Luis Renner
Institute of Systems Architecture, Chair of Operating Systems

Device Driver Support for RDMA Networks on L4Re

Research Project // Dresden, 26th August 2024

Contents

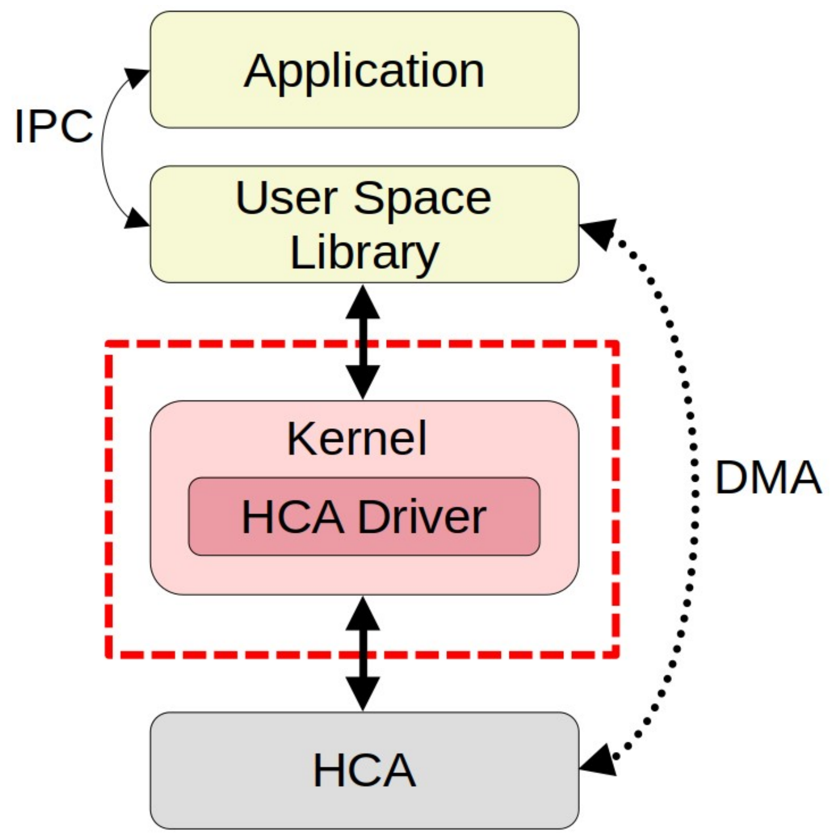
I. Introduction

Motivation

- RDMA used for low latency, high bandwidth networking
- special hardware and driver support required for RDMA
- L4Re can be used in large distributed systems e.g. as a Hypervisor
- currently L4Re lacks native driver support for any RDMA hardware
- l4linux kernel on top of L4 can run(?) Linux drivers but only as proof of concept

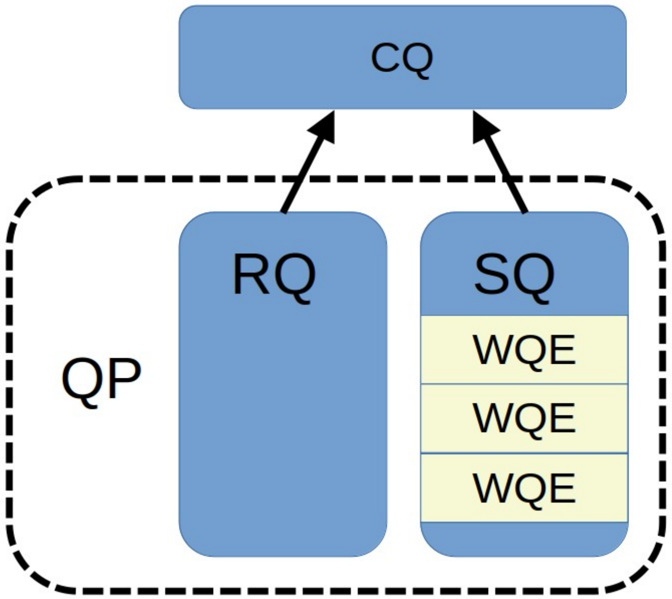
Remote Direct Memory Access (RDMA)

- Direct Memory Access (DMA)
  - hardware can write directly to main memory
  - has to be setup by kernel driver using syscalls
- Remote Direct Memory Access (RDMA)
  - special NICs called Host Channel Adapters (HCA)
  - allow for DMA over the network



Queue Pairs

- network connection abstraction through Queue Pairs (QP)
  - Send Queue (SQ) with Work Queue Entries (WQE) to send data
  - Receive Queue (RQ) with Work Queue Entries that specify where the received data should stored in memory
  - SQ and RQ report all completed WQE to a Completion Queue (CQ)

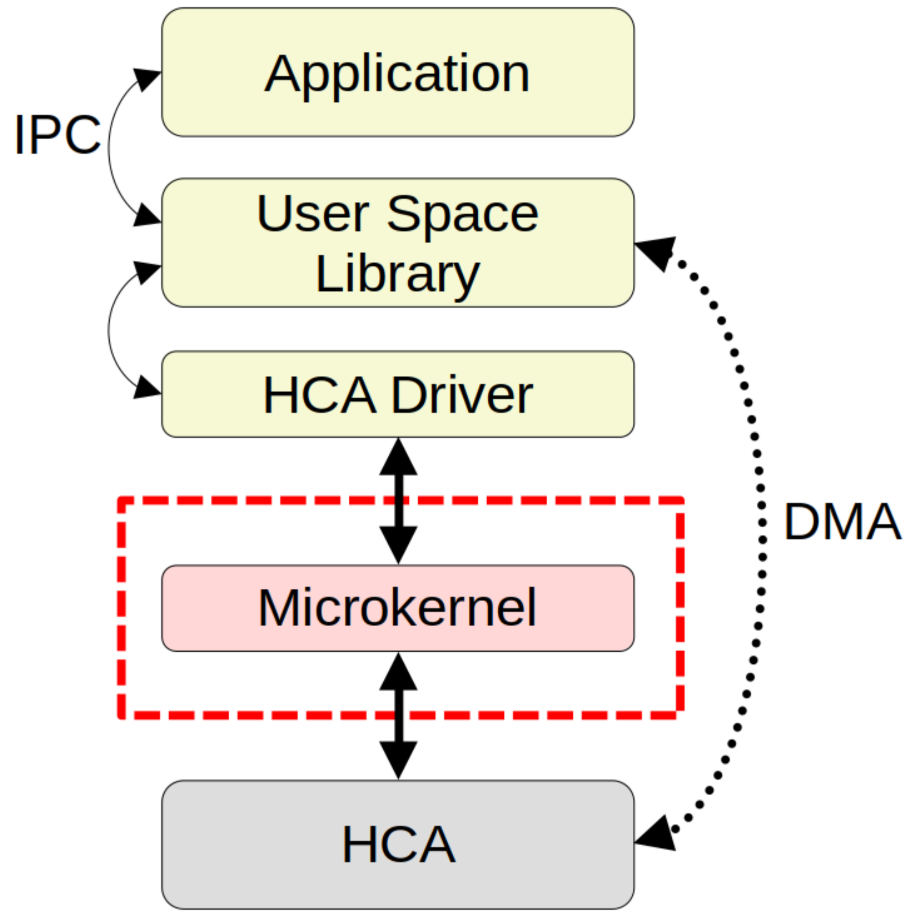


L4 Runtime Environment (L4Re)

- Microkernel operating system
  - provides memory isolation, IPC, resource mapping and threads
  - does not enforce policies (aside from scheduling)
- no syscalls like with monolithic kernels
  - reference “pointers“ to kernel objects called capabilities
  - kernel object members function as syscalls
  - capabilities are separate for all tasks

RDMA in L4Re

- HCA driver is now part of user space
- user space library talks to driver through normal IPC instead of syscalls
- driver talks to HCA through L4Re capabilities
- driver provides HCA DMA memory to user space library



Project Setup

- Mellanox Technologies (NVIDIA) ConnectX-6 Dx HCA (RoCE)
    - uses ConnectX-4 architecture (upwards compatibility)
    - User Space Library → libmlx5
    - HCA Driver → mlx5_core
- mlx5_core needs to be reimplemented using L4Re capabilities (too complicated to port)
- libmlx5 needs to be ported to use IPC instead of syscalls to talk to the driver
- Too much for one person therefore split in two!

