`	The communicator containing the UE whose sequence number the call stores in the integer rank.
`rank`	The sequence number of the calling UE.

3.4.12 int RCCE_comm_split(int (*color)(int, void *), void *aux, RCCE_COMM *comm)

`RCCE_comm_split()` creates a new communicator, `comm`, based on the global communicator, `RCCE_COMM_WORLD`, the function `color()`, and any custom data the programmer wants to furnish.

`color()` is called as `color(rank, aux)` and returns an integer. All UEs must call `RCCE_comm_split()` with identical arguments.

All UEs with the same return value for `color()` will be placed in the same communication group. The UEs inside that group are ranked in the order of their rank within the global communicator.

Note the differences between RCCE and `MPI_Comm_split()`. `RCCE_comm_split()` has

- No input communicator; the global communicator is assumed.
- No key value, the global rank is assumed.
- No color value, but a uniform function to compute the color, based on global rank and (possibly) other attributes.

The current version of RCCE does not provide the ability to delete RCCE communicators. However, if a communicator is no longer needed, it can safely be overwritten by a new invocation of `RCCE_comm_split()`.

`comm`	The new communicator, based on the global communicator, `RCCE_COMM_WORLD`.
`color`	Function pointer to `color()`. All UEs with the same return value for `color()` will be placed in the same communication group.
`aux`	Any custom data the programmer wants to furnish.

3.5 Communication: Description

Note that if `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.

3.5.1 int RCCE_send(char *privbuf, size_t size, int ID)

`RCCE_send()` sends an arbitrarily sized buffer (does not have to be a multiple of 32 bytes) of data from the private memory of the sending UE to the private memory of the receiving UE. Its parameters must be matched exactly by those on the receiving side. The call is blocking. Blocking means it will not complete until a matching receive has been posted.

`privbuf`	Starting address (private memory) of data to be sent.
`size`	Size of message in bytes.
`ID`	Rank of target UE.

3.5.2 int RCCE_recv(char *privbuf, size_t size, int ID)

`RCCE_recv()` receives an arbitrarily sized buffer (does not have to be a multiple of 32 bytes) of data from the private memory of the sending UE to the private memory of the receiving UE. Its parameters must be matched exactly by those on the sending side.

`privbuf`	Starting address (private memory) of data to be sent.
`size`	Size of message in bytes.
`ID`	Rank of target UE.

3.5.3 int RCCE_recv_test(char *privbuf, size_t size, int ID, int *test)

`RCCE_recv_test()` is the non-blocking version of `RCCE_recv()`. It returns immediately. It does not wait for the required message to actually arrive. If the message has arrived, it is stored in the proper location in private memory, and the variable `test` is set to one. If the message has not arrived, `test` is set to zero.

`privbuf`	Starting address (private memory) of data to be received.
`size`	Size of message in bytes.
`ID`	Rank of target UE.
`test`	Variable that indicates whether the receive was (1) or was not (0) successful.

3.5.4 int RCCE_reduce(char *inbuf, char *outbuf, int number, int datatype, int operation, int root, RCCE_COMM comm)

`RCCE_reduce()` applies an element-wise reduction to an array of `number` words across a collection of UEs and places the result in one UE.

`inbuf`	Starting address of input array (private memory).
`outbuf`	Starting address of result array (private memory).
`number`	Length of array (words) of values to be reduced.
`datatype`	Constant indicating the numerical data type. Choices are `RCCE_INT`, `RCCE_LONG`, `RCCE_FLOAT`, and `RCCE_DOUBLE`.
`operation`	Constant indicating the reduction operation. Choices are `RCCE_SUM`, `RCCE_MAX`, `RCCE_MIN`, and `RCCE_PROD`.
`root`	Rank of the UE within communicator `comm` that receives the result of the reduction operation.
`comm`	Communicator whose UEs all contribute to the reduction.

3.5.5 int RCCE_allreduce(char *inbuf, char *outbuf, int number, int datatype, int operation, RCCE_COMM comm)

`RCCE_allreduce()` applies an element-wise reduction to an array of `number` words across a collection of UEs and places the result in all UEs inside communicator `comm`.

