The SccKit 1.4.0 User's Guide

Revision 0.92

(Parts 1-7)

What's New in sccKit 1.4.0

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 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.

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

Revision History for Document

0.9	Initial revision
0.91	Added tables for IODELAY and Power Management
0.92	Added range for eMAC general configuration registers; changed atomic increment table to state that a read returns the old value, not the updated value.

Table of Contents

List of Tables

List of Figures

1 Introduction

The sccKit provides an interactive user interface to the SCC cores. It has both a GUI and a command line interface. With the sccKit, you can train the system interface (the interface to the MCPC), load SCC Linux on the cores, display and modify SCC configuration registers, monitor SCC performance, as well as many other actions.

This manual describes the additional features that are part of sccKit Release 1.4.0. Key to this release is an expanded FPGA functionality.

Recall that the SCC board (called the Rocky Lake board) consists of a BMC (the Board Management Controller), a FPGA, and the Rock Creek chip. The MCPC (the Management Console PC) has two ethernet ports (one called eth0 that connects to the Internet and another called eth1 that connects to the BMC.) and a PCIe interface.

- The BMC is used for hardware control and monitoring. Users telnet into the BMC to power on/off the Rock Creek chip, read status, etc. The BMC connects to the cores with a JTAG interface. The BMC also connects to the FPGA.
- The FPGA takes the place of an SCC chipset. It connects to the Rock Creek chip over the PCIe interface. It connects to the Rock Creek chip over the SIF (system interface).
- The Rock Creek chip is a 4 X 6 (x X y) array of 24 tiles with two P54C cores per tile.

With Release 1.4.0, the SCC board makes use of its eMAC interface. Previously, the cores communicated with the MCPC over the PCIe interface. With 1.4.0, the cores use both the PCIe interface and the eMAC interface.

This manual introduces the features that are new with sccKit 1.4.0. These include the eMAC interface, additional FGPA registers (the global timestamp, the global interrupt, and the atomic increment counters), the virtual display, and the ability to enable the software RAM and software UART. Then, the manual describes the architecture of the FPGA so that users can write their own operating system to run on the cores or modify the existing implementation.

2 Summary of Additional Features

The key features that are available with sccKit 1.4.0 are the eMAC interface and additional FPGA registers. Other features include a virtual display and the Software RAM and Software UART. In addition, sccKit 1.4.0 provides a new command called `sccWrite` and new options to the commands, `sccDump`, `sccBoot`, and `sccPowercycle`.

Although not part of the release, POP-SHM (privately owned, public shared memory) is available for use. It's currently working, but has not been tested enough to go beyond the beta level. You need a custom SCC Linux (a POP-SHM enabled Linux). This custom Linux is available for download from our public SVN, but it is still at a beta stage.

Other features, not part of the release and not yet available for use, include a programming interface to the P54C performance counters, and a programming API to set and read the voltage in an SCC power domain. Also, in the future, the SCC board will enable a SATA interface and a self-booting feature. Self-booting means that the SCC will boot without the need for an MCPC, using a disk drive off the SATA interface.

For information about how to obtain the sccKit software and how to install it, refer to the file *How to Install sccKit 1.4.0*. If you are using a system in the SCC Data Center, sccKit 1.4.0 has been or will be installed for you.

3 Additions to the sccKit Command Line

3.1 sccWrite

With `sccWrite`, you can write off-chip memory, the message-passing buffer, a configuration register, a LUT entry, or a system interface FPGA address.

The switches and arguments to `sccWrite` are as follows:

`-c` or `--crb` *route address value*
: Write *value* to a 32- bit configuration register or LUT entry of tile 0xYX

-d or `--ddr3` *route address value*
: Write value to an off-chip memory location through the memory controller at tile 0xYX.

`-m` or `--mpb` *route address value*
: Write *value* to the message-passing buffer for tile 0xYX.

`-s` or `-sif` *address value*
: Write *value* to a 32-bit system interface FPGA register

Here is an example of using `sccWrite` to write the memory location at 0x84000000.

```
username@system:~$ sccWrite -d 0x00 0x84000000 0x80108001
INFO: Welcome to sccWrite 1.4.0 (build date Dec 21 2010 - 18:16:31)...
INFO: Wrote 0x80108001 to address 0x084000000 in DDR3 memory (MC of Tile
0x00)...
username@system:~$
```

Note that you cannot run `sccWrite` while also executing sccGui. If you try, you will get a message that you are unable to connect to the PCIe driver because it is in use.

You can also write the location 0x84000000 with the flit widget in sccGui. After writing that location with the `sccWrite` example just shown, you can read that same location with the flit widget. Figure 1 shows what you get after clicking the READ button and specifying the address 0x84000000 and tile 0x00, which has a memory controller on its West Port, PERIW.