Goals

- implement basic mlx5_core functions
  - research HCA interfaces
  - initialize and teardown HCA
  - setup Work Queues
- implement L4Re IPC
  - research IPC in L4Re
  - provide basic IPC server with test client
  - design & implement interface between libmlx5 and driver
- Ping Pong Demo!

II. ConnectX-4 Architecture & Implementation

1. L4Re Devices and DMA Spaces
2. HCA Interfaces
3. HCA Initialization and Teardown
4. Events and Interrupts
5. Work Queues and Data Flow
6. L4Re IPC Interface

L4Re Devices and DMA Spaces

server/src/device.*
server/src/mem.*

- Device can be accessed through L4vbus capability
- vbus is provided by IO service (conf/l4rdma.io)
- create DMA Space capability using Factory
- assign DMA domain to Device

- DMA Spaces can be used to allocate DMA memory chunks
- DMA memory needs to be pinned (Device uses physical addresses)

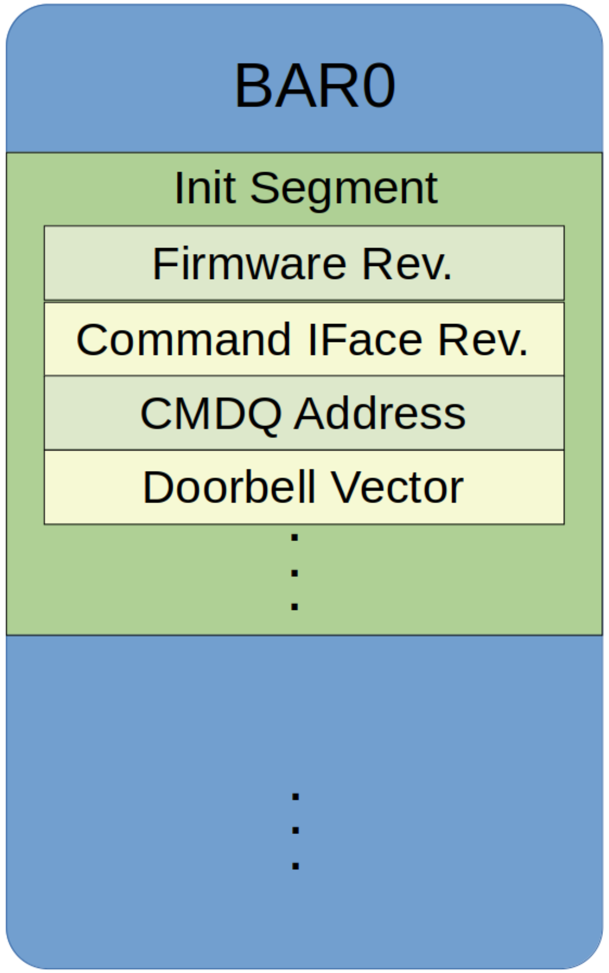
HCA Interfaces

1) PCI Interface
2) ConnectX-4 Command Interface
3) User Access Region (UAR) of HCA

PCI Interface

server/src/device.*

- access to PCI IO memory through Device capability
- IO memory is little endian
- map BAR0 into Dataspace
- start of BAR0 is InitSeg
- BAR0 is big endian



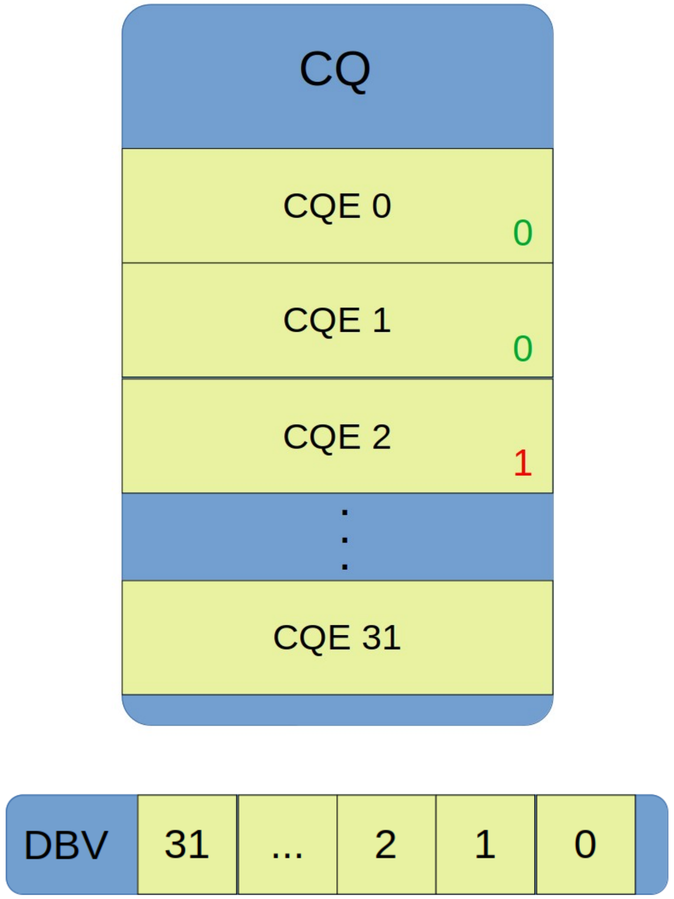
ConnectX-4 Command Interface

- HCA functions wrapped into commands
- like WQE in WQ commands are given as Command Queue Entries (CQE) in a Command Queue (CQ)
- each command has an Operation Code (opcode) to indicate the command and an Operation Modifier (opmod) if one command has multiple functions
- specific input and output data is different for every command

