LIBNVIPC SPECIFICATION

Document | December 21, 2022

Version 1.0

Document Change History

cuBB SDK 21-4 Release

Date	Authors	Summary of Change
Feb 25, 2022	Peter G	Initial release.

TABLE OF CONTENTS

Introduction

Purpose and Scope

This document describes the libnvipc library, which is a high-performance, shared memory IPC solution for communication between 5G-NR MAC and PHY processes.

Using this library, some CPU memory and GPU memory pools are created and shared between two process for transfer messages. This library is built as a dynamic library and can be used in other generic IPC cases.

Important Terms

This section defines important acronyms, abbreviations, and terms that are required to understand this document.

Table 1. Terms and abbreviations

Term or Abbreviation	Definition
IPC	Inter-process communication
CAS	Compare and swap
SHM	Shared memory

References

This section contains technical resources used to develop libnvipc.

Table 2. References

SWE Number	Input Work Product	Revision	Location
	Non-Blocking Concurrent Queue Algorithm	*30 January 2008*	https://www.cs.rochester.edu/research/synchronization/pseudocode/queues.html

Architecture Details

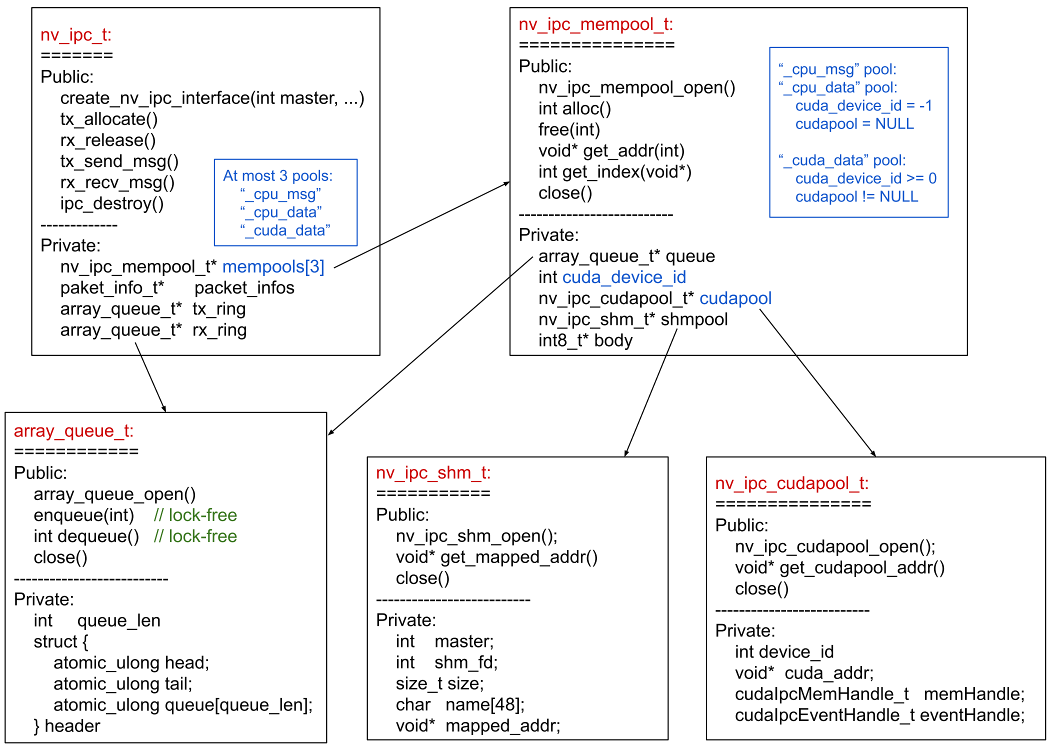
The libnvipc library is a generic Shared Memory IPC solution. Although it is implemented using C to be widely compatible, the architecture is similar to a C++ object-oriented implementation.

Shared Memory IPC

The Shared Memory IPC is used for communication between different processes on the same system. This architecture implements two features that are independent with each other:

- IPC message transfer
- Synchronization (use semaphore or event_fd)

The following diagram describes the architecture of the SHM IPC message transfer feature:



An IPC message is divided into two parts:

- MSG: Handled in the control logic, which runs in the CPU thread.
- DATA part: Handled with high performance computing, which runs in the CPU thread or GPU thread.

The `nv_ipc_msg_t` struct is defined to represent a generic IPC message. The MSG and DATA parts are stored in different buffers. The presence of the MSG part is mandandary, while the DATA part is optional. `data_buf` is null when no DATA part exists.

```
typedef struct {
    int32_t cell_id; // Cell ID
    int32_t msg_id; // IPC message ID
    int32_t msg_len; // MSG part length
    int32_t data_len; // DATA part length
    int32_t data_pool; // DATA memory pool ID
    void*   msg_buf; // MSG part buffer pointer
    void*   data_buf; // DATA part buffer pointer
} nv_ipc_msg_t;
```

The MSG buffer is allocated from the CPU shared memory pool. The DATA buffer can be allocated from the CPU shared memory pool or CUDA shared memory pool, so in total there are three types of memory pool. The `nv_ipc_mempool_id_t` enum type is defined as the memory pool indicator.

```
typedef enum {
    NV_IPC_MEMPOOL_CPU_MSG   = 0,  // CPU SHM pool for MSG part
    NV_IPC_MEMPOOL_CPU_DATA  = 1,  // CPU SHM pool for DATA part
    NV_IPC_MEMPOOL_CUDA_DATA = 2,  // CUDA SHM pool for DATA part
    NV_IPC_MEMPOOL_NUM       = 3
} nv_ipc_mempool_id_t;
```