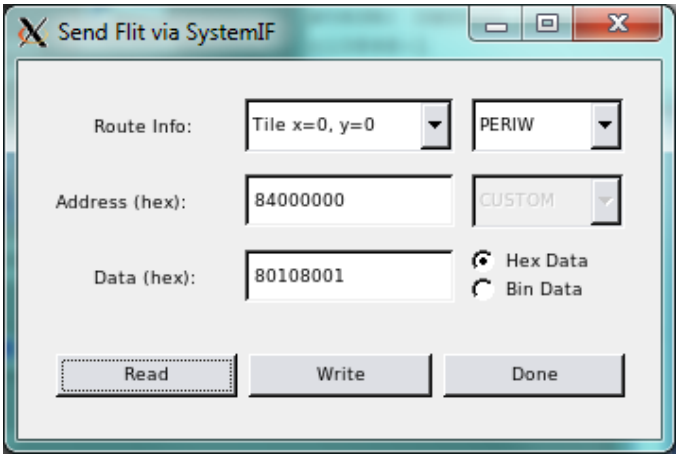

Figure 1: Using the flit widget in sccGui to read the value at the location 0x84000000

3.2 sccDump

The command `sccDump` has an additional switch that allows you to display the value of sthe system interface FPGA registers.

The new switch and argument to the `sccDump` command are as follows:

`-s` or `--sif` *address*
: Dump system interface FPGA register at *address*.

Here is an example of using `sccDump` to display the value of the SIF ID register a 0x8010.

```
username@system:~$ sccDump -s 0x8010
INFO: Packet tracing is disabled...
INFO: Initializing System Interface (SCEMI setup)....
INFO: Successfully connected to PCIe driver...
INFO: Welcome to sccDump 1.4.0 (build date Dec 21 2010 - 18:14:44)...
INFO: FPGA register at address 0x00008010 (32784d) contains 0x20110308
(537985800d)...
username@system:~$
```

3.3 sccBoot

The command `sccBoot` has an additional switch that allows you to run the DDR3 memory test. Previously, you could only do this from sccGui.

The new switch and argument to the `sccBoot` command are as follows:

`-m` or `--memory`
: Tests the complete DDR3 memory. This option takes a while. After execution the whole DDR3 system memory (all four memory controllers) are initialized with zero.

3.4 sccPowercycle

Without switches the command `sccPowercycle` does what it did before. `sccPowercycle` now has three additional switches that allow you to load a new bitstream and choose between a hard and a soft power cycle.

The new switches and arguments to the `sccPowercycle` command are as follows:

-s *on*/*off*	Switches the connected Rocky Lake or Copperridge on or off.
-f *bitstream*	Loads a new bitstream file. The specified bitstream file must be on the USB stick. You must provide a complete pathname. For example, the pathname might be `/mnt/flash4/rl_20110308_ab.bit`.
-r	Performas a hard power cycle; powers off the SCC, sleeps, powers it back on, rescans PCI, resets the SCC, resets the FPGA, and reloads `crbif`.
noargs	resets the SCC, resets the FPGA, reloads `crbif`. This is the legacy version.

4 The eMAC Interface

In earlier versions of sccKit, `eth1` was only used to connect to the BMC. Programs running on the cores did not know about eth1. With sccKit 1.4.0, `eth1` is part of the eMAC interface. Programs running on the cores can send data to the MCPC over `eth1`. You still need a way to telnet to the BMC. This is often done with a virtual ethernet connection called `eth1:1` that uses the same hardware as `eth1`.

The driver `crbif` is the software that facilitates communication between the MCPC and the Rocky Lake board. The name `crbif` is a legacy name. The `crb` stands for Copper Ridge Board, a predecessor to the Rocky Lake board. The driver `crbif` has two components.

- The PCIe interface to the sccKit software components (for example, the sccGui and sccKit commands)
- The Ethernet device `crb0` that communicates with the device `pc0` on the actual SCC core (Ethernet over PCIe).

With sccKit Release 1.4.0, the driver crbif is still loaded, but only the first component is active. With 1.4.0, the cores use Ethernet over eMAC to communicate with the MCPC.

With the older release, the core's routing table looks as follows.

```
root@rck00:~> route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
127.0.0.1       0.0.0.0         255.255.255.255 UH    0      0        0 lo
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 pc0
192.168.0.0     0.0.0.0         255.255.255.0   U     0      0        0 mb0
0.0.0.0         192.168.1.254   0.0.0.0         UG    0      0        0 pc0
root@rck00:~>
```

With the new release, the core's routing table looks as follows.

```
root@rck00:~> route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
127.0.0.1       0.0.0.0         255.255.255.255 UH    0      0        0 lo
```

```
192.168.3.0     0.0.0.0         255.255.255.0   U     0      0        0 emac0
192.168.0.0     0.0.0.0         255.255.255.0   U     0      0        0 mb0
0.0.0.0         192.168.3.254   0.0.0.0         UG    0      0        0 emac0
root@rck00:~>
```

Ethernet over PCIe and Ethernet via EMAC are mutually exclusive. When you choose Ethernet over PCIe, the system operates as it did before 1.4.0. There is probably no reason why you would want to choose Ethernet over PCIe and not Ethernet over eMAC. Performance improves with Ethernet over eMAC. Etherenet over PCIe degrades to a legacy feature with a known high-I/O-load bug.