Implementation

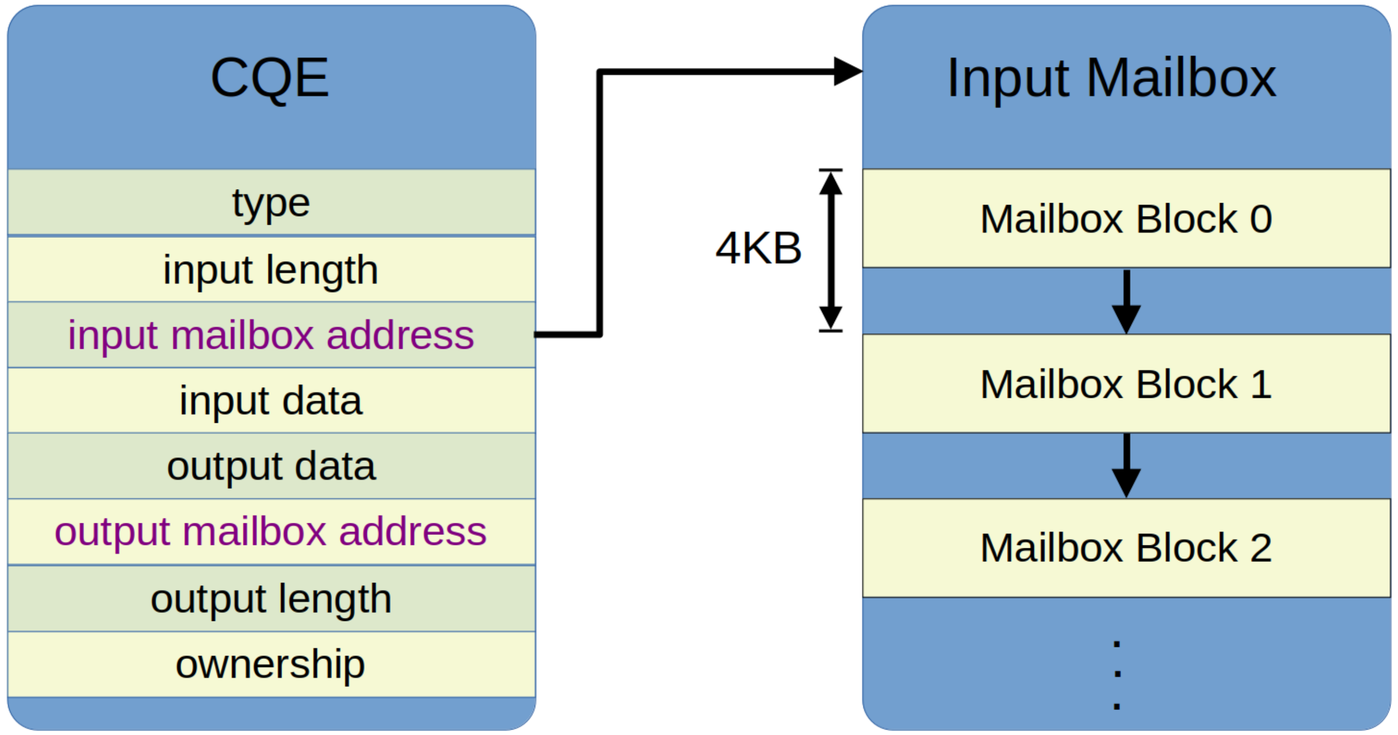
server/src/cmd.*

- write address of CQ DMA memory to InitSeg
- last bit of CQE is the ownership bit
  - 1 → hardware owned (set by SW)
  - 0 → software owned (set by HW)
  - poll to see if HW has completed CQE
- set bit in Doorbell Vector (DBV) in InitSeg to notify HW of CQE (32bit long)



Mailboxes

- only the first 8Byte of input and output are part of CQE
- overflow goes into mailboxes



Executing QUERY_HCA_CAP

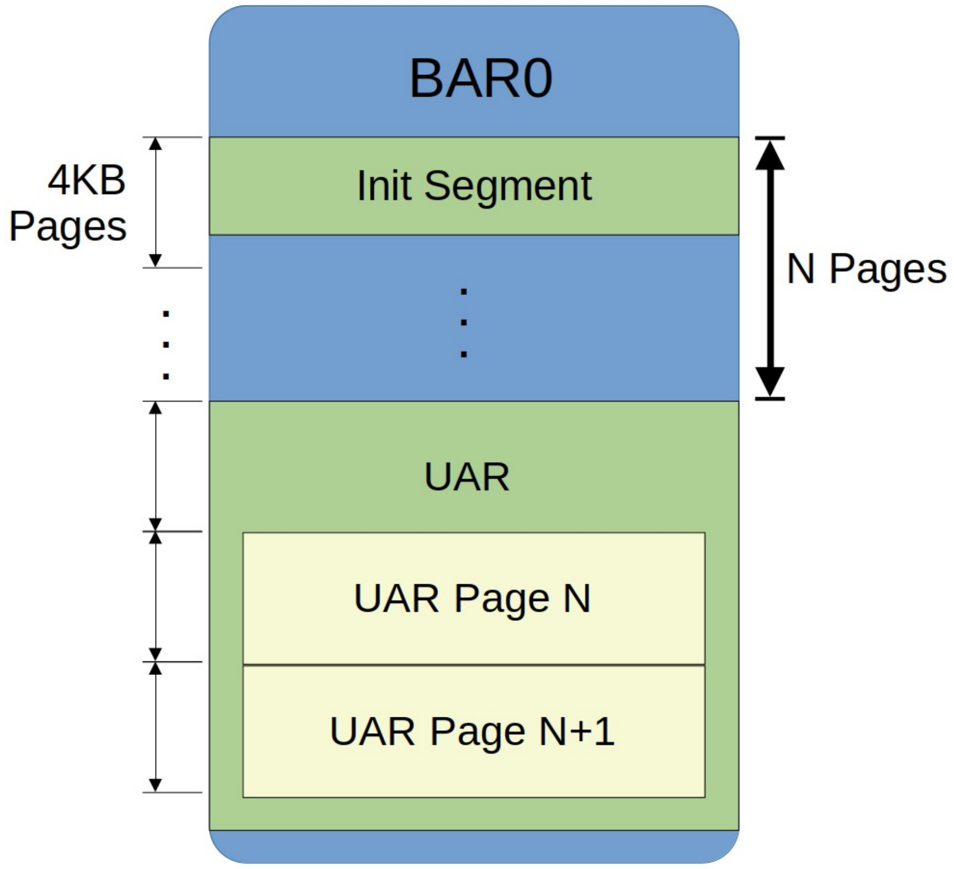
server/src/driver.*

```
l4_uint32_t slot;
l4_uint32_t payload[1] = {0};

slot = create_cqe(ctx, QUERY_HCA_CAP, 0x0, &payload, 1, QUERY_HCA_CAP_OUTPUT_LENGTH);
ring_doorbell(ctx.dbv, &slot, 1);
validate_cqe(ctx.cq, &slot, 1);

l4_uint32_t hca_cap[QUERY_HCA_CAP_OUTPUT_LENGTH];
get_cmd_output(ctx, slot, hca_cap, QUERY_HCA_CAP_OUTPUT_LENGTH);
```

User Access Region (UAR)

server/src/uar.*

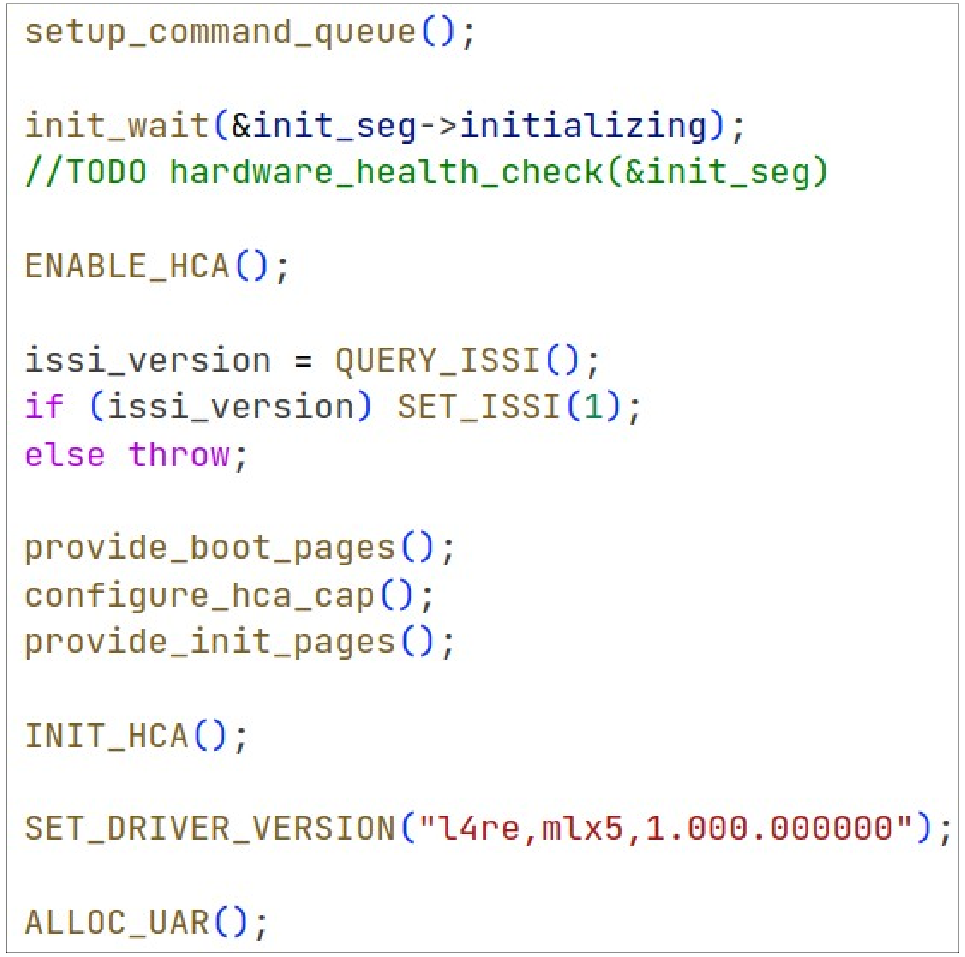
- region of BAR0 that can be mapped to the user space library to control WQs on HCA
- consists of formatted pages (UARP)
- UAR starts somewhere far below InitSeg
- ALLOC_UAR command is only executed once, management is left to the driver



