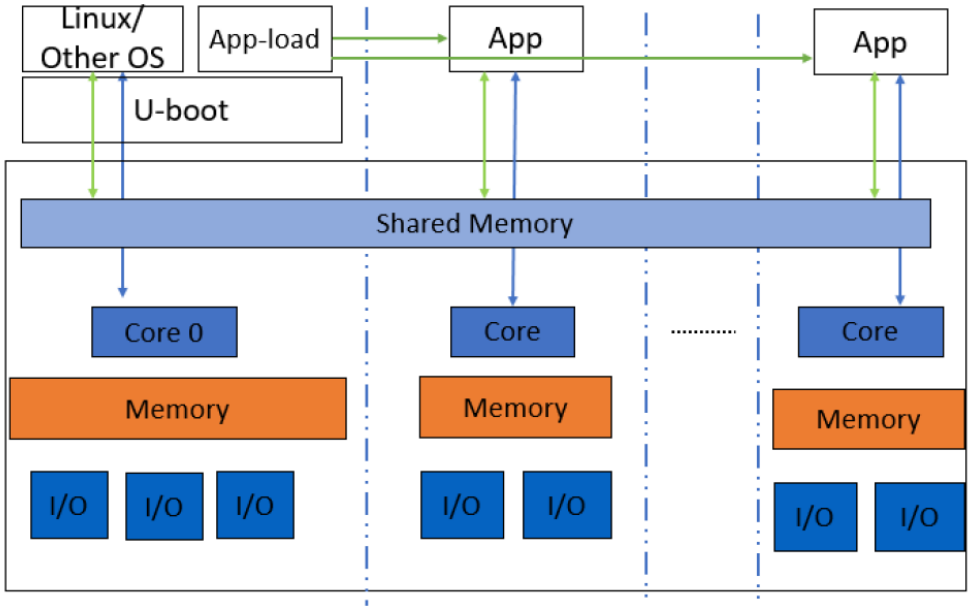
# Inter-core Communication Application Development Based on OpenIL Baremetal Framework

This document introduce OpenIL Baremetal framework architecture using NXP Layerscape platforms, describes how to run a sample baremetal project and inter-core Communication application development based on OpenIL Baremetal framework. Inter-core communication(ICC) application works on Linux core(master) and Baremetal core(slave), provides the data transfer between cores via SGI inter-core interrupt and share memory blocks.

## Baremetal Framework architecture for QorIQ Layerscape platforms.

The following figure depicts the baremetal framework architecture.




In Baremetal framework application, core0 runs as master which runs the operating system such as Linux, Vxworks. Slave cores run on the Baremetal application. Interrupts between different cores and high-performance mechanism for data transer, communicating via shared memory. The master core0 runns u-boot, it then loads the baremetal application to the slave cores and starts the baremetal application.

## Running Baremetal Binary

If using OpenIL to compile the baremetal image, the baremetal image is included in sdcard.img and the master core runs the baremetal image on slave cores automatically. If using the standalone compilation method, you need to perform the steps below to run the baremetal binary from U-Boot prompt of master core.

After starting U-Boot on the master, download the bare metal image: u-boot.bin on 0x84000000 using the command
below:

1. => *tftp 0x84000000 xxxx/u-boot.bin*

0x84000000 is the address of CONFIG_SYS_TEXT_BASE on bare metal

2. Then, start the baremetal cores using the command below:

*=> cpu start 0x84000000*

3. Last, the UART1 port displays the logs, and the bare metal application runs on slave cores successfully.

The figure below displays a sample output log.

```
U-Boot 2017.07-21736-g7fb4afc-dirty (Mar 15 2018 - 15:50:12 +0800)

CPU:   Freescale LayerScape LS1021E, Version: 2.0, (0x87081120)
Clock Configuration:
       CPU0(ARMV7):1000 MHz,
       Bus:300  MHz, DDR:800  MHz (1600 MT/s data rate),
Reset Configuration Word (RCW):
       00000000: 0608000a 00000000 00000000 00000000
       00000010: 20000000 08407900 60025a00 21046000
       00000020: 00000000 00000000 00000000 00038000
       00000030: 20024800 841b1340 00000000 00000000
I2C:   ready
DRAM:  256 MiB
EEPROM: NXID v16777216
In:    serial
Out:   serial
Err:   serial
Core[1] in the loop...
i2c read: 0xa0
[ok]i2c test ok
IRQ 0 has been registered as SGI
IRQ 195 has been registered as HW IRQ
SGI signal: Core[1] ack irq : 0
[ok]GPIO test ok
=>
```