When you choose Ethernet over EMAC, the system communicates with the MCPC over the PCIe interface for sccKit commands and over the eMAC interface for core programs.

With earlier versions of sccKit, all communication with the MCPC went over the PCIe interface. Sometimes when a program did lots of I/O, it would hang the SCC system. This problem is alleviated when you use the eMAC interface.

The SCC system has four eMAC interfaces. They are labeled A, B, C, D. Currently only one interface is used. The examples here assume that the enabled eMAC interface is A, but it need not be.

Not all SCC boards (Rocky Lake boards) have four working eMAC ports. You can see what eMAC ports work on your SCC system by telnetting into the BMC. The signon message contains a line similar to

```
Usable GB ETH 1111
```

In this example, the 1111 indicates that all four eMAC ports are working. They read ABCD, left to right. If instead you saw 1000, then that would indicate that only your A port was working.

4.1 How to Tell if the eMAC Interface is Enabled

Enter the directory `/opt/sccKit` and display the file `systemSettings.ini`. The lines beginning with `maxTransId` are new with sccKit 1.4.0. The last line, `sccMacEnable=a`, indicates that the eMAC inteface is enabled and using port A.

```
username@yoursystem:/opt/sccKit$ cat systemSettings.ini
[General]
CRBServer=10.3.16.126:5010
memorySize=8
platform=RockyLake
maxTransId=64
sccFirstMac=00:45:4D:41:44:31
sccHostIp=192.168.3.254
sccFirstIp=192.168.3.1
sccMacEnable=a
username@yoursystem:/opt/sccKit$
```

4.2 Cabling for sccKit 1.4.0

With sccKit 1.4.0, the `eth1` interface on the MCPC is now part of the eMAC interface. However, the MCPC must still be on the same subnet as the BMC so that you can telnet to the BMC from the MCPC. You can satisfy this requirement in many ways. One way is to create a virtual Ethernet

connection called `eth1:1` to communicate with the BMC. Refer to *How to Install sccKit 1.4.0* for details.

5 Additional FPGA Registers

The new FPGA bitstream provided with sccKit 1.4.0 has several new registers available to programs running on the cores. The key additional registers fall into three categories; a global timestamp counter, atomic increment counters, and global interrupt registers. This section describes those key additional registers. For information about other FPGA capabilities refer to **Error! Reference source not found.**.

Access these new FPGA registers through a new LUT entry.

```
Entry 0xf9: {9'h003,3'b101,10'h100};
```

Recall how the LUT translation works. The LUT translates a 32-bit core address into a 34-bit system address. The 0xf9 selects a LUT entry that returns 22 bits. The lower 10 bits of these 22 bits are prepended to the lower 24 bits of the core address to obtain the system address. Refer to the *SCC External Architecture Specification* for details.

When running SCC Linux, you can access one of the new FPGA registers as physical address

```
0xf9000000 + <the SIFCFG addr>
```

To see some example code on how to do this, look at Appendix A in the *SCC Programmer's Guide*. The example there reads the TileID register. It executes an `mmap()`, specifying a file descriptor obtained from opening `/dev/rckncm` and a physical address `CRB_OWN + MYTILEID` where

```
#define CRB_OWN   0xf8000000
#define MYTILEID  0x100
```

For example, to read the Global Timestamp Counter, replace `CRB_OWN` with `0xf9000000` and `MYTILEID` with 0x08224 to read the lower 32 bits of the counter and 0x8228 to read the upper 32 bits of the counter.

5.1 The Global Timestamp Counter

The Global Timestamp Counter provides a common time base across all the cores. Each core does have its own timestamp counter, but these are not synchronized.

SIFCFG: TESET Addr	Reg	Bits	Type	Description
08224	Global Timestamp Counter	[31:0]	RO	Provides lower 32 bits of global timestamp counter based on 125MHz system clock of the FPGA
08228		[31:0]	RO	Provides upper 32 bits of global timestamp counter based on 125MHz system clock of the FPGA

Table 1: The Global Timestamp Counter

5.2 The Atomic Increment Counters

The FPGA provides 2 * 48 = 96 32-bit atomic increment counters. The addresses of the counter sets start at the 4K page boundaries 0xE000 and 0xF000

Each counter consists of a pair of registers, the atomic increment counter register and the initialization counter register. The initialization counters alternate with the increment counters.

The atomic counter can be loaded with a 32 bit value though a write access to the initialization counter register. A read to this register provides the actual value of the atomic counter. Reading and writing to the atomic increment counter register increments (read) or decrements (write) the counter value. In case of a read access, the unmodified value will be provided back as read data.