HCA Initialization

server/src/driver.*

- ENABLE_HCA must be the first command
- Interface Step Sequence ID (ISSI)
  - starts at ISSI = 0
  - this driver is developed for ISSI = 1
- driver provides HCA with DMA memory pages for control structures



```
setup_command_queue();

init_wait(&init_seg->initializing);
//TODO hardware_health_check(&init_seg)

ENABLE_HCA();

issi_version = QUERY_ISSI();
if (issi_version) SET_ISSI(1);
else throw;

provide_boot_pages();
configure_hca_cap();
provide_init_pages();

INIT_HCA();

SET_DRIVER_VERSION("l4re,mlx5,1.000.000000");

ALLOC_UAR();
```

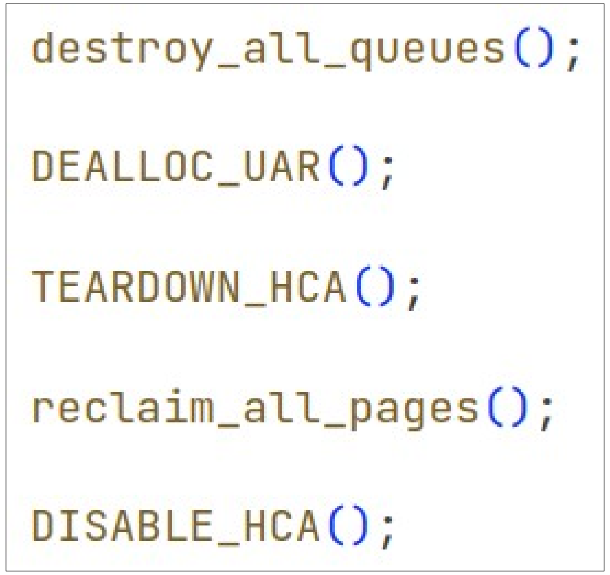
Provide Pages

- execute QUERY_PAGES command
  - ↳ opmod 0x1 for boot pages
  - ↳ opmod 0x2 for init pages
- execute MANAGE_PAGES in a loop until all pages are provided
- pages are represented by their physical addresses in the command payload

HCA Teardown

server/src/driver.*

- first destroy all queues and all other structures that were created or allocated on the HCA (except the CQ)
- TEARDOWN_HCA make sure HCA no longer uses provided DMA pages
- reclaim pages by running MANAGE_PAGES in a loop
- DISABLE_HCA is always the last command executed

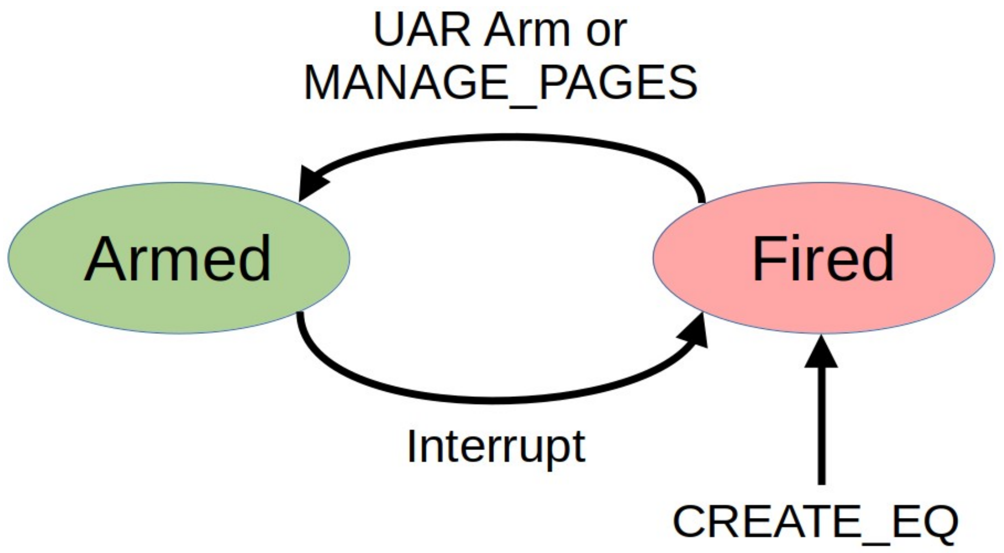


```
destroy_all_queues();

DEALLOC_UAR();

TEARDOWN_HCA();

reclaim_all_pages();

DISABLE_HCA();
```

Events and Interrupts

server/src/event.*

- HCA handles interrupts, errors, page requests and other asynchronous events through Event Queues (EQ)
- EQs can be polled like the CQ
- events can also trigger interrupts
- EQ has an interrupt state machine
- EQ needs to be armed to fire interrupts



Message Interrupts (MSI)

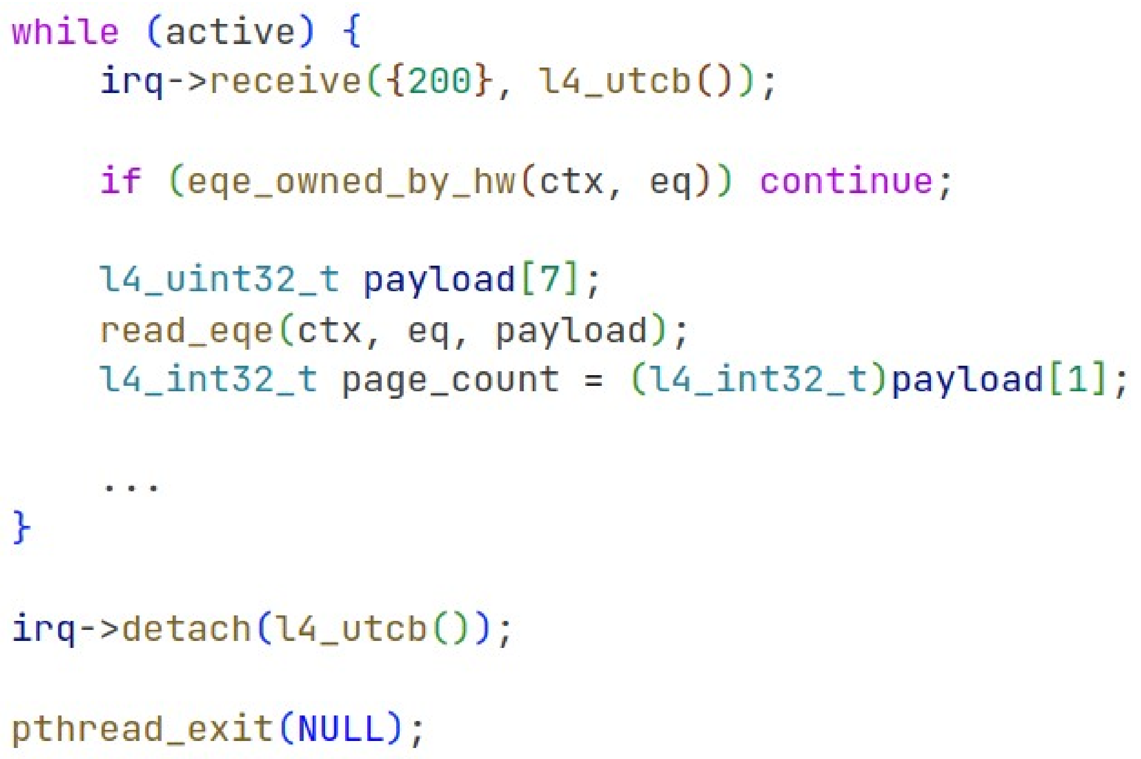
server/src/device.*
server/src/interrupt.*