`inbuf`	Starting address of input array (private memory).
`outbuf`	Starting address of result array (private memory).
`number`	Length of array (words) of values to be reduced.
`datatype`	Constant indicating the numerical data type. Choices are `RCCE_INT`, `RCCE_LONG`, `RCCE_FLOAT`, and `RCCE_DOUBLE`.
`operation`	Constant indicating the reduction operation. Choices are `RCCE_SUM`, `RCCE_MAX`, `RCCE_MIN`, and `RCCE_PROD`.
`root`	Rank of the UE within communicator `comm` that receives the result of the reduction operation.
`comm`	Communicator whose UEs all contribute to the reduction.

3.5.6 int RCCE_bcast(char *buf, size_t number, int root, RCCE_COMM comm)

This function copies `number` bytes at address `buf` on the UE with rank `root` to address `buf` on all other UEs in communicator `comm`.

`buf`	Starting address of input array (private memory).
`number`	Number of bytes to be sent.
`root`	Rank of the UE that is the source of the message.
`comm`	Communicator whose UEs receive the message.

3.6 Synchronization: Description

3.6.1 int RCCE_barrier(RCCE_COMM *comm)

`RCCE_barrier()` specifies a barrier at this point in the code. This is a collective routine that must be called jointly by all UEs in the communicator.

`comm`	Communicator whose UEs participate in the barrier. Predefined communicators are `RCCE_COMM_WORLD`, and, in case of fine-grain power management, `RCCE_P_COMM`, which comprises all the UEs that lie in the local power domain. Any other communicators to be used in a barrier must be constructed by using `RCCE_comm_split()`.

3.6.2 Int RCCE_fence(void)

`RCCE_fence()` defines a sequence point in the program at which the following conditions hold.

- A compiler can not reorder memory access operations across the `RCCE_fence()` function.
- All memory operations that occur in program-order prior to this statement must complete.
- No memory operations that follow this statement in program-order must begin before returning from this function. This behaves as an OpenMP flush construct without an argument list clause. Note that we need the fence to apply to both private and MPB memory.

4 Gory RCCE API

This section defines the functions that comprise the low level or *gory* version of RCCE. The gory interface exposes the details of the MPB, including memory allocation and management of flags. This interface forces the programmer to manage all details of the MPB. It is intended for expert programmers who want complete control over the MPB. Currently, you can not use both the basic and gory interfaces in a single program.

Not surprisingly, there is considerable overlap between gory and basic interfaces. This section describes only those functions whose API is different from that of the basic RCCE API. As before, nominally or actually collective functions are displayed in bold face.

The gory RCCE API provides maximum flexibility in implementing applications; for example the gory RCCE API allows unsynchronized message passing (no handshaking). Many of the library's low level functions and data structures are exposed with the gory API. Because this exposure is undesirable for most programmers, the gory API is not the default, but can be obtained by specifying the `API=gory` flag when building the library and the application. Without it, the basic API described in Basic RCCE API is used.

4.1 Core Utilities

```
int     RCCE_init(int *, char***)
int     RCCE_finalize(void)
int     RCCE_ue(void)
int     RCCE_num_ues(void);
int     RCCE_error_string(int, char *, int *)
int     RCCE_debug_set(int)
int     RCCE_debug_unset(int)
int     RCCE_wtime(void)
int     RCCE_comm_split(int (*)(int, void *), void *aux, RCCE_COMM *)
int     RCCE_comm_rank(RCCE_COMM, int *)
int     RCCE_comm_size(RCCE_COMM, int *)
int     RCCE_comm_split(int (*)(int, void *),void *, RCCE_COMM *)
```

4.2 Memory Management

volatile char *RCCE_malloc(size_t)
void **RCCE_free(volatile char *)**
int **RCCE_flag_alloc(RCCE_FLAG *)**
int **RCCE_flag_free(RCCE_FLAG *)**

4.3 Communication

```
int     RCCE_put(volatile char *, volatile char *, int, int)
int     RCCE_get(volatile char *, volatile char *, int, int)
int     RCCE_flag_write(RCCE_FLAG *, RCCE_FLAG_STATUS, int)
int     RCCE_flag_read(RCCE_FLAG, RCCE_FLAG_STATUS *, int)
```

4.4 Synchronization

`void` RCCE_barrier(RCCE_COMM *comm)
`void` RCCE_fence(void)
`int` RCCE_wait_until(RCCE_FLAG, RCCE_FLAG_STATUS))