SIFCFG: TSET Addr	Reg	Bits	Type	Description
0E000	Atomic Increment Counter #0	[31:0]	R/W	Read access causes an atomic increment of the register value. The old value gets returned. Write access causes an atomic decrement of the register value.
0E004	Initialization Counter #0	[31:0]	R/W	Read access returns the current register value. Write access initializes the register with new value.
0E008	Atomic Increment Counter #1	[31:0]	R/W	Read access causes an atomic increment of the register value. The old value gets returned. Write access causes an atomic decrement of the register value.
0E00C	Initialization Counter #1	[31:0]	R/W	Read access returns the current register value. Write access initializes the register with new value.
...0E178	Atomic Increment Counter #47	[31:0]	R/W	Read access causes an atomic increment of the register value. The old value gets returned. Write access causes an atomic decrement of the register value.
…0E17C	Initialization Counter #47	[31:0]	R/W	Read access returns the current register value. Write access initializes the register with new value.
0F000	Atomic Increment Counter #48	[31:0]	R/W	Read access causes an atomic increment of the register value. The

SIFCFG: TSET Addr	Reg	Bits	Type	Description
				old value gets returned. Write access causes an atomic decrement of the register value.
0F004	Initialization Counter #48	[31:0]	R/W	Read access returns the current register value. Write access initializes the register with new value.
0F008	Atomic Increment Counter #49	[31:0]	R/W	Read access causes an atomic increment of the register value. The old value gets returned. Write access causes an atomic decrement of the register value.
0F00C	Initialization Counter #49	[31:0]	R/W	Read access returns the current register value. Write access initializes the register with new value.
...0F178	Atomic Increment Counter #95	[31:0]	R/W	Read access causes an atomic increment of the register value. The old value gets returned. Write access causes an atomic decrement of the register value.
…0F17C	Initialization Counter #95	[31:0]	R/W	Read access returns the current register value. Write access initializes the register with new value.

Table 2: The Atomic Increment Counters

5.3 Global Interrupt Controller

Even before sccKit 1.4.0, a core could interrupt another core, but to do that a core had to write into a configuration register on the other core.

sccKit 1.4.0 offers more features and greater flexibility. The FPGA now has 48 interrupt status registers, 48 interrupt mask registers, 48 interrupt reset registers, and 48 interrupt request registers, one for each core.

A core can interrupt another core by setting the corresponding bit in that core's request register. If that core's interrupt mask bit is not set, the interrupt occurs. The interrupted core then jumps to its service routine and reads its status register to determine where the interrupt came from.

When the interrupted core completes its service processing, it must clear its local interrupt and write to its reset register in the FPGA to clear its status and pending bits.

For each SCC core, the FPGA contains an interrupt logic unit in the register bank module which collects the events coming from internal interrupt sources or from IPIs sent by other cores and handles the generation of the interrupt packets.

An interrupt packet is a write request from the FPGA to the CRB register within the tile of the receiving core which sets one of the two local interrupt bits. These bits are directly connected to the LINT0/LINT1 pins of the core. The current FPGA architecture supports 6 internal interrupt sources as well as 48 IPI sources: IPI0…IPI47, eMAC0…eMAC3 and two reserved sources. The interrupt events from the eMAC modules are generated when the HW has written new packets into the buffer.

In summary, the interrupt control flow is as follows.

1. Interrupt event(s) or IPI(s) set status bit(s)
    1. If not masked and no IRQ is pending, send IRQ packet to core.
    2. Otherwise do not send packet.
2. Core receives interrupt and jumps to service routine
    1. Reads status register.
    2. Determines interrupt source(s) and starts processing.
3. When processing has been finished
    1. Clears local interrupt bit in CRB.
    2. Writes to IRQ-Reset register to clear IRQ pending bit and the corresponding status bit(s).
4. If one of the status bits is still active, the next IRQ packet is sent to core

.

Figure 2 shows an overview of the global interrupt controller.