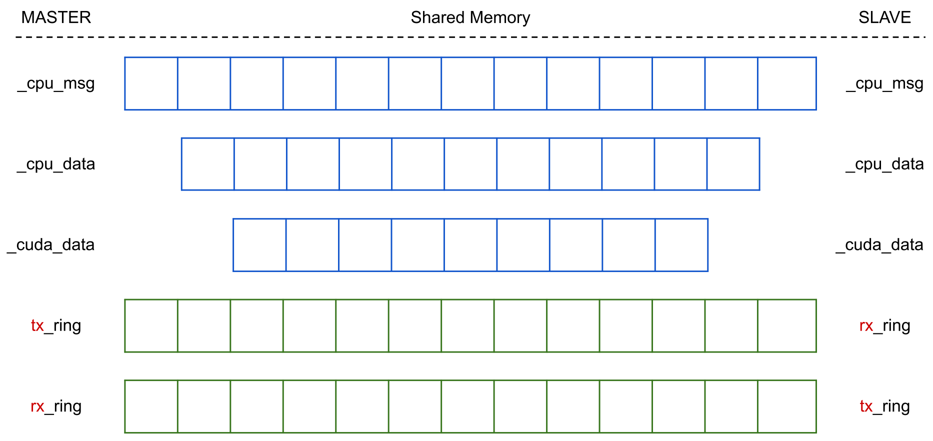
The array queue has the following features:

- FIFO (first in first out)
- Lock-free: Supports multiple producers and multiple consumers without lock.
- Finite size: The max length is defined at initial: N.
- Valid values are integers: 0, 1, …, N-1, can be used as the node index/pointer.
- Duplicate values are not supported.

Based on the lock-free array queue, generic memory pools and ring queues can be easily implemented or created, and they are lock-free as well:

- Memory pool: array queue + memory buffer array
- FIFO ring queue: array queue + queue node array

At most, three shared memory pools and two ring queues will be created per configuration:



Each memory pool is an array of fixed size buffers. The buffer size and pool length (buffer count) are configurable for each memory pool. If the buffer size or pool length is configured to be 0, that memory pool will not be created. The memory pools and ring queues are created in shared memory to be accessible between different processes.

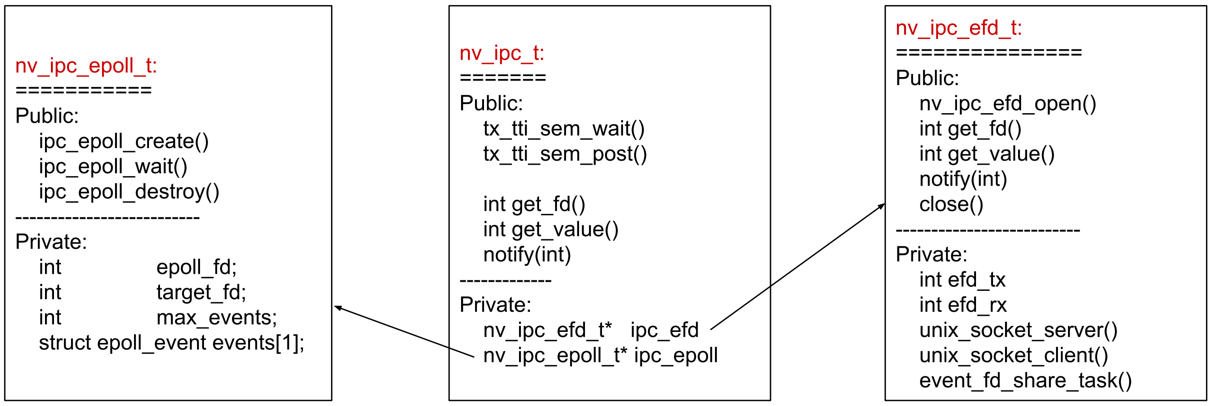
SHM name at /dev/shm/	IPC direction	Memory Pool ID
<prefix>_cpu_msg	Duplex	*NV_IPC_MEMPOOL_CPU_MSG*

<prefix>_cpu_data	Duplex	*NV_IPC_MEMPOOL_CPU_DATA*
<prefix>_cuda_data	Duplex	*NV_IPC_MEMPOOL_CUDA_DATA*

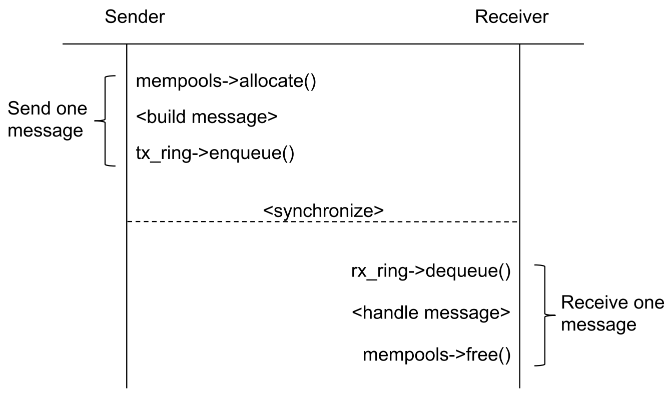
And two ring queues are created to deliver the message buffer indices. The TX ring in sender and RX ring in receiver are the same ring in SHM.