4.5 Auxiliary Functions (based on elementary functions defined above)

`volatile char *`RCCE_malloc_request(size_t, size_t *)
`int` RCCE_send(char *, volatile char *, size_t, RCCE_FLAG *, RCCE_FLAG *, size_t, int)
`int` RCCE_recv(char *, volatile char *, size_t, RCCE_FLAG *, RCCE_FLAG *, size_t, int);
`int` RCCE_recv_test(char *, volatile char *, size_t, RCCE_FLAG *, RCCE_FLAG *, size_t, int, int *);

4.6 Power management

`int` RCCE_power_domain(void)
`int` RCCE_power_domain_master(void)
`int` RCCE_power_domain_size(void)
`int` RCCE_istep_power(int, RCCE_REQUEST *)
`int` RCCE_wait_power(RCCE_REQUEST *)
`int` RCCE_step_frequency(int)

4.7 Memory management: Description (MPB address space)

4.7.1 volatile char * RCCE_malloc (size_t size)

`RCCE_malloc()` allocates `size` bytes in MPB in the address range assigned to the calling UE. This is a collective routine that must be called jointly by all UEs. Size must be a multiple of 32, and return values will be aligned on 32-byte boundaries.

`size` The number of bytes to allocate in memory

4.7.2 void RCCE_free (volatile char *ptr)

`RCCE_free()` deallocates previously allocated space in the MPB at address `ptr`. This is a collective routine that must be called jointly by all UEs.

`ptr` Pointer to previously allocated space that will be freed.

4.7.3 int RCCE_ flag_alloc (RCCE_FLAG *flag)

`RCCE_flag_alloc()` allocates a flag variable on the calling UE. This is a collective routine that must be called jointly by all UEs.

`flag` Pointer to a flag in local MPB memory.

4.7.4 int RCCE_ flag_free (RCCE_FLAG *flag)

`RCCE_flag_free()` allocates a previously allocated flag variable in the MPB. This is a collective routine that must be encountered jointly by all UEs.

`flag` Pointer to a flag in local MPB memory.

4.8 Communication: Description

This section describes the basic one-sided RCCE communication routines.

Note that MPB virtual addresses need to be aligned on 32-byte boundaries (there is no such requirement for addresses in private memory). Also, the `num_bytes` in the calls below has to be an integral multiple of 32.

4.8.1 int RCCE_put(volatile char * target, volatile char *src, int num_bytes, int ID)

`RCCE_put()` copies the contents of the buffer pointed to by `src` into the location pointed to by `target`.

`target`	An MPB address that will be converted to the appropriate address on UE with rank ID that points to where the data will go.
`src`	The address in private memory or the local MPB of the calling UE that points to the data to be put.
`num_bytes`	Number of bytes to copy to the target.
`ID`	Rank of the target UE.

4.8.2 int RCCE_get(volatile char * target, volatile char *src, int num_bytes, int ID)

`RCCE_get()` copies the contents of the buffer pointed to by `src` into the location pointed to by `target`.

`target`	Address in calling UE's local MPB or the private memory of the calling UE that points to the data to be gotten.
`src`	An offset that, when combined with the ID, points to the source MPB.
`num_bytes`	Number of bytes to copy to the target.
`ID`	Rank of the target UE.

4.8.3 int RCCE_flag_write(RCCE_FLAG *flag, RCCE_FLAG_STATUS status, int ID)

`RCCE_flag_write()` sets the flag status of the UE with rank ID.

`flag`	Pointer to a flag in MPB memory.
`status`	An enumerated value (`RCCE_FLAG_SET` or `RCCE_FLAG_UNSET`) to be stored in the flag location of the target UE.
`ID`	Rank of the target UE.

4.8.4 int RCCE_flag_read(RCCE_FLAG *flag, RCCE_FLAG_STATUS *status, int ID)

`RCCE_flag_read()` reads the flag status of the UE with rank ID.

`flag`	Pointer to a flag in MPB memory.
`status`	An enumerated value (`RCCE_FLAG_SET` or `RCCE_FLAG_UNSET`) to be read in the flag location of the target UE.
`ID`	Rank of the target UE.