## Inter-core communication(ICC) application Development Based on Baremetal Framework

ICC application is structured base on the following two basics:

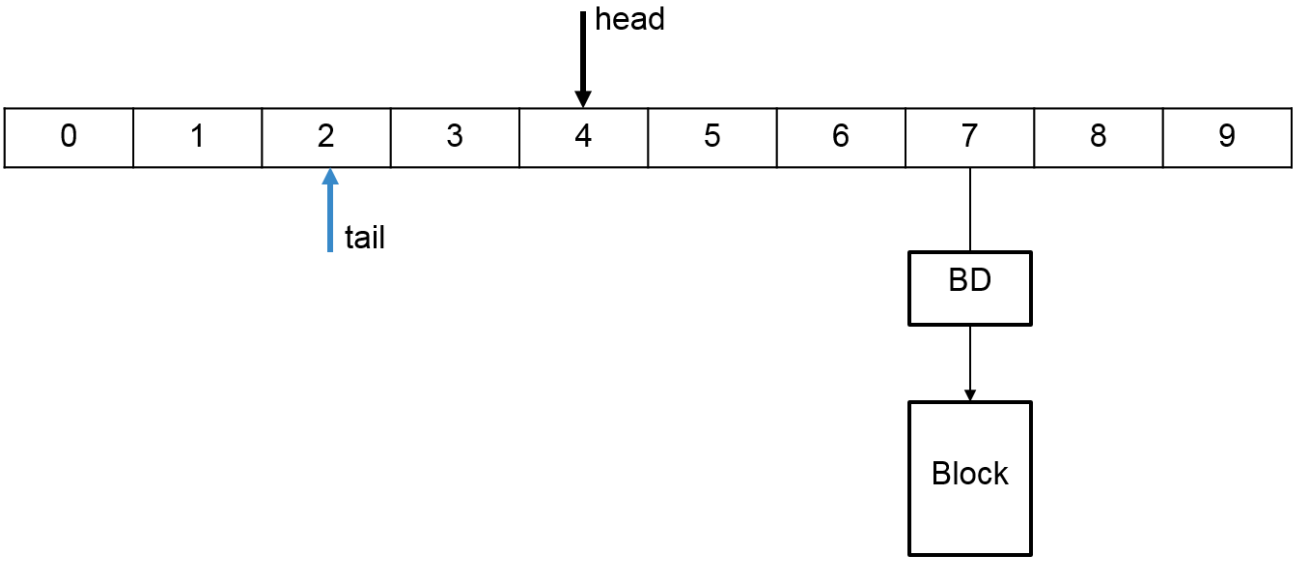
SGI: Software-generated Interrupts in ARM GIC, used to generate inter-core interrupts. The ICC module uses the number 8
SGI interrupt for all Linux and Baremetal cores

Shared memory: A memory space shared by all platform cores. The base address and size of the share memory should be
defined in header files before compilation

.

The figure below shows the basic operating principle for data transfer from Core 0 to Core 1. After the data writing and head point
moving to next, Core 0 triggers a SGI (8) to Core 1, then Core 1 gets the BD ring updated

status and reads the new data, then moves the tail point to next.