SHM name at /dev/shm/	Internal name	IPC direction	PHY/PRIMARY	MAC/SECONDARY
<prefix>_shm	<prefix>_ring_m2s	Uplink	TX	RX
<prefix>_shm	<prefix>_ring_s2m	Downlink	RX	TX

The following diagram describes the synchronization components architecture. It is based on `eventfd` to support multiple I/O with poll/epoll/select:



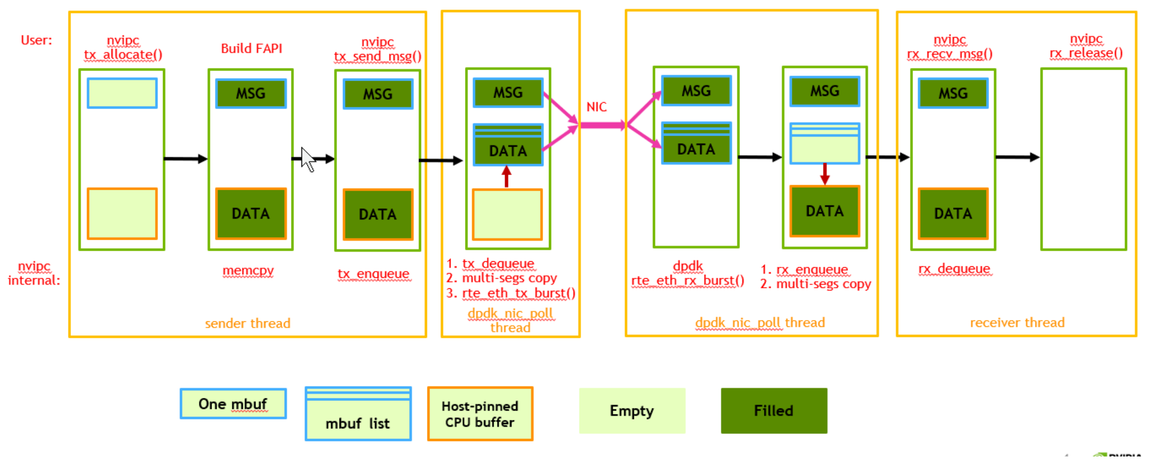
A typical IPC message transfer flow is as below:



DPDK over NIC IPC

The APIs are used for communication between processes on different systems that are connected through NIC. The APIs are compatible with the SHM IPC case; they just need to load different parameters from a YAML configuration file.

In the internal implementation of nvipc, lock-free queues are used for TX and RX. A `dpdk_nic_poll` thread is created at initialization to do the DPDK polling task on the NIC, soo it requires a dedicated CPU core on each side.
From the nvipc user perspective, the user calls `tx_send_msg()` and `rx_recv_msg()`; it is a wrapper of the `enqueue` and `dequeue` to the internal TX and RX queues. The `dpdk_nic_poll` thread runs in background to perform the actual packet transfer through DPDK over NIC. The following architecure diagram describes this process.



Design Details

Design Alternatives

- **UDP IPC**: Easy to implement and capture log by tcpdump, but has below disadvantages:
  - UDP packet size limitation is 65,507 bytes (65,535 – 8 byte UDP header – 20 byte IP header)
  - The memory copy between kernel and user space causes delay.
- **SHM IPC**: Pre-allocate CPU/GPU memory pools and share it between processes. Since no additional memory is allocated/released and memory is copied, it is most efficient.

Static Design

Configuration Data

Config yaml file (for NVIDIA cuphycontroller):

```
# Transport settings for nvIPC
transport:
  type: dpdk
  udp_config:
    local_port: 38556
    remort_port: 38555
  shm_config:
    primary: 0
    prefix: nvipc    # Note: prefix string length should < 32
    cuda_device_id: -1
    ring_len: 8192
    mempool_size:
      cpu_msg:
        buf_size: 8192
        pool_len: 4096
      cpu_data:
        buf_size: 576000
        pool_len: 1024
      cuda_data:
        buf_size: 307200
        pool_len: 0    # Set to 0 to do not create CUDA memory pool
  dpdk_config:
    primary: 0
    prefix: nvipc
    # local_nic_pci: mlx5_core.sf.2    # For run on BlueField DPU
    local_nic_pci: 0000:b6:00.0
    peer_nic_mac: 02:c0:47:39:92:fb
    nic_mtu: 1536
    cuda_device_id: -1
    need_eal_init: 1
    lcore_id: 7
    mempool_size:
      cpu_msg:
        buf_size: 8192
        pool_len: 4096
      cpu_data:
        buf_size: 576000
        pool_len: 1024
      cuda_data:
        buf_size: 307200
        pool_len: 0
  app_config:
    grpc_forward: 0
    debug_timing: 0
    pcap_enable: 0
```

Config debug with coredump:

lib/nvIPC/CMakeLists.txt

```
      set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -O0 -g")
```

External Interface and Specification

Here is the public API interface of the library:

```
struct nv_ipc_t {
    // De-initiate and destroy the nv_ipc_t instance
    int (*ipc_destroy)(nv_ipc_t* ipc);

    // Memory allocate and release
    int (*tx_allocate)(nv_ipc_t* ipc, nv_ipc_msg_t* msg, uint32_t options);
    int (*rx_release)(nv_ipc_t* ipc, nv_ipc_msg_t* msg);

    // Send and receive (enqueue in TX ring queue and dequeue in RX ring queue)
    int (*tx_send_msg)(nv_ipc_t* ipc, nv_ipc_msg_t* msg);
    int (*rx_recv_msg)(nv_ipc_t* ipc, nv_ipc_msg_t* msg);

    // Synchronization option 1: sem_post and sem_wait
    int (*tx_tti_sem_post)(nv_ipc_t* ipc);
    int (*rx_tti_sem_wait)(nv_ipc_t* ipc);

    // Synchronization option 2: get an event_fd for RX and epoll on it
    int (*get_fd)(nv_ipc_t* ipc);
    int (*notify)(nv_ipc_t* ipc, int value);
    int (*get_value)(nv_ipc_t* ipc);
    // CUDA memory copy function
    int (*cuda_memcpy_to_host)(nv_ipc_t* ipc, void* host, const void* device, size_t size);
    int (*cuda_memcpy_to_device)(nv_ipc_t* ipc, void* device, const void* host, size_t size);

    // Deprecated. rx_allocate is equal to tx_allocate, tx_release is equal to rx_release.
    int (*rx_allocate)(nv_ipc_t* ipc, nv_ipc_msg_t* msg, uint32_t options);
    int (*tx_release)(nv_ipc_t* ipc, nv_ipc_msg_t* msg);
};
int set_nv_ipc_default_config(nv_ipc_config_t* cfg, nv_ipc_module_t module_type);
int load_nv_ipc_yaml_config(nv_ipc_config_t* cfg, const char* yaml_path, nv_ipc_module_t
module_type);
nv_ipc_t* create_nv_ipc_interface(const nv_ipc_config_t* cfg);
```