4.9 Synchronization: Description

In addition to the synchronization routines in the basic interface, the gory interface allows one to build custom synchronization protocols using the flag variables and `RCCE_wait_until()`.

4.9.1 int RCCE_wait_until (RCCE_FLAG flag, RCCE_FLAG_STATUS status):

`RCCE_wait_until()` blocks until the `status` of `flag` matches the input parameter.

`flag`	Opaque data type representing a flag in MPB memory.
`status`	Status value that must be matched by the status field of the synchronization flag. Possible values are `RCCE_FLAG_SET` and `RCCE_FLAG_UNSET`. All flags have status `RCCE_FLAG_UNSET` at program start.

4.10 Auxiliary Functions: Description

4.10.1 volatile char *RCCE_malloc_request(size_t request, size_t *result)

`RCCE_malloc_request()` returns a pointer to MPB space.

`request`	Requested size in bytes (multiple of 32 bytes).
`result`	Actually returned size in bytes (multiple of 32 bytes).

4.10.2 int RCCE_send(char *privbuf, volatile char *combuf, size_t combuf_size, RCCE_FLAG *ready, RCCE_FLAG *sent, size_t size, int ID)

`RCCE_send()` sends an arbitrarily sized buffer (does not have to be a multiple of 32 bytes) of data from the private memory of the sending UE to the private memory of the receiving UE. Its parameters must be matched exactly by those on the receiving side. The call is blocking. Blocking means it will not complete until a matching receive has been posted. It uses MPB memory of the sending core to store data in transit.

`privbuf`	Starting address (private memory) of data to be sent.
`combuf`	Starting address of intermediate MPB memory through which data are transferred.
`combuf_size`	Size in bytes of intermediate space in MPB used for this communication.
`ready`	Flag variable that indicates whether the receiver is ready to receive message.
`sent`	Flag variable that indicates that the message has been sent.
`size`	Size of message in bytes.
`ID`	Rank of target UE.

4.10.3 int RCCE_recv(char *privbuf, volatile char *combuf, size_t combuf_size, RCCE_FLAG *ready, RCCE_FLAG *sent, size_t size, int ID)

`RCCE_recv()` receives an arbitrarily sized buffer (does not have to be a multiple of 32 bytes) of data from the private memory of the sending UE to the private memory of the receiving UE. Its parameters must be matched exactly by those on the sending side.

`privbuf`	Starting address (private memory) of data to be received.
`combuf`	Starting address of intermediate MPB memory through which data are transferred.
`combuf_size`	Size in bytes of intermediate space in MPB used for this communication.
`ready`	Flag variable that indicates whether the receiver is ready to receive message.
`sent`	Flag variable that indicates that the message has been sent.
`size`	Size of message in bytes.
`ID`	Rank of target UE.

4.10.4 int RCCE_recv_test(char *privbuf, volatile char *combuf, size_t chunk, RCCE_FLAG *ready, RCCE_FLAG *sent, size_t size, int ID, int *test)

`RCCE_recv_test()` is the non-blocking version of `RCCE_recv()`. It returns immediately. It does not wait for the required message to actually arrive. If it has, it is stored in the proper location in private memory and the variable `test` is set to one. If it hasn't, `test` is set to zero.

`privbuf`	Starting address (private memory) of data to be received.
`combuf`	Starting address of intermediate MPB memory through which data are transferred.
`combuf_size`	Size in bytes of intermediate space in MPB used for this communication.
`ready`	Flag variable that indicates whether receiver is ready to receive message.
`sent`	Flag variable that indicates that message has been sent.
`size`	Size of message in bytes.
`ID`	Rank of target UE.
`test`	Variable that indicates whether the receive was (1) or was not (0) successful

5 Power Management API

The Power management API is the same for both basic and gory interfaces. Since most programs will not use this API, by default it is not included in the RCCE build process. Note that the Power Management API is an active area of research for RCCE. Expect changes to the functionality and the API as we gain experience with SCC power management.