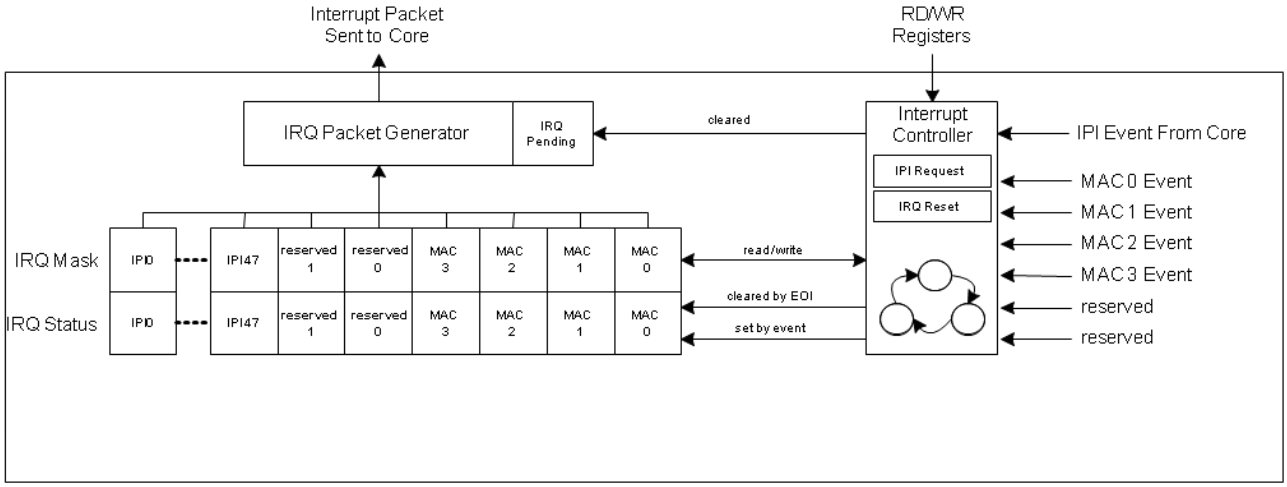

Figure 2: Overview of the global interrupt controller

5.3.1 Global Interrupt Controller Registers

SIFCFG: IRQ Addr	Reg	Bits	Type	Description
0D000	Interrupt Status Core 0	[31:0]	RO	High-active status set by HW can be read. [5:0] = reserved, eMAC3..eMAC0 [31:6] = IPI25…IPI0
0D004	Interrupt Status Core 0	[22:0]	RO	High-active status set by HW can be read. [22:0] = MCPC, IPI47…IPI26
		[31:23]	R0	Unused
…	…			
0D178	Interrupt Status Core 47	[31:0]	RO	High-active status set by HW can be read. [5:0] = reserved, eMAC3..eMAC0 [31:6] = IPI25…IPI0
0D17C	Interrupt Status Core 47	[22:0]	RO	High-active status set by HW can be read. [22:0] = MCPC, IPI47…IPI26
		[31:23]	R0	Unused
0D200	Interrupt Mask Core 0	[31:0]	R/W	High-active mask bit to disable interrupt delivery individually [5:0] = reserved, eMAC3..eMAC0 [31:6] = IPI25…IPI0
0D204	Interrupt Mask Core 0	[22:0]	R/W	High-active mask bit to disable interrupt delivery individually [22:0] = MCPC, IPI47…IPI26
		[31:23]	R0	Unused
…	…			
0D378	Interrupt Mask Core 47	[31:0]	R/W	High-active mask bit to disable interrupt delivery individually [5:0] = reserved,

SIFCFG: IRQ Addr	Reg	Bits	Type	Description
				eMAC3..eMAC0 [31:6] = IPI25…IPI0
0D37C	Interrupt Mask Core 47	[22:0]	R/W	High-active mask bit to disable interrupt delivery individually [22:0] = MCPC, IPI47…IPI26
		[31:23]	R0	Unused
0D400	Interrupt Reset Core 0	[31:0]	WO	Writing 1 to a bit position clears the according bit in the status register. [5:0] = reserved, eMAC3..eMAC0 [31:6] = IPI25…IPI0
0D404	Interrupt Reset Core 0	[22:0]	WO	Writing 1 to a bit position clears the according bit in the status register. [22:0] = MCPC, IPI47…IPI26
		[31:23]	R0	Unused
…	…			
0D578	Interrupt Reset Core 47	[31:0]	WO	Writing 1 to a bit position clears the according bit in the status register. [5:0] = reserved, eMAC3..eMAC0 [31:6] = IPI25…IPI0
0D57C	Interrupt Reset Core 47	[22:0]	WO	Writing 1 to a bit position clears the according bit in the status register. [22:0] = MCPC, IPI47…IPI26
		[31:23]	R0	Unused
0D600	IPI Request Core 0	[31:0]	WO	Writing 1 to a bit position sets the according bit in the status register. [31:0] = IPI31…IPI0
0D604	IPI Request Core 0	[15:0]	WO	Writing 1 to a bit position sets the according bit in the status register. [15:0] = IPI47…IPI32

SIFCFG: IRQ Addr	Reg	Bits	Type	Description
		[31:16]	R0	Unused
…	…			
0D778	IPI Request Core 47	[31:0]	WO	Writing 1 to a bit position sets the according bit in the status register. [31:0] = IPI31…IPI0
0D77C	IPI Request Core 47	[15:0]	WO	Writing 1 to a bit position sets the according bit in the status register. [15:0] = IPI47…IPI32
		[31:16]	R0	Unused
0D800 …0D8BC	Interrupt Config Core 0…47	[0]	RW	Configures the local interrupt pins of the core that will be used 0 – LINT 0 1 – LINT 1
		[31:1]	RO	Reserved
0D900	IRQ Request MCPC	[31:0]	WO	Writing 1 to a bit position sets the according bit in the status register. [31:0] = MCPC31…MCPC0
0D904	IRQ Request MCPC	[15:0]	WO	Writing 1 to a bit position sets the according bit in the status register. [15:0] = MCPC47…MCPC32
		[31:16]	R0	Unused

Table 3: The global interrupt registers

6 The Virtual Display

To use the virtual display, you must have an MCPC desktop, not just an ssh connection. You can get a desktop by making an ssh connection to your MCPC and starting up vncserver.

Then, on your local machine, run a VNC client. If your local machine is running Windows 7, the clients RealVNC and Ultr@VNC Viewer work fine. Refer to the file *How to VNC into the SCC DC from Windows* on http://communities.intel.com/community/marc.

To start a virtual display on cores 0 through 1, enter

```
sccDisplay -u 0..1
```

If you leave out the core specification, you start up a virtual display on all cores reachable by a ping. Press `<Right CTRL>H` to get the help screen as shown in Figure 3. When you enable

broadcasting, it is enabled for all cores. This is different from `scccKonsole`. With `sccKonsole`, you can enable broadcasting for individual cores.