- interrupts generated by EQs are PCI message interrupts
- in L4Re MSIs are available through the Interrupt Controller Unit (ICU) capability
- the MSI can be bound to an IRQ object using the ICU
- every IRQ object is bound to a thread (this thread can then wait for an interrupt with IRQ→receive())
- the MSI info from the ICU also needs to be entered into the MSI table in the PCI memory space (number of the table entry is given at CREATE_EQ)

Page Request Handling

server/src/driver.*

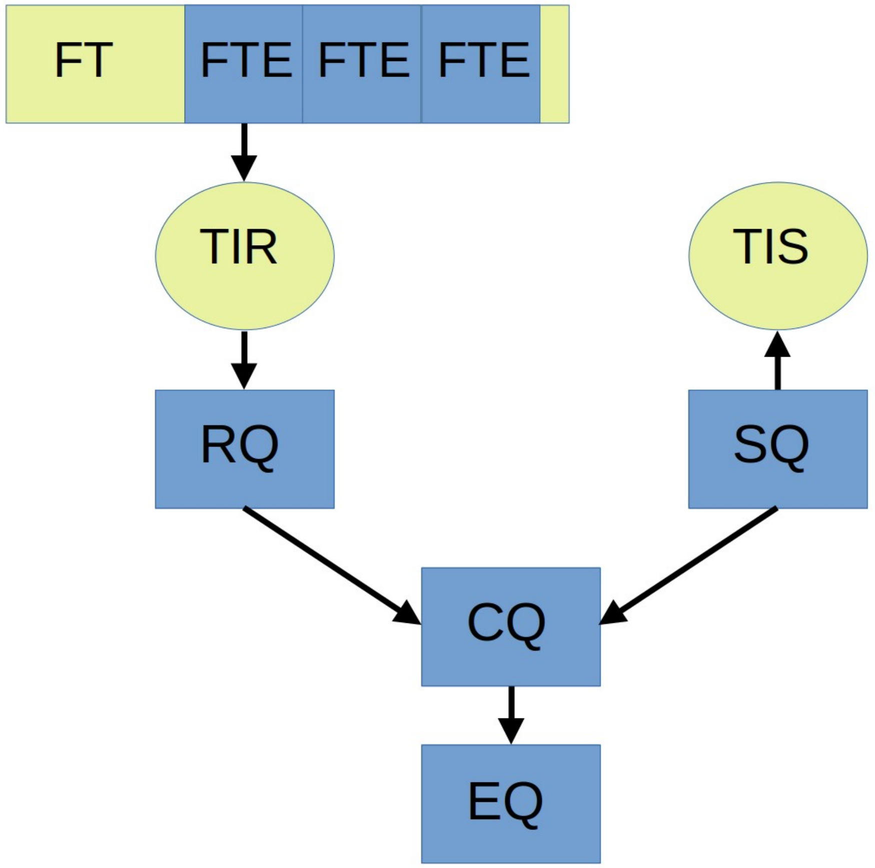
- HCA asks for more memory pages or wants to return them
- handled by the driver
- needs to be handled for continues operation
- create separate handler thread



```
while (active) {
    irq->receive({200}, l4_utcb());

    if (eqe_owned_by_hw(ctx, eq)) continue;

    l4_uint32_t payload[7];
    read_eqe(ctx, eq, payload);
    l4_int32_t page_count = (l4_int32_t)payload[1];

    ...
}

irq->detach(l4_utcb());

pthread_exit(NULL);
```

Work Queues and Data Flow

- incoming packets go through Flow Table (FT)
- Transport Interface Receive (TIR) represents and handles one incoming packet flow
- Transport Interface Send (TIS) represents and handles one outgoing packet flow
- SQ and RQ report to Completion Queue (CQ)