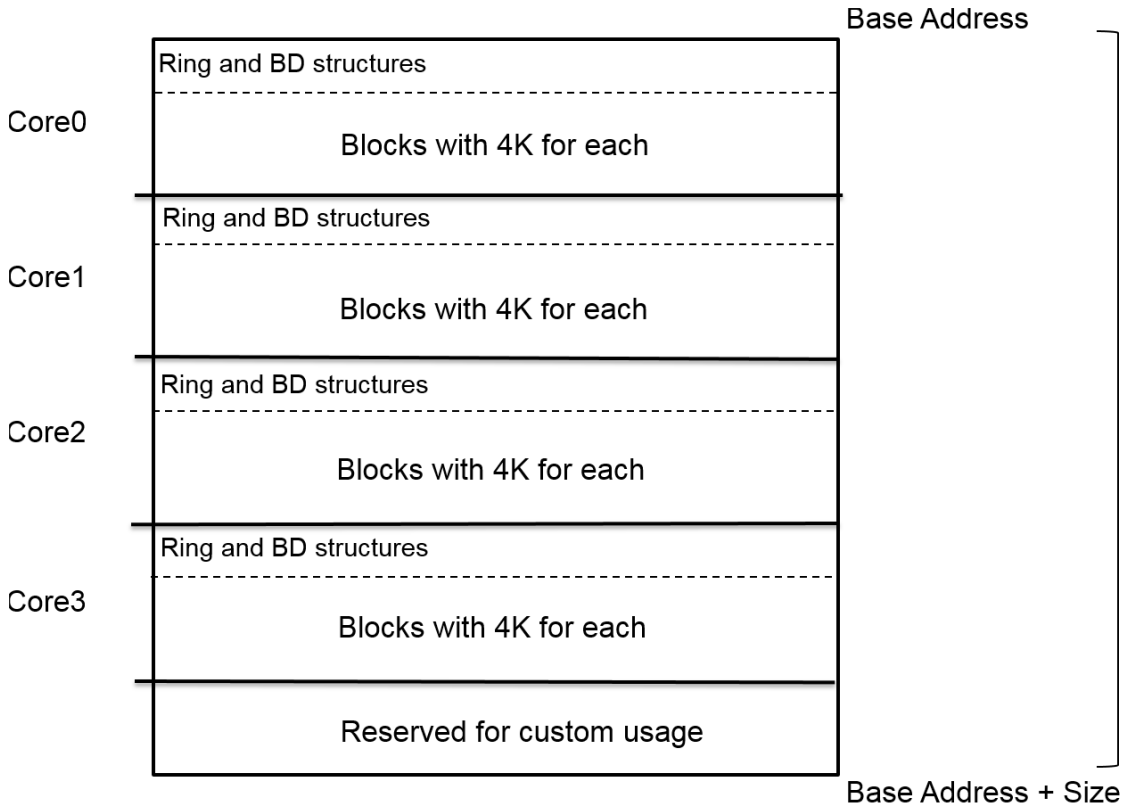

ICC callback handler for received data:

```
static void icc_irq_handler(int hw_irq, int src_coreid)
{
……
                ring = (struct icc_ring *)ICC_CORE_RING_BASE(src_coreid, mycoreid);
                valid = icc_ring_valid(ring);
                for (i = 0; i < valid; i++) {
                        desc = ring->desc + ring->desc_tail;
                        block_addr = desc->block_addr;
                        byte_count = desc->byte_count;

                        irq_handle = (void (*)(int, unsigned long, unsigned int))
                                g_icc_irq_cb[src_coreid];
                        if (irq_handle)
                                irq_handle(src_coreid, block_addr, byte_count);
                        else
                                printf(
                                        "Get the SGI %d from core %d; block: 0x%lx,
    byte: %d\n",
                                        hw_irq, src_coreid, block_addr, byte_count);

                        /* add desc_tail */
                        ring->desc_tail = (ring->desc_tail + 1) % ring->desc_num;
                }
                invalidate_dcache_range(CONFIG_SYS_DDR_SDRAM_SHARE_BASE,
                        CONFIG_SYS_DDR_SDRAM_SHARE_BASE +
                    CONFIG_SYS_DDR_SDRAM_SHARE_SIZE);
    }
```

All the ICC ring structures, BD structures and blocks for data are in the shared memory. A four-core platform ICC module would
map the shared memory as shown in the figure below.




Generally, Core 0 runs Linux as master core, other cores run Baremetal as slaves. They obtain the same size of share memory to structure the rings and BDs, and split the blocks space with 4k unit for each block. The reserved space at the top of the share memory is out of the ICC module and for the custom usage.

## Running ICC demo application

The ICC module command examples on LS1021A-IoT Linux (Core 0) + Baremetal (Core 1) system: Run icc send 0x2 0x1f 128 to send 128 bytes data 0x1f to core 1.

*root@OpenIL-Ubuntu:~# icc send 0x2 0x1f 128*
*gic_base: 0xb6fa0000, share_base: 0xa7e87000, share_phy: 0xb0000000, block_phy: 0xb0200000*
*ICC send testing ...*
*Target cores: 0x2, bytes: 128*
*ICC send: 128 bytes to 0x2 cores success*
*all cores: reserved_share_memory_base: 0xbf000000; size: 16777216*
*mycoreid: 0; ICC_SGI: 8; share_memory_size: 125829120*
*block_unit_size: 4096; block number: 30208; block_idx: 0*
*#ring 0 base: 0xa7e87000; dest_core: 0; SGI: 8*
*desc_num: 128; desc_base: 0xb0000048; head: 0; tail: 0*
*busy_counts: 0; interrupt_counts: 0*
*#ring 1 base: 0xa7e87024; dest_core: 1; SGI: 8*
*desc_num: 128; desc_base: 0xb0000448; head: 1; tail: 1*
*busy_counts: 0; interrupt_counts: 1*

At the same time, Core 1 prints the receival information.

=> Get the ICC from core 0; block: 0xb0200000, bytes: 128, value: 0x1f