5.1 Power Management

```
int    RCCE_power_domain(void)
int    RCCE_power_domain_master(void)
int    RCCE_power_domain_size(void)
int    RCCE_iset_power(int, RCCE_REQUEST *, int, int)
int    RCCE_wait_power(RCCE_REQUEST *)
int    RCCE_set_frequency(int, int *)
```

5.2 Power Management: Description

RCCE's power management functions are available if the library and application are compiled with `PWRMGMT=1`. Specify `PWRMGMT=1` on the `make` command line. To understand power management for the SCC processor, you need to understand the interplay between frequency and voltage and how these interact with the global clock that drives the chip.

The frequency for any component within SCC is defined relative to a global clock set during platform start-up. This clock can vary from one session to another, but in almost every case it is fixed such that the reference frequency of the chip is 1.6 GHz. All frequencies within the SCC processor are defined in terms of a divider applied to this base frequency. The values for the divider are 1 to 16.

The frequency can be changed for the mesh and the tiles. Currently, RCCE provides an API for changing the frequency of the tiles, not the entire mesh. For the tiles in the current version of the SCC processor, the maximum allowed tile frequency for a global clock at 1.6GHz is 800MHz. This corresponds to a divider of 2.

The SCC processor also lets you change the voltages used within the chip. Once again, we are only concerned with the voltage used for the tiles. Voltages can vary from a minimum of 0.6v to a maximum of 1.3v in steps of size 0.1v. These are nominal voltages.

Although frequency can vary at the granularity of a tile, RCCE voltages are set for a block of four tiles. These blocks are called voltage islands. There are six voltage islands on the SCC processor as shown in Figure 1. Notice the mapping from core ID to voltage domain. Regardless of hardware details which could modify actual IDs, in RCCE we will maintain these logical IDs.

Voltage and frequency can be varied independently, but they are not truly independent. For a given voltage, there is a maximum allowable frequency. Exceeding that frequency can destabilize if not actually damage a chip. The RCCE power management API opts for a safe approach that varies frequency and voltage together.

Because voltage and frequency are varied together within a voltage island, RCCE calls voltage islands power domains.

You define the divider for the desired frequency (which ranges from 2 to 16). RCCE internally determines the minimum voltage appropriate for that frequency. A voltage change request is

initiated. This is a high latency operation (up to millions of network cycles). The function that initiates a power change returns immediately with a request ID. A call to a “wait” function takes that request ID and blocks until the change has completed. In this way, you can overlap useful work with the power change.

If voltage and frequency changes are carried out incorrectly, the system may experience problems. Currently, to minimize this possibility, RCCE implements a restricted power management protocol.

- First, RCCE designates a master controller for each voltage island. power domain. The identity of this core is exposed through the power management API. Only this master controller can issue frequency or voltage change requests; for other cores in a given power control domain, the call to such functions returns immediately (with a return value of `RCCE_SUCCESS`).
- Second, while the frequency can in principle be changed on a per tile basis, RCCE sets the frequency collectively within a power domain.

Cores within a power domain will in some cases need to participate in collective communication operations with all the cores in a power domain. To support these collectives, we define a communicator `RCCE_P_COMM` for each power domain.

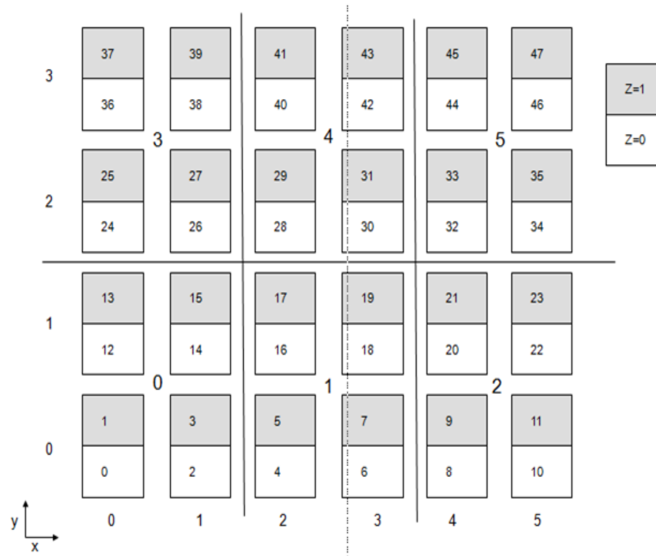

Figure 1 SCC processor logical layout showing core IDs, (x,y) tile coordinates, and power domains (0 through 5).