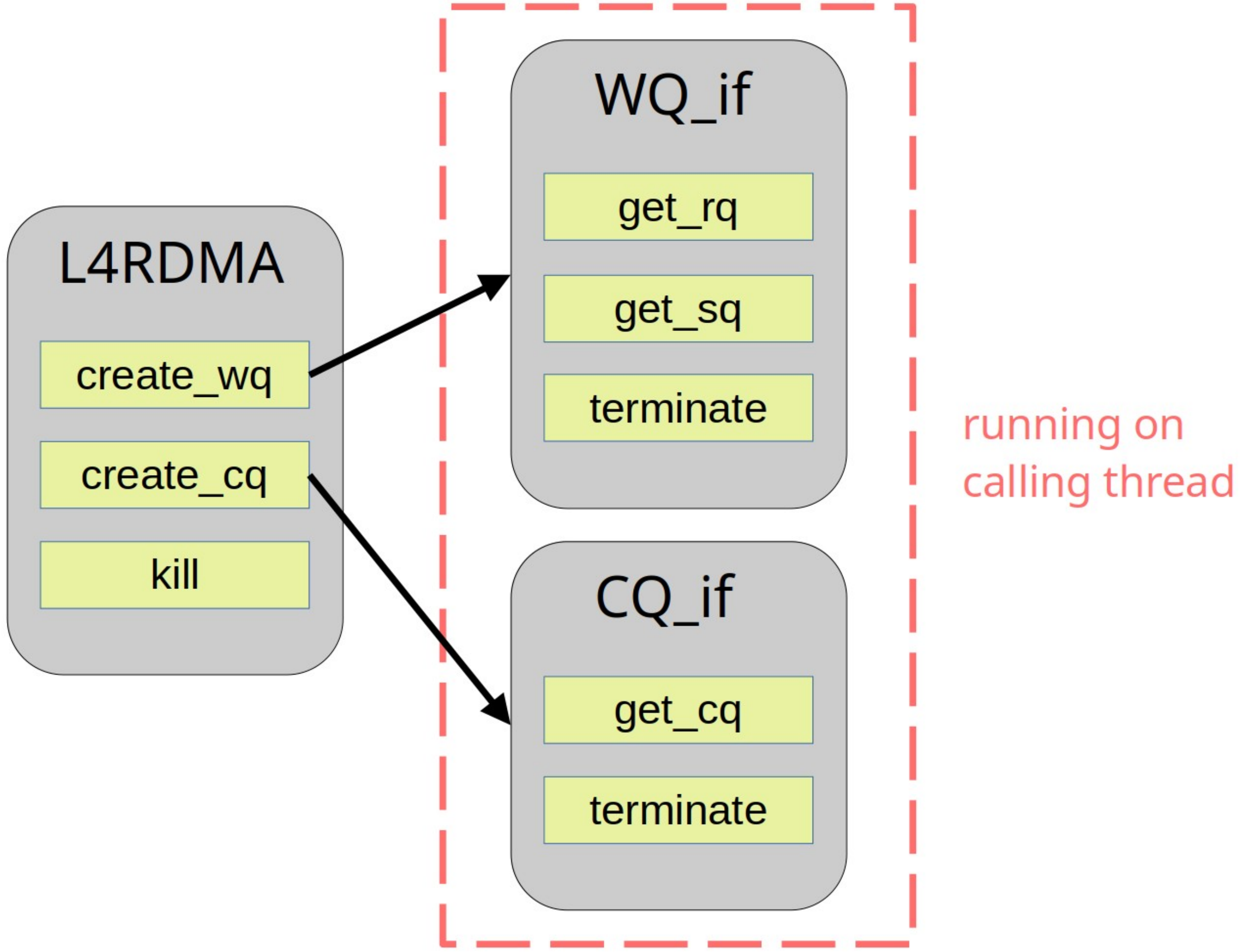


L4Re IPC Interface

- create Interface as a kernel object struct
- add inline RPC function definitions
- create a class using the Epiface template with the Kobject struct
- implement RPC functions as members of the class
- create interface of interface running on the caller thread by using Registry

Prototype IPC Interface

server/src/*_interface
server/src/interface.h
server/include/l4rdma.h



Server Loop Kill

server/src/main.cc

```
init_hca();

PRH_OPT handler_thread_opt;
pthread_t handler_thread = setup_event_queue(handler_thread_opt);

try {
    main_srv.loop();
} catch (...) {
    handler_thread_opt.active = false;
    pthread_join(handler_thread, NULL);
    teardown_hca();
}
```

III. Demo

:-)

IV. Conclusion

1. Summary
2. Future Work
3. Sources

Summary

? implement basic mlx5_core functions

- ✓ research HCA interfaces
- ✓ initialize and teardown HCA
- ✗ setup Work Queues

? implement L4Re IPC

- ✓ research IPC in L4Re
- ✓ provide basic IPC server with test client
- ✗ design & implement driver interface between libmlx5 and driver

✗ Ping Pong Demo!

Future Work

Allot of future work!!!

- WQ + TIR + TIS + FT
- pass real queue buffer memory in the IPC interface
- extend the IPC interface to allow for more configuration
- protect the control IPC calls like kill
- work on a Ping Pong Demo together with the user space library