Exit with `<Right CTRL>Q`. Do not X the window.

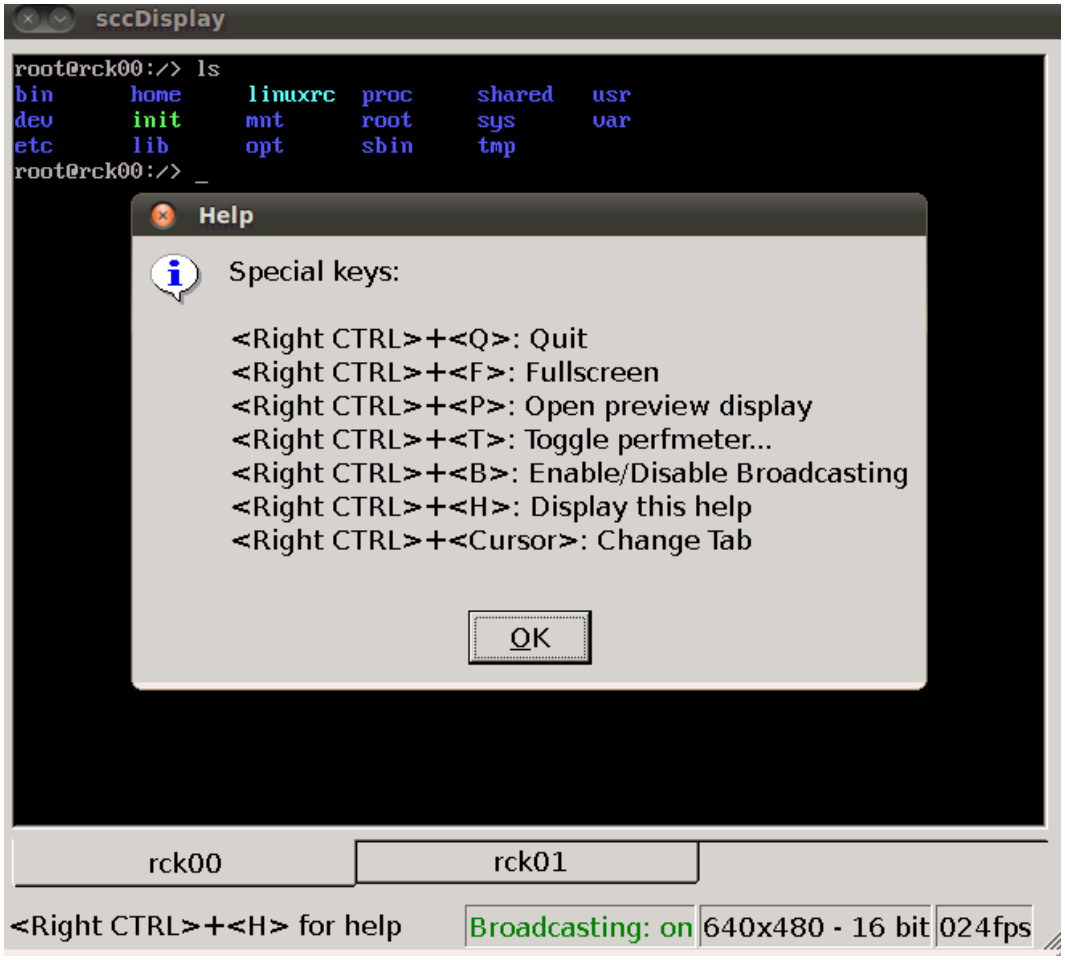

Figure 3: sccDisplay showing the help screen with broadcasting enabled.

Note the `-u` in the `sccDisplay` invocation. This tells `sccDisplay` to use a UDP connection to transfer keyboard strokes and mouse-movements from `sccDisplay` to the actual SCC core.

A newer Linux image (not yet available) allows you to use TCP. TCP provides error checking and is more reliable.

You can set the virtual display resolution as shown in Figure 4 and Figure 5.