5.2.1 int RCCE_power_domain(void)

`RCCE_power()` returns the sequence number (zero through five) of the physical power domain of the calling UE. The power domains and RCCE core IDs are shown in Figure 1.

5.2.2 int RCCE_power_domain_master(void)

`RCCE_power_domain()` returns the rank (UE) of the master of the local power domain.

5.2.3 int RCCE_power_domain_size(void)

`RCCE_power_domain()` returns the number of cores within the local power domain that are participating in the computation.

5.2.4 int RCCE_iset_power(int Fdiv, RCCE_REQUEST *request, int * Fdiv_new, int * Vlevel_new)

`RCCE_iset_power()` is a non-blocking function that initiates a discrete stepping of the power and frequency within the power domain of the calling UE. It only initiates a change in the power and the frequency when called by the master of the power domain. For other UEs in the power domain, the function returns RCCE_SUCCESS immediately without issuing any power management commands.

The input to the function specifies the desired tile frequency divider (an integer ranging from 2 to 16). The function will determine the lowest voltage level that is consistent with the input value of the frequency divider and initiate the voltage change. Output values define the actual settings for the frequency divisor and the voltage level.

`RCCE_iset_power()` returns a `request` that is used in a call to `RCCE_wait_power()` to block the issuing core until the target power level has been reached. There can only be a single power change request in flight at any one time for each power domain. Additional requests are denied, and the function returns an error code.

`Fdiv`	The requested value for the frequency divisor
`request`	Request identifier.
`Fdiv_new`	The tile frequency divisor that will be active once the voltage change is complete.
`Vlevel_new`	The voltage level (0 to 7) once the voltage change is complete. In volts, this equals (0.6 + `Vlevel_new`* 0.1).

5.2.5 int RCCE_wait_power(RCCE_REQUEST *request)

`RCCE_wait_power()` blocks the power domain master UE until the previously specified power level has been reached (see `RCCE_iset_power()`). The supplied `request` parameter is reset and can be used for a new power stepping command.

`request`	Request identifier.

5.2.6 int RCCE_set_frequency_divider(int Fdiv, int Fdiv_new)

`RCCE_set_frequency_divider()` changes the frequency in the power domain of the calling UE without affecting the voltage. This function only initiates the desired change in the frequency if called by the master UE of the power domain. For other UEs, the call returns immediately with a return code of `RCCE_SUCCESS`. Because frequency changes are almost instantaneous, RCCE does not provide a non-blocking version to hide latency. The `RCCE_set_frequency_divider()` function will not allow a programmer to set a frequency divisor that results in a frequency too high for the current voltage.

`Fdiv`	The input requested value for the frequency divisor.
`Fdiv_new`	The new value of the frequency divisor.

6 The RCCE Emulator

To validate the RCCE API and get an early start on developing software for SCC, we created a functional emulator of RCCE. It runs on top of OpenMP, since this provides a *least common denominator* that extends across the platforms we care about (Linux, Windows, and LRB simulators). Currently, the emulator is only available for C programs.

6.1 Key Elements

The key elements of the emulator are the following.

`RCCE_emulator_driver` A program to set up the MPBs to emulate the SCC communication environment and to create the threads that mimic UEs.

`RCCE library` The collection of functions that make up RCCE.

`RCCE.h` The header file with function prototypes, references to other required header files, error return codes, and any types we need to define.

In addition, the user of the emulator writes a function called `RCCE_APP()`.

`RCCE_APP(int argc, char **argv)`

This function is the point of entry into the SCC application by all UEs. A conforming program can not assume that `RCCE_APP()` implies a barrier. On actual SCC hardware the programmer can replace `RCCE_APP()` with `main()`, relink with the native RCCE library, and then run the program unchanged.

6.2 Caution about Global Data

RCCE programmers need to be careful with global data in their programs.

The emulator is based on OpenMP, which in turn is based on threads with global data (such as data with a static storage class) shared between threads.

Use of the SCC chip is based on processes with global data local to each UE. This creates a fundamental inconsistency between SCC and the emulator.

Programmers will need to take this inconsistency into account and avoid global data in RCCE programs or manually make such data `threadprivate` for the emulator and then remove the `threadprivate` clauses when moving to SCC, or use conditional compilation. In future versions of RCCE, we may (partly) insulate the programmer from these issues by adding a `RCCE_GLOBAL` macro.