Initiation

Here is the reference code for initiation. The first starting process is PRIMARY, the latter starting process is SECONDARY. PRIMARY is responsible for creating and initiating SHM pools and ring queues. SECONDARY looks up the created pools and queues:

```
    // Create configuration
    nv_ipc_config_t config;
    config.ipc_transport = NV_IPC_TRANSPORT_SHM;
    if(set_nv_ipc_default_config(&config, module_type) < 0) {
        LOGE(TAG, "%s: set configuration failed\n", __func__);
        return -1;
    }

    // Optional: Override the default configurations
    config.transport_config.shm.cuda_device_id = test_cuda_device_id;

    // Create IPC interface: nv_ipc_t ipc
    nv_ipc_t* ipc;
    if((ipc = create_nv_ipc_interface(&config)) == NULL) {
        LOGE(TAG, "%s: create IPC interface failed\n", __func__);
        return -1;
    }
```

After the IPC interface is successfully created, you can see items under `/dev/shm/ folder`. For example, if `<prefix>=”nvipc”:`

```
    ls -al /dev/shm/nvipc*
    nvipc_shm
    nvipc_cpu_msg
    nvipc_cpu_data
    nvipc_cuda_data
    nvipc.log
```

De-initiation

```
    if(ipc->ipc_destroy(ipc) < 0) {
        LOGE(TAG, "%s close IPC interface failed\n", __func__);
    }
```

Send

The procedure for sending is as follows
:

allocate buffers –> fill content –> send

When fill content, for CUDA memory, the data_buf is a CUDA memory pointer which can't be accessed directly in CPU memory space. The IPC API interface provide basic memcpy functions to copy between CPU memory and CUDA memory. For more CUDA operation, user can directly access the GPU memory buffer with CUDA APIs.

```
nv_ipc_msg_t send_msg,
send_msg.msg_id    = fapi_msg_id; // Optional: FAPI message ID
send_msg.msg_len   = fapi_msg_len; // Max length is the MSG buffer size, configurable
send_msg.data_len  = fapi_data_len; // Max length is the MSG buffer size, configurable
send_msg.data_pool = NV_IPC_MEMPOOL_CPU_DATA; // Options: CPU_MSG, CPU_DATA, CUDA_DATA

// Allocate buffer for TX message
if(ipc->tx_allocate(ipc, &send_msg, 0) != 0)
{
    LOGE(TAG, "%s error: allocate buffer failed\n", __func__);
    return -1;
}

// Fill the MSG content
int8_t fapi_msg[SHM_MSG_BUF_SIZE];
memcpy(send_msg.msg_buf, fapi_msg, fapi_len);

// Fill the DATA content if data exist.
int8_t fapi_data[SHM_MSG_DATA_SIZE];
if (send_msg.data_pool == NV_IPC_MEMPOOL_CPU_DATA) { // CPU_DATA case
    memcpy(send_msg.data_buf, fapi_data, send_msg.data_len);
} else if (send_msg.data_pool == NV_IPC_MEMPOOL_CUDA_DATA) { // CUDA_DATA case
    if(ipc->cuda_memcpy_to_device(ipc, send_msg.data_buf, fapi_data, send_msg.data_len) < 0){
        LOGE(TAG, "%s CUDA copy failed\n", __func__);
    }
} else { // NO_DATA case
    // NO data, do nothing
}

// Send the message
if(ipc->tx_send_msg(ipc, &send_msg) < 0){
    LOGE(TAG, "%s error: send message failed\n", __func__);
    // May need future retry or release the send_msg buffers
    // If fail, check configuration: ring queue length > memory pool length
}
```

Receive

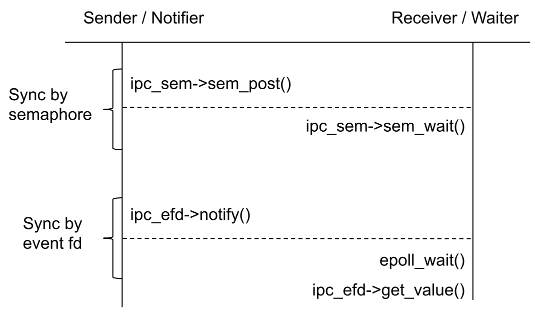
The procedure for receiving is as follows:

receive –> handle message –> release buffers

```
nv_ipc_msg_t recv_msg
if(ipc->rx_recv_msg(ipc, &recv_msg) < 0)
{
    LOGV(TAG, "%s: no more message available\n", __func__);
    return -1;
}

// Example: Handle MSG part
int8_t fapi_msg[SHM_MSG_BUF_SIZE];
memcpy(fapi_msg, recv_msg.msg_buf, recv_msg.msg_len);

// Example: Handle DATA part
int8_t fapi_data[SHM_MSG_BUF_SIZE];
if (recv_msg.data_pool == NV_IPC_MEMPOOL_CPU_DATA) { // CPU_DATA case
    memcpy(fapi_data, recv_msg.data_buf, &recv_msg.data_len);
} else if (recv_msg.data_pool == NV_IPC_MEMPOOL_CUDA_DATA) { // CUDA_DATA case
    if(ipc->cuda_memcpy_to_host(ipc, fapi_data, recv_msg.data_buf, recv_msg.data_len) < 0){
        LOGE(TAG, "%s CUDA copy failed\n", __func__);
    }
} else { // NO_DATA case
    // NO data, do nothing
}

if(ipc->rx_release(ipc, &recv_msg) < 0){
    LOGW(TAG, "%s: release error\n", __func__);

}
```