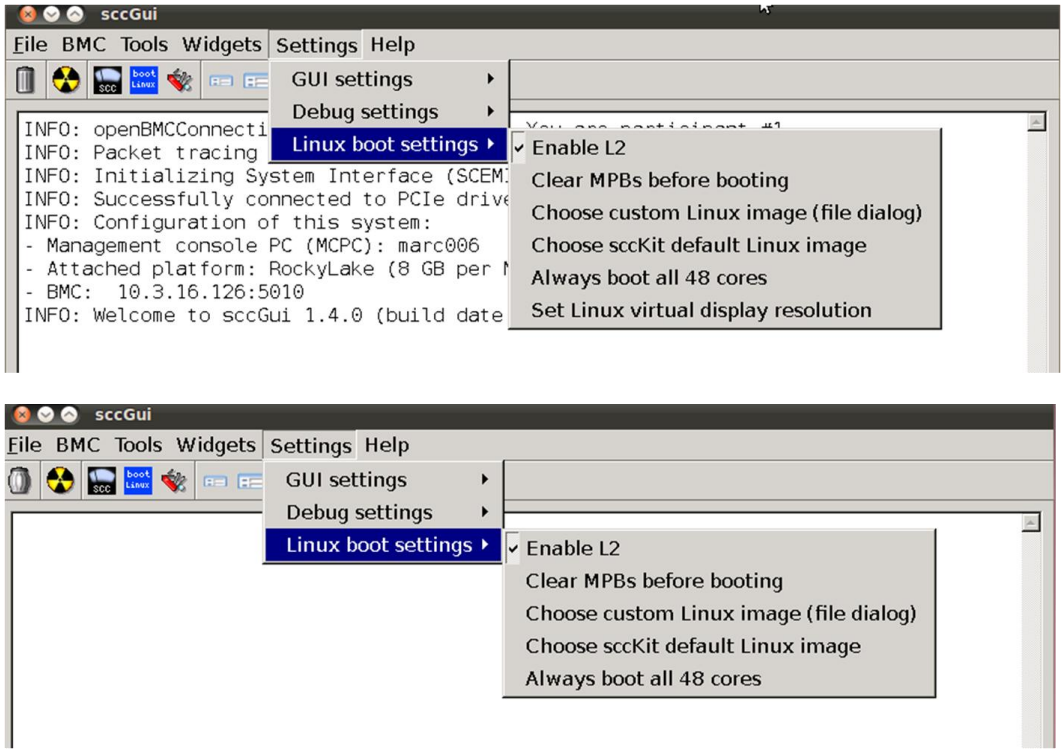

Figure 4: sccGui Linux boot settings for 1.4.0 (top) and 1.3.0 (bottom)

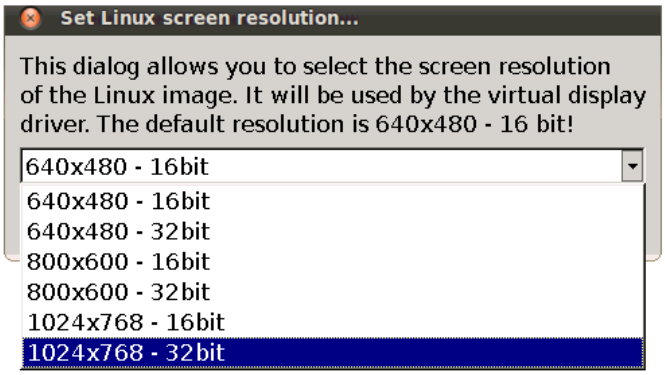

Figure 5: The menu dropdown that appears when you select Set Linux virtual display resolution.

Figure 6 shows a screenshot from a demo running `sccDisplay` on 48 cores. The screenshot shows the preview display, which you can get by issuing `<Right CTRL>P`. When you see the preview display, you can click in one of the tiles to make it the active window.