7 Appendix A: Glossary

Collective Operation	An operation in RCCE that must be called jointly by all UEs. It does not imply any synchronization. It does require, however, that the collective operation has the same program order with respect to RCCE operations on all UEs.
CPU	The entire many-core chip (all 48 cores). We use CPU when referring to the processor in a single socket.
MPB	The on-chip shared memory region physically associated with each tile and logically associated with each UE. This memory defines a general shared data space, but in RCCE, this is used strictly as a communications buffer. The MPB is logically segmented so that each UE has a local segment to use for its RCCE communication operations.
Partition	A logically related subset of cores often associated with a distinct power domain.
Private memory	Private memory refers exclusively to the off-chip DRAM associated with each core.
Processing Element (core)	The hardware element upon which a UE executes. For SCC, this is the core (which includes the caches associated with the core).
UE	Unit of execution. This terms refers to a process, thread or other agent that moves the program counter forward. It is a shorthand for *thread or process*.
Tile	SCC is a tiled architecture. The tile is the unit of replication for organizing the cores on the chip. For SCC, a tile has two cores and one router.

8 Appendix B: RCCE and SCC Hardware

RCCE software must talk to SCC hardware, but it simplifies the SCC hardware interface it presents to the user. As a RCCE application programmer, you need not concern yourself with the details of SCC hardware.

However, RCCE is opensource software and, as part of your research, you may want to modify how RCCE works internally. To do that, you need to understand how RCCE talks to SCC hardware.

8.1 Tile ID

RCCE software and SCC hardware use different tile IDs. RCCE software indexes a tile starting with 0 and ending with 23. Tile 0 is the tile in the lower left. The RCCE tile ID increments as you move to the left. The RCCE tile ID in the lower right is 5. The next row starts with a RCCE tile ID of 6 and ends with 11. The top row (the fourth row) starts with a RCCE tile ID of 18 and ends with 23. There are 24 tiles, from 0 to 23.

SCC hardware uses the TileID register to store the tile ID. The SCC hardware tile ID is an 8-bit hex value with y in the upper 4 bits and x in the lower 4 bits. Note that defining the SCC hardware tile ID in this way means that it is not continuous.

The SCC tile ID also increments as you move to the left; x increments and y stays the same and you move left in a row. The SCC tile ID in the lower left is 0x00 (0d). The SCC tile ID in the lower right is 0x05 (5d). These tile IDs are written as 0x*yx*. The next row starts with a SCC tile ID of 0x10 (16d) and ends with 0x15 (21d). The top row (the fourth row) starts with a SCC tile ID of 0x30 (48d) and ends with 0x35 (53d). Figure 2 shows the RCCE tile IDS and SCC IDs.

8.2 Power Domains

SCC hardware refers to voltage domains. RCCE calls them power domains because RCCE combines voltage and frequency into a domain.

RCCE numbers its six power domains from 0 to 5. Power domain 0 is the 2x2 tile array in the lower left. The power domains increment as you move to the right. The power domain in the lower right is 2. The second row starting from the left is 3, 4, 5.

SCC hardware recognizes 8 voltage domains, labeled V0 through V7. Voltage Domains V2 and V6 are the same and represent the entire mesh. V0 is the voltage domain in the upper left. The voltage domain increments as you move to the right, skipping V2. The voltage domains in the upper row are then V0, V1, V3. The voltage domains in the lower row are V4, V5, and V7. Note that V6 is skipped. Figure 2 shows the RCCE power domains and SCC voltage domains.