Synchronization

Since above memory pools and ring queues support lock-less concurrence, the use of synchronization APIs is not mandandary.
Two tyles of synchronization APIs are provided: semaphore style and event_fd style. Each side can choose any tyles no matter what the other side chooses, but keep using one tyles in one side.



In low level of the SHM IPC library event_fd is implemented. The semaphore API interface is a wapper of the event_fd implementation.
The APIs are ready to use after IPC interface successfully created by create_nv_ipc_interface().

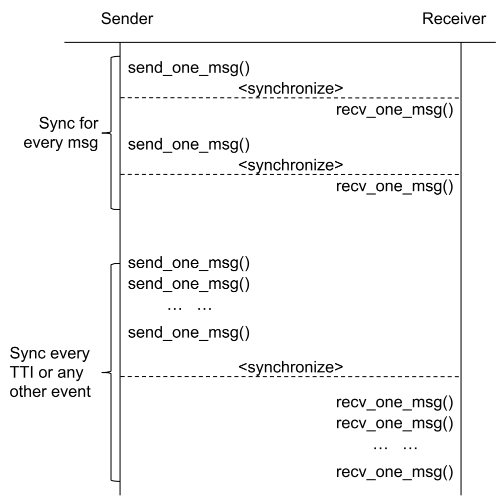
For semaphore tyles, it's easy to use:

```
Receiver:
    ipc->tx_tti_sem_wait(ipc);

Sender:
    ipc->tx_tti_sem_post(ipc);
```

For event_fd style, user can get the the fd and use epoll functions to monitor multiple I/O:

```
Receiver:
    struct epoll_event ev, events[MAX_EVENTS];

    int epoll_fd = epoll_create1(0);
    if(epoll_fd == -1)
    {
        LOGE(TAG, "%s epoll_create failed\n", __func__);
    }

    int ipc_rx_event_fd = ipc->get_fd(ipc);  // IPC synchronization API: get_fd()
    ev.events           = EPOLLIN;
    ev.data.fd          = ipc_rx_event_fd;
    if(epoll_ctl(epoll_fd, EPOLL_CTL_ADD, ev.data.fd, &ev) == -1)
    {
        LOGE(TAG, "%s epoll_ctl failed\n", __func__);
    }

    while(1)
    {
        int nfds = epoll_wait(epoll_fd, events, MAX_EVENTS, -1);
        if(nfds == -1)
        {
            LOGE(TAG, "epoll_wait notified: nfds=%d\n", nfds);
        }

        for(int n = 0; n < nfds; ++n)
        {
            if(events[n].data.fd == ipc_rx_event_fd)
            {
                ipc->get_value(ipc);  // IPC synchronization API: get_value()
                // Receive incoming message here
            }
        }
    }
    close(epoll_fd);

Sender:
    ipc->notify(ipc, 1);  // IPC synchronization API: notify()
```

Synchronization can be called one time for each message or one time for a whole TTI per user's requirement.



Dependencies

- GCC version >= 4.9.2 (Other compilers were not verified)
- CUnit library (For internal test only).

Team or Company	Deliverable	Ref	Commit
Open Source	No	Open source: https://gitlab.com/cunity/cunit IP Audit: Bug 200578584	

Integration Validation Plan

Functionality to Validate	Teams Participating	Interfaces Covered	Date Complete
Use CUnit to test nv_ipc_t		All the APIs in nv_ipc_t	Mar 6, 2020

Dynamic Design

Lock-free array queue

The below document and pseudo code describes the theory of a typical lock-free queue. The document was written by Maged M. Michael and Michael L. Scott.
https://www.cs.rochester.edu/u/scott/papers/1996_PODC_queues.pdf
https://www.cs.rochester.edu/research/synchronization/pseudocode/queues.html

Disadvantage: In the above pseudo code, there's memory allocation in ENQUEUE operation. The node allocation may cause performance decrease and unknown delay:

```
E1:   node = new_node()      // Allocate a new node to enqueue a node data
```

Improvement solution: Improve the allocation by using a pre-allocate fixed size node array. Assume size = N, the queue supports enqueuing integer value in range 0, 1, …, N-1. When enqueue, allocate the array element whose index is equals to the enqueuing value. The allocation will always succeed and doesn't cost memory allocation time:

```
E1:   node = array[value]      // Allocate array[value] to enqueue an integer value
```

Another difference with the original queue design is that there's one node in queue (the initial node) when the queue is empty, but in this implementation, there's no node in the empty queue. This change is necessary because of the new node allocation mechanism.