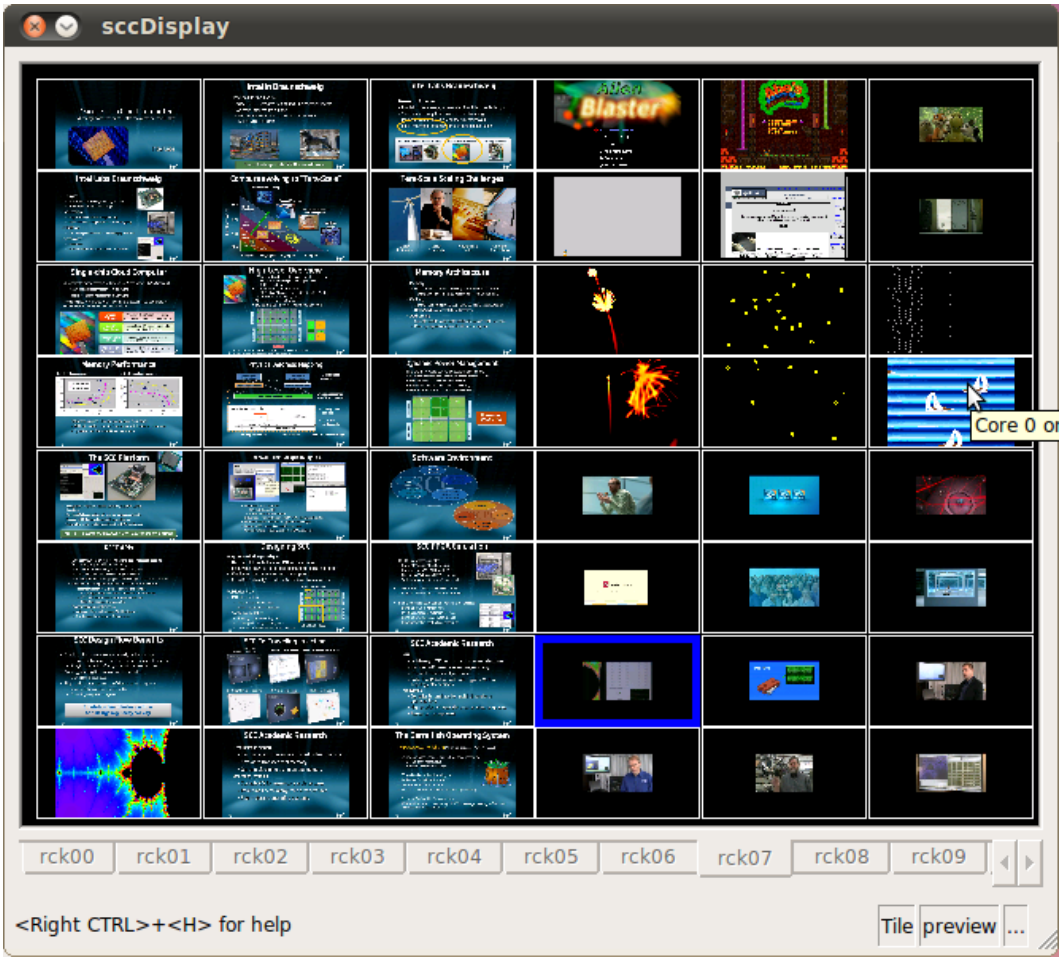

Figure 6: sccDisplay showing the graphical capabilities of all 48 cores at once

7 Software RAM and Software UART

To enable the Software RAM and Software UART feature, bring up sccGui and select `Settings➔Debug settings➔Enable software RAM and UART`. Figure 7 shows the additional menu item present with sccKit 1.4.0.

This feature is useful when debugging new Linux kernels. When this feature is enabled, all I/O writes to addresses 0x2f8 and 0x3f8 are printed on the software UART screen. This screen is a sccKonsole-like window. Once you enable the feature, the UART window will automatically pop-up when an SCC core writes to I/O ports 0x2f8 or 0x3f8.

Also, the software RAM feature is also useful when using printf in bareMetalC programs. Without it, you run the risk of a heavy I/O crash.

The FPGA generates automatic write responses when a core issues I/O-writes. Thus, the core is immediately free to send the next request to the FPGA. When all 48 cores "fire" I/O-writes to the System Interface, it will eventually crash. The Software RAM and Software UART feature disables the automatic generation of write responses. Instead, the I/O request reaches sccKit and

sccKit itself creates the response flit. This limits the maximum amount of I/O requests at a time to 48. Enabling this feature does prevent a crash, but will decrease performance.

This feature also enables the cores to access data from a virtual RAM (1GB of SoftRAM exists in sccGui). You can execute programs from that SoftRAM, allowing you to trace code fetches from the core.

Note that when you enable Software RAM and Software UART, you cannot use the `crbif` network driver. This is because the `crbif` network driver relies on the automatic response generation. Software RAM and Software UART will only work on eMAC enabled systems or systems that don't make use of a network (such as bareMetalC workloads). That is, with this feature, you must use Ethernet over eMAC, not Ethernet over PCIe.

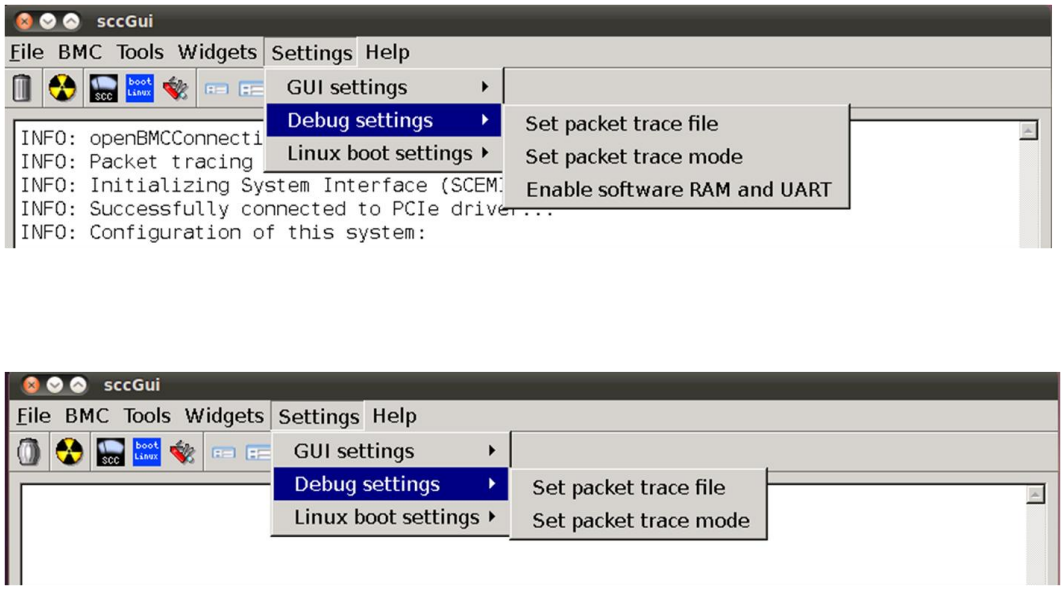

Figure 7: sccGui Debug settings for 1.4.0 (top) and 1.3.0 (bottom).