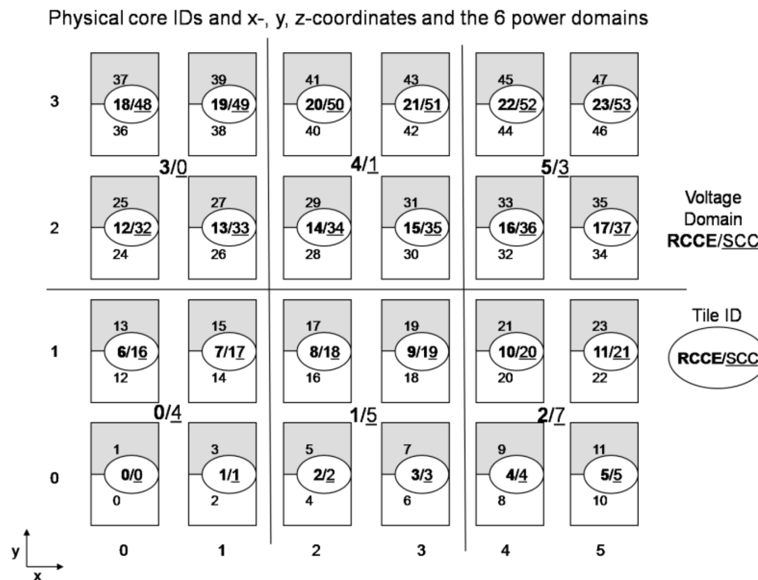

Figure 2 SCC and RCCE IDs

8.3 Changing the Voltage

RCCE changes the voltage and frequency together. RCCE does this for safety reasons.

If you access the hardware directly, you change voltage and frequency separately. Change the voltage by writing the VRC register with memory-mapped I/O. Its address is determined by the SCC LUT tables. It changes if those tables are modified.

The voltage is changed by writing a 17-bit value to the VRC register. Bit 16 must be 1. Bits 15:11 are don't-care. Bits 10:08 are the voltage ID called the VID. The VID uses the hardware convention for numbering a power domain. a power domain.

Bits 7:0 are the VID value. The VID value is 8 bits, specifying 256 values. Each step is 6.25mv. 0x00 is a voltage of zero; 0x01 is 6.25mv; 0x02 is 13mv, etc. However, the voltage maxes out at 1.3v. For safety reasons, you cannot set a voltage greater than 1.3v. VID values equal to and greater than 0xD0 (208d) all specify 1.3v.

8.4 Changing the Frequency

The router clock is initially set at either 800MHz or 1.6GHz. You chose a value when you booted Linux on the cores with either sccGui or sccBoot. When the router frequency is 800MHz, the default tile frequency is 533MHz. When the router frequency is 1.6GHz, the default tile frequency is 800MHz.

By writing bits 25:08 of the GCU with memory-mapped I/O, you can change the tile frequency. The GCU is the Global Clock Unit configuration register. Its address is determined by the SCC LUT tables. It changes if those tables are modified.

Table 1 lists the possible tile frequencies when the router clock is 800MHz. Table 2 lists the possible tile frequencies when the router clock is 1.6GHz. The tile frequency in both tables is 1.6GHz divided by the RCCE frequency divider.

Tile Frequency (MHz)	RCCE Frequency Divider	GCU Config Setting [25:08]
800	2	00 0111 0000 1110 0001
533	3	00 1010 1000 1110 0010
400	4	00 1110 0000 1110 0011
320	5	01 0001 1000 1110 0100
266	6	01 0101 0000 1110 0101
228	7	01 10001 000 1110 0110
200	8	01 11000 000 1110 0111
178	9	01 11111 000 1110 1000
160	10	10 00110 000 1110 1001
145	11	10 01101 000 1110 1010
133	12	10 10100 000 1110 1011
123	13	10 11011 000 1110 1100
114	14	11 00010 000 1110 1101
106	15	11 01001 000 1110 1110
100	16	11 10000 000 1110 1111

Table 1 Tile Frequency Settings for Router Clock of 800MHz

Tile Frequency (MHz)	RCCE Frequency Divider	GCU Config Setting [25:08]
800	2	00 0111 0000 0111 0001
533	3	00 1010 1000 0111 0010
400	4	00 1110 0000 0111 0011
320	5	01 0001 1000 0111 0100
266	6	01 0101 0000 0111 0101
228	7	01 1000 1000 0111 0110
200	8	01 1100 0000 0111 0111
178	9	01 1111 1000 0111 1000
160	10	10 0011 0000 0111 1001
145	11	10 0110 1000 0111 1010
133	12	10 1010 0000 0111 1011
123	13	10 1101 1000 0111 1100
114	14	11 0001 0000 0111 1101
106	15	11 0100 1000 0111 1110
100	16	11 1000 0000 0111 1111

Table 2 Tile Frequency Settings for Router Clock of 1.6GHz