Sources

(1) Mellanox Adapters Programmer's Reference Manual ConnectX-4

(2) linux/drivers/net/ethernet/mellanox/mlx5/core

(3) github.com/tmiemietz/luna

(4) github.com/tmiemietz/ixylon

(5) l4re.org

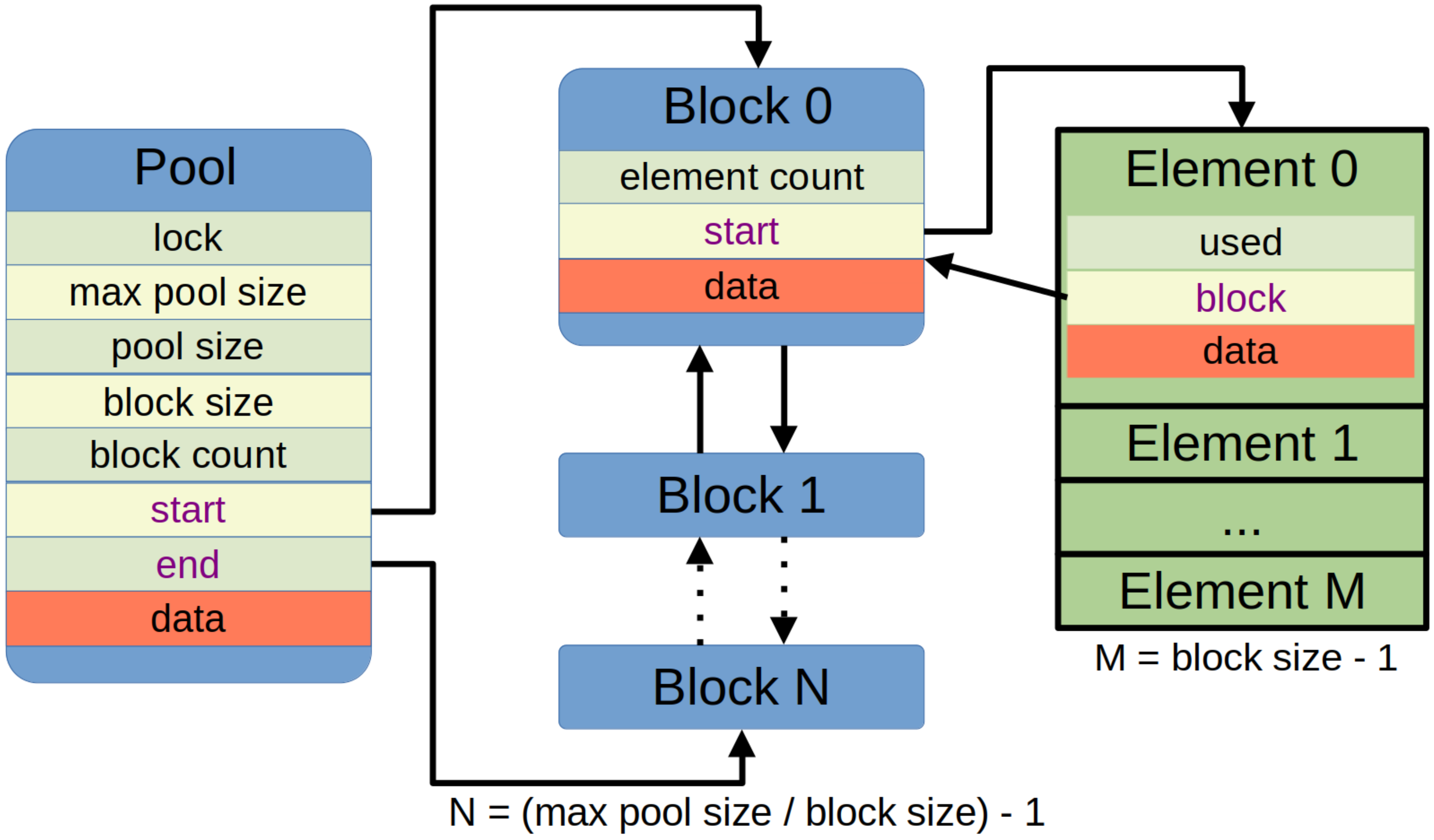
(6) wiki.osdev.org

(7) docs.nvidia.com/networking/display/rdmaawareprogrammingv17

Source Code → **github.com/lrenr/l4rdma**

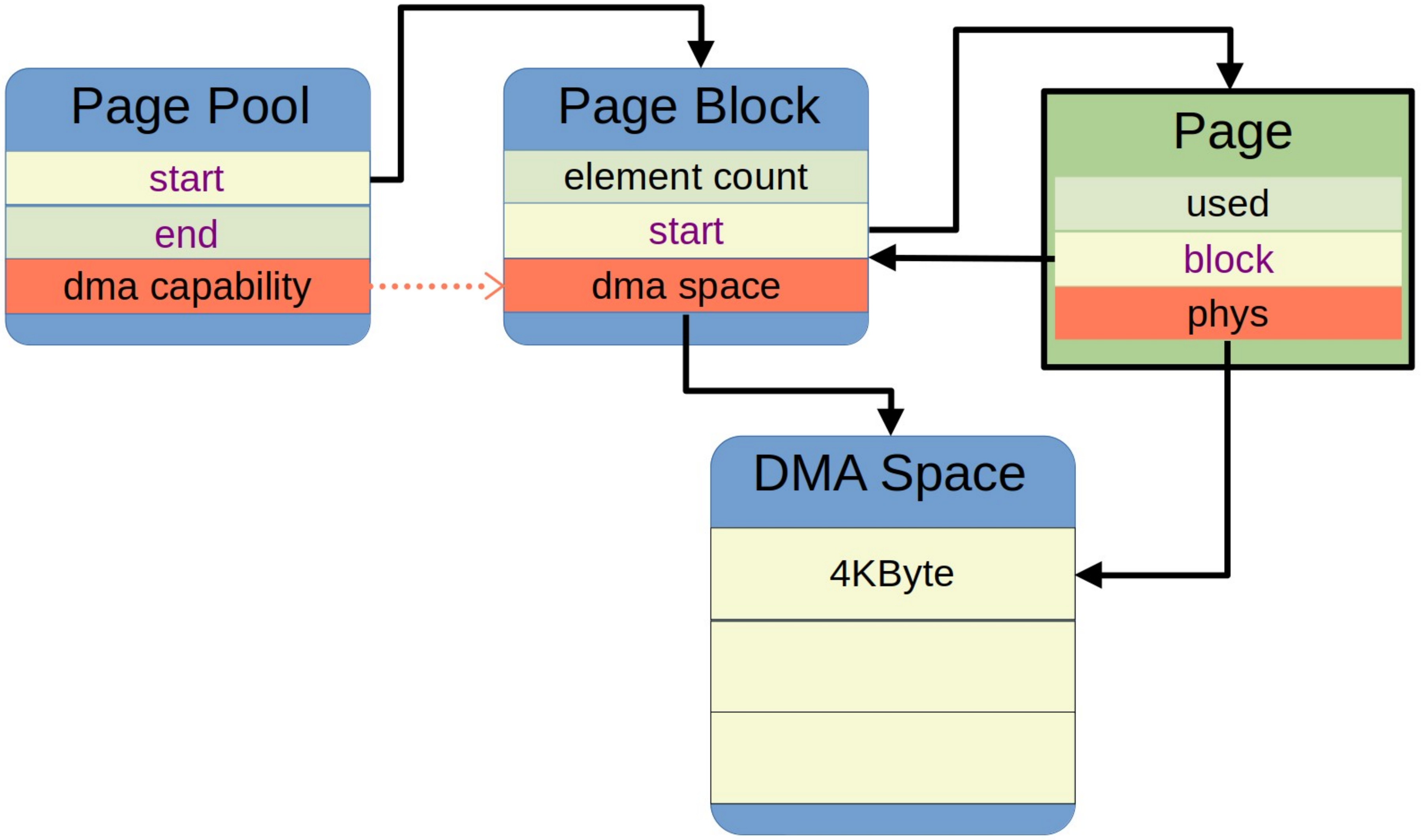
Bonus Slides

Pool (Block) Allocator



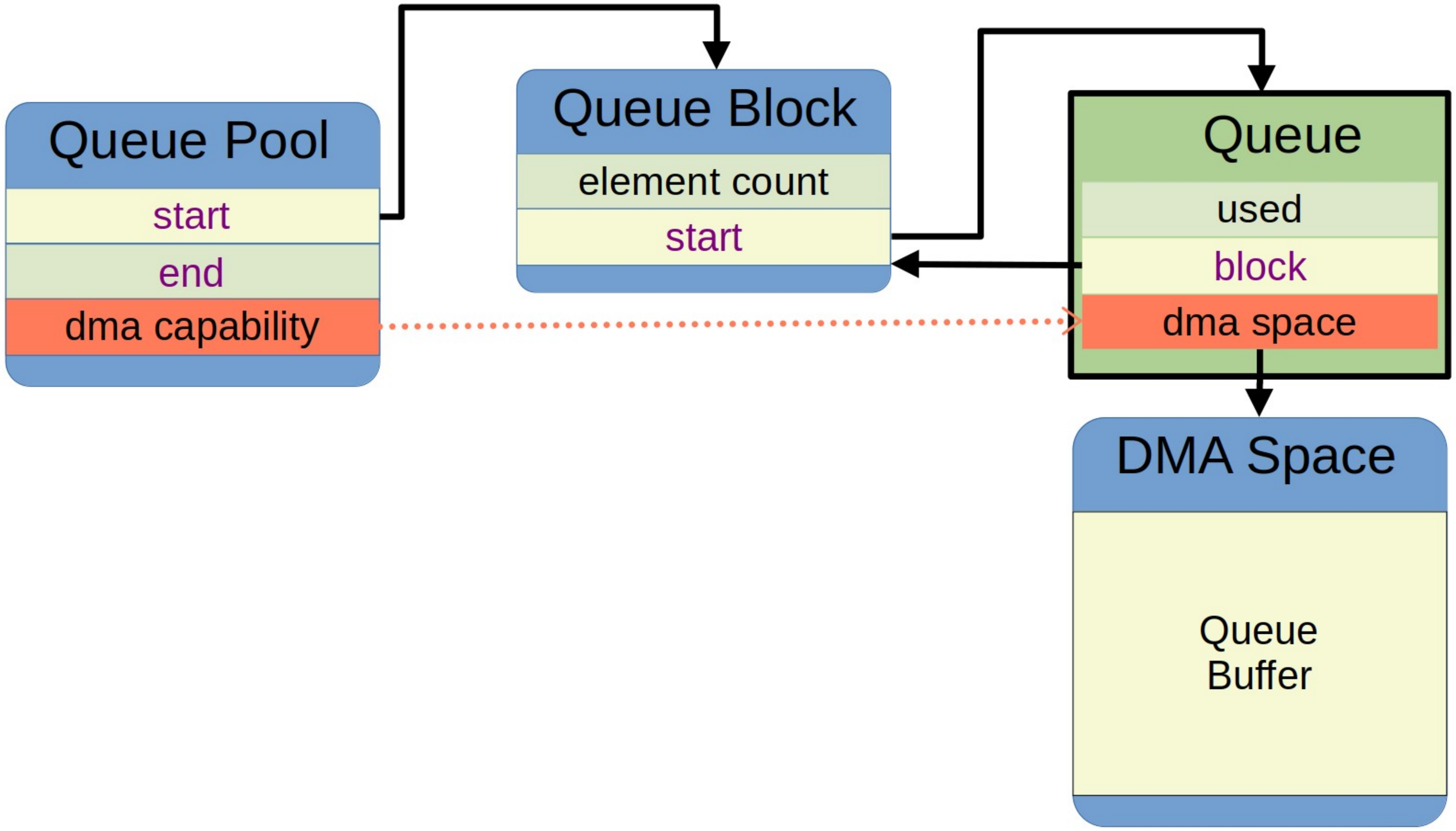
Page Pool

server/src/mem.*



Queue Pool

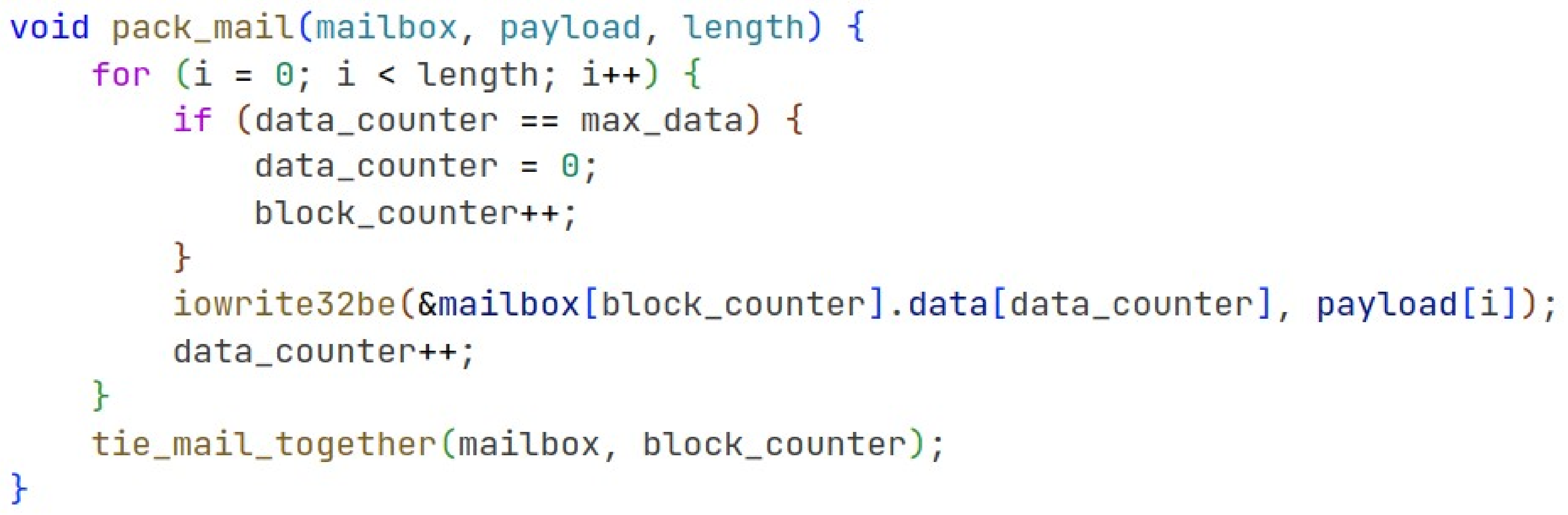
server/src/queue.*



Packing Mailbox

server/src/cmd.*

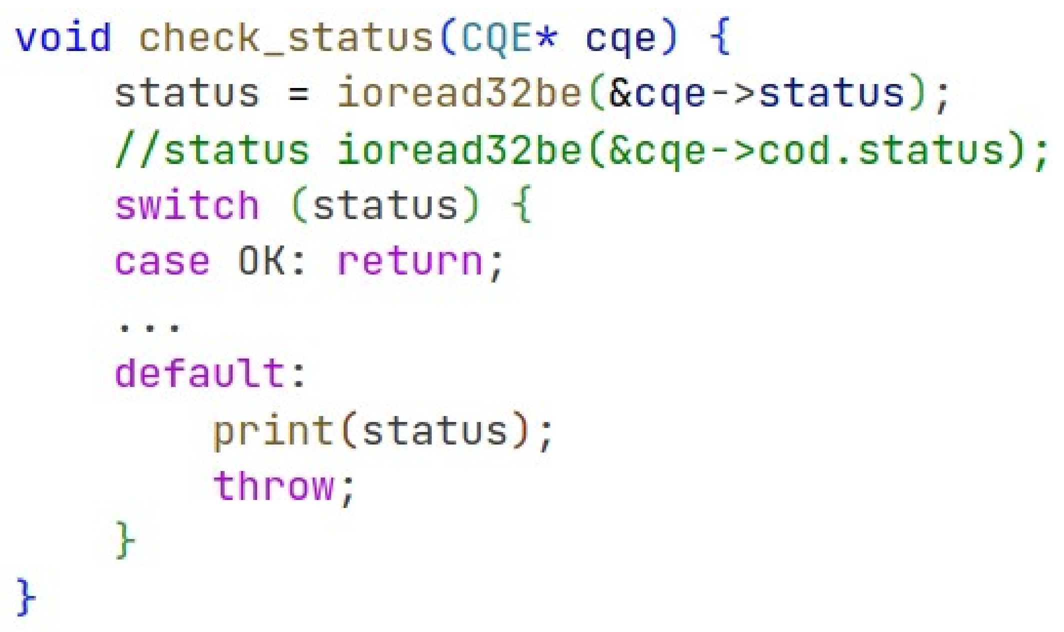
- first 8Byte of input and output are part of CQE
- input and output length of command must be known before and written to the CQE



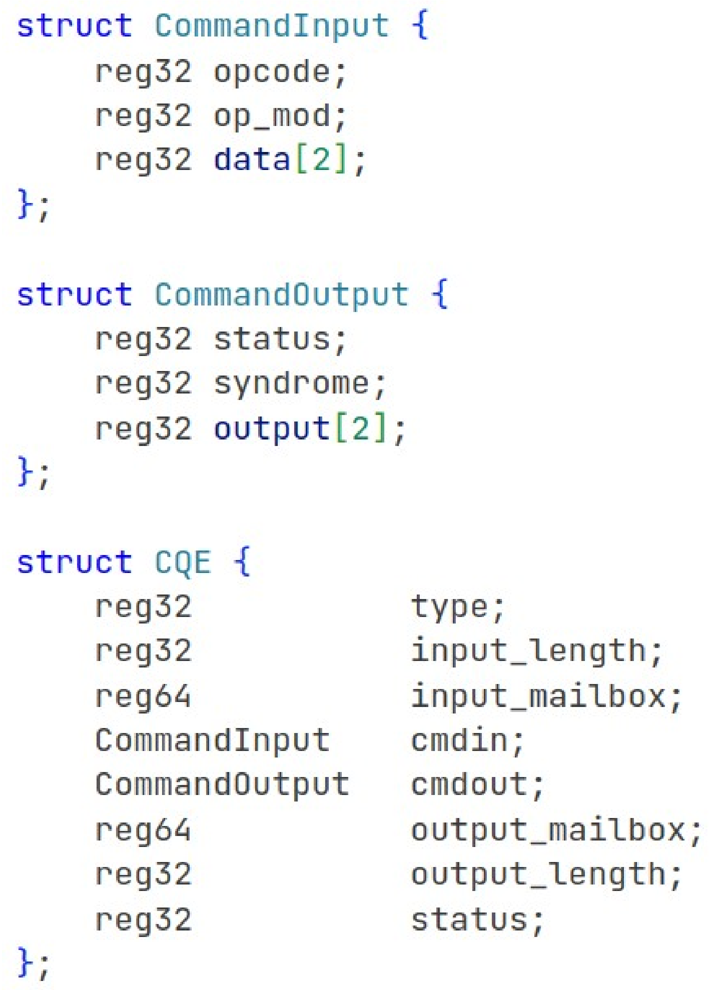
```
void pack_mail(mailbox, payload, length) {
    for (i = 0; i < length; i++) {
        if (data_counter == max_data) {
            data_counter = 0;
            block_counter++;
        }
        iowrite32be(&mailbox[block_counter].data[data_counter], payload[i]);
        data_counter++;
    }
    tie_mail_together(mailbox, block_counter);
}
```

Status Check

- separate status for CQE format and command execution



```
void check_status(CQE* cqe) {
    status = ioread32be(&cqe->status);
    //status ioread32be(&cqe->cod.status);
    switch (status) {
    case OK: return;
    ...
    default:
        print(status);
        throw;
    }
}
```



```
struct CommandInput {
    reg32 opcode;
    reg32 op_mod;
    reg32 data[2];
};

struct CommandOutput {
    reg32 status;
    reg32 syndrome;
    reg32 output[2];
};

struct CQE {
    reg32           type;
    reg32           input_length;
    reg64           input_mailbox;
    CommandInput    cmdin;
    CommandOutput   cmdout;
    reg64           output_mailbox;
    reg32           output_length;
    reg32           status;
};
```