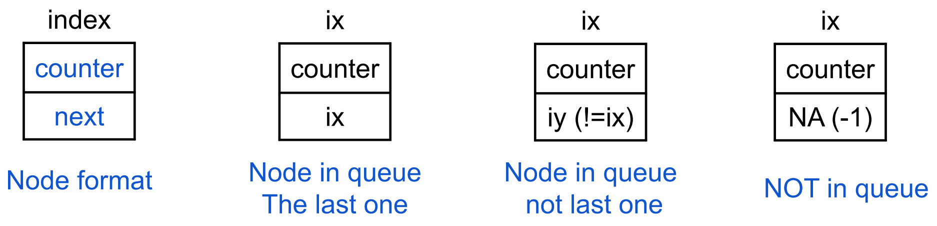
Below is the queue data structure:

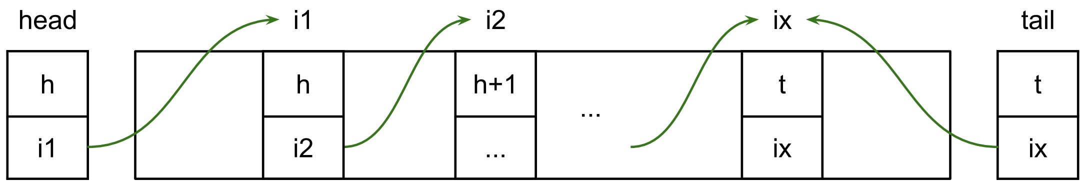
```
typedef struct {
    int32_t  next;
    uint32_t counter;
} note_t;

typedef union {
    note_t   node;
    uint64_t ulong;
} cas_union_t;

typedef struct {
    atomic_ulong head;      // Point to the latest enqueued node.
    atomic_ulong tail;      // Point to the earliest enqueued node.
    atomic_ulong queue[];   // Queue array data
} array_queue_header_t;
```

And the queue APIs:

```
struct array_queue_t
{
    int32_t (*get_length)(array_queue_t* queue);

    int (*enqueue)(array_queue_t* queue, int32_t value);

    int32_t (*dequeue)(array_queue_t* queue);

    int (*close)(array_queue_t* queue);
};

// The memory size pre-allocated for the queue structure
#define ARRAY_QUEUE_HEADER_SIZE(queue_len) (align_8(sizeof(array_queue_header_t) +
sizeof(atomic_ulong) * (queue_len)))

// header size should be calculated by ARRAY_QUEUE_MEMORY_SIZE(length)
array_queue_t* array_queue_open(int primary, const char* name, void* header, int32_t length);
```

Here is the node structure and the node array. Each node contains a "counter" and "next" fields. The node value is equal to its array index, so there's no need to define it in the structure.

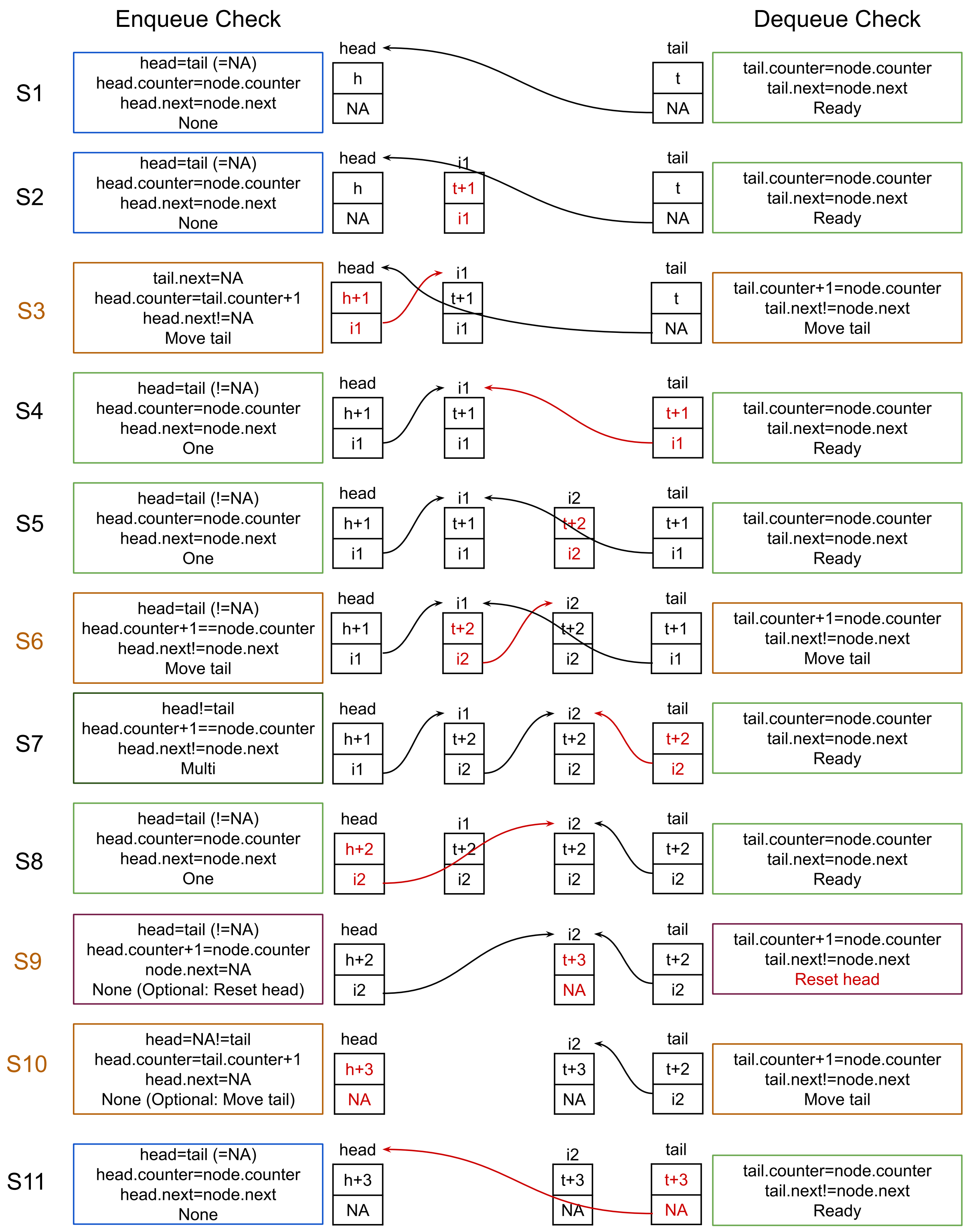




Counter is used to avoid ABA problem. The atomic functions (CAS) operate on uint64_t type data, they change "counter" and "next" together.
Below diagram demonstrates all possible state transition flows of the array queue:

Empty (initial) --> One Node --> Multi Nodes --> One Node --> Empty

Of all the states, there are 3 final states: Empty (S1, S2, S11), One Node in Queue (S4, S5, S8), Multi Nodes in Queue (S7). Other states (S3, S6, S9, S10) are temporary states. At final state it's ready to go ahead with enqueue/dequeue CAS. At temporary states the head or the tail need to be moved to final states first.
We can see that for a generic node (i = 0, 1, 2, …, N-1), its value equals to its array index: value = index = i. The queue head and tail have the same structure with generic nodes, and head can be treated as a special node: value = index = -1 (NA).
The value changed in last step is marked red.


Enqueue Check
Dequeue Check
S1
head=tail (=NA)
head.counter=node.counter
head.next=node.next
None
head
h
NA
tail
t
NA
tail.counter=node.counter
tail.next=node.next
Ready
S2
head=tail (=NA)
head.counter=node.counter
head.next=node.next
None
head
h
NA
i1
t+1
i1
tail
t
NA
tail.counter=node.counter
tail.next=node.next
Ready
S3
tail.next=NA
head.counter=tail.counter+1
head.next!=NA
Move tail
head
h+1
i1
i1
t+1
i1
tail
t
NA
tail.counter+1=node.counter
tail.next!=node.next
Move tail
S4
head=tail (!=NA)
head.counter=node.counter
head.next=node.next
One
head
h+1
i1
i1
t+1
i1
tail
t+1
i1
tail.counter=node.counter
tail.next=node.next
Ready
S5
head=tail (!=NA)
head.counter=node.counter
head.next=node.next
One
head
h+1
i1
i1
t+1
i1
i2
t+2
i2
tail
t+1
i1
tail.counter=node.counter
tail.next=node.next
Ready
S6
head=tail (!=NA)
head.counter+1==node.counter
head.next!=node.next
Move tail
head
h+1
i1
i1
t+2
i2
i2
t+2
i2
tail
t+1
i1
tail.counter+1=node.counter
tail.next!=node.next
Move tail
S7
head!=tail
head.counter+1==node.counter
head.next!=node.next
Multi
head
h+1
i1
i1
t+2
i2
i2
t+2
i2
tail
t+2
i2
tail.counter=node.counter
tail.next=node.next
Ready
S8
head=tail (!=NA)
head.counter=node.counter
head.next=node.next
One
head
h+2
i2
i1
t+2
i2
i2
t+2
i2
tail
t+2
i2
tail.counter=node.counter
tail.next=node.next
Ready
S9
head=tail (!=NA)
head.counter+1=node.counter
node.next=NA
None (Optional: Reset head)
head
h+2
i2
i2
t+3
NA
tail
t+2
i2
tail.counter+1=node.counter
tail.next!=node.next
Reset head
S10
head=NA!=tail
head.counter=tail.counter+1
head.next=NA
None (Optional: Move tail)
head
h+3
NA
i2
t+3
NA
tail
t+2
i2
tail.counter+1=node.counter
tail.next!=node.next
Move tail
S11
head=tail (=NA)
head.counter=node.counter
head.next=node.next
None
head
h+3
NA
i2
t+3
NA
tail
t+3
NA
tail.counter=node.counter
tail.next!=node.next
Ready

Control Flow

Memory pool

```
struct nv_ipc_mempool_t
{
    int32_t (*alloc)(nv_ipc_mempool_t* mempool);

    int (*free)(nv_ipc_mempool_t* mempool, int32_t index);

    int (*get_index)(nv_ipc_mempool_t* mempool, void* buf);

    void* (*get_addr)(nv_ipc_mempool_t* mempool, int32_t index);

    int (*get_buf_size)(nv_ipc_mempool_t* mempool);

    int (*get_pool_len)(nv_ipc_mempool_t* mempool);

    int (*memcpy_to_host)(nv_ipc_mempool_t* mempool, void* host, const void* device, size_t size);

    int (*memcpy_to_device)(nv_ipc_mempool_t* mempool, void* device, const void* host, size_t
size);

    int (*close)(nv_ipc_mempool_t* mempool);
};

nv_ipc_mempool_t* nv_ipc_mempool_open(int primary, const char* name, int buf_size, int pool_len,
int cuda_device_id);
```

With the use of the array_queue, it's easy to implement the memory pool manager (buffer allocate/release). Here is pseudo code for buffer allocate and free:

```
Initiate:
    array_queue_t* queue = array_queue_open(length);
    for(int i = 0; i < length; i++) {
        queue->enqueue(i);
    }

Allocate:
    int index = queue->dequeue();
    void* buf = queue->get_addr(index);
    return buf;

Free buf:
    int index = queue->get_index(buf);
    queue->enqueue(index);
```

FIFO Ring (for send and receive message)

FIFO rings are embedded in nv_ipc_shm_if.c. Below is the pseudo code:

```
Initiate:
    array_queue_t* tx_ring = array_queue_open(tx_name, length);
    array_queue_t* rx_ring = array_queue_open(rx_name, length);

Send:
    // Allocate memory buffer and fill with message first
    tx_ring->enqueue(msg_index);

Receive:
    int msg_index = rx_ring->dequeue();
    // Get buffer from memory pool by msg_index, get the message and free the buffer
```

Error Handling

All the APIs return negative integer value (-1) or NULL pointer when fails or queue is empty; return 0, positive integer or valid pointer when success. User can check the return value to get whether the function call was successful.
In all error cases, the error reason will be printed with ERROR level log. User can check /dev/shm/nvipc.log or /tmp/nvipc.log to get the fail reason.

Logging and Debugging

nvlog

A high performance, ordered, lock-free logger is implemented in nvlog.c.
Firstly, the logs are cached to SHM memory /dev/shm/phy.log. The SHM log file has fixed size and it is configured to 4MB by default. Can be changed at:

```
#define SHM_SIZE_BIT_LEN 22 // 4MB
#define LOG_SHM_SIZE (1 << SHM_SIZE_BIT_LEN)
```

If too many logs exceed the SHM file size, then the /dev/shm/nvipc.log file will be over written. A background thread is waiting to save half of the SHM cache to /tmp/nvipc.log every time when available.
Each log line will be added with logging time, module type (primary/secondary), log level and an increasing counter.
By default, the statistic info printed every 1000 times of allocate/send/recv/release. Can change log level to LOG_DEBUG to print the msg_id, msg_len, data_len, data_pool of each FAPI message.
Hard code the DEFAULT_LOG_LEVEL value and recompile libnvipc.so or call nv_ipc_log_set_level(LOG_DEBUG) in the user program.

```
#define DEFAULT_LOG_LEVEL LOG_INFO // 3
#define DEFAULT_LOG_LEVEL LOG_DEBUG // 4
void nv_ipc_log_set_level(int level);
```

```
2020-03-17 04:46:17.499906 M I 28850 QUEUE: nvipc_cpu_msg: enqueue try_max=1 counter: enq=12784001 deq=12779906 available~4095
2020-03-17 04:46:17.512319 M I 28851 QUEUE: nvipc_cpu_msg: dequeue try_max=1 counter: enq=12784096 deq=12780000 available=4095
2020-03-17 04:46:17.672086 S I 28852 QUEUE: nvipc_ring_s2m: enqueue try_max=2 counter: enq=7340001 deq=7340000 available~1
2020-03-17 04:46:17.672117 M I 28853 QUEUE: nvipc_ring_s2m: dequeue try_max=1 counter: enq=7340002 deq=7340000 available=1
2020-03-17 04:46:17.754056 M I 28854 QUEUE: nvipc_cpu_msg: enqueue try_max=1 counter: enq=12786001 deq=12781905 available~4096
2020-03-17 04:46:17.766276 M I 28855 QUEUE: nvipc_cpu_msg: dequeue try_max=1 counter: enq=12786095 deq=12782000 available=4094
2020-03-17 04:46:17.797613 S I 28856 QUEUE: nvipc_cpu_data: dequeue try_max=1 counter: enq=3251024 deq=3250000 available=1023
2020-03-17 04:46:17.933682 M I 28857 QUEUE: nvipc_ring_m2s: enqueue try_max=3 counter: enq=5442001 deq=5441999 available~2
2020-03-17 04:46:17.933685 S I 28858 QUEUE: nvipc_ring_m2s: dequeue try_max=3 counter: enq=5442001 deq=5442000 available=0
```

Enable core dump

```
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -O0 -g")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -O0 -g")
ulimit -c unlimited
```

For CentOS docker container running on Ubuntu host case, need to run below additional step on host:

```
echo "core.%p" | sudo tee /proc/sys/kernel/core_pattern
```

State Machine

See Lock-free Array Queue section above.

Test Automation)

Note: Test automation is mandatory only for GPU Driver features, optional otherwise

Unit test program: `tests/cunit/nvipc_cunit`. It covers below cases:

(1) Run single process or fork to 2 processes: PRIMARY and SECONDARY.
(2) test_ipc_open: Create IPC inteface in PRIMARY or SECONDARY.
(3) test_assign_cpu: Assign CPU core for process: bind PRIMARY to CPU core 0, bind SECONDARY to CPU core 2.
(4) test_ring_single_thread: Create one thread in PRIMARY running dequeue in PRIMARY; Create one thread in SECONDARY running enqueue.
(5) test_ring_multi_thread: Create multi-threads in PRIMARY running dequeue; Create multi-threads in SECONDARY running enqueue
(6) test_cpu_mempool: Create one thread both in PRIMARY and SECONDARY running alloc and free; test alloc all and check full, then free all.
(7) test_blocking_transfer: Test single thread blocking wait transferring between PRIMARY and SECONDARY (sync by event_fd).
(8) test_epoll_transfer: Test single thread epoll transferring between PRIMARY and SECONDARY (sync by event_fd).
(9) test_no_sync_transfer: Test single thread transferring without synchronization (infinite loop polling) between PRIMARY and SECONDARY.
(10) test_transfer_multi_thread: Test multi-thread transferring without synchronization (infinite loop polling) between PRIMARY and SECONDARY.
(11) test_ipc_close: Close/destroy the IPC interface, all test finished.

High Availability

(1) Sending-Receiving delay

According to Test (8) the average delay is about 1.6us (above log shows min=0.34us, avg=1.57us max=48.6us).

According to Test (9) the average delay is about 80us (above log shows min=0.36us, avg=80us max=294us).

(2) Bandwidth, packet count per second

According to Test (7) (8) the speed is about 673847 packets per second.
According to Test (9), the speed is about 2278611 packets per second.
The bandwidth depends on packet size:
Packet size: 1KB ---- Bandwidths: 0.67GB/s, 2.3GB/s.
Packet size: 10KB ---- Bandwidths: 6.7GB/s, 23GB/s.
Packet size: 100KB ---- Bandwidths: 67GB/s, 230GB/s.