Table of Contents

1. Introduction

This document describes RpMsgLite software library, and its internal components used for communication between applications residing on heterogeneous computing units which are interconnected over a shared memory hardware interface. It can be roughly thought of as an IPC mechanism where the 'P' stands for Processor instead of a Process in the IPC mechanisms offered by operating systems.

rpmsg-lite is a "C" library that is modularized such that it can be used in various settings like bare-metal, general purpose and real time operating systems.

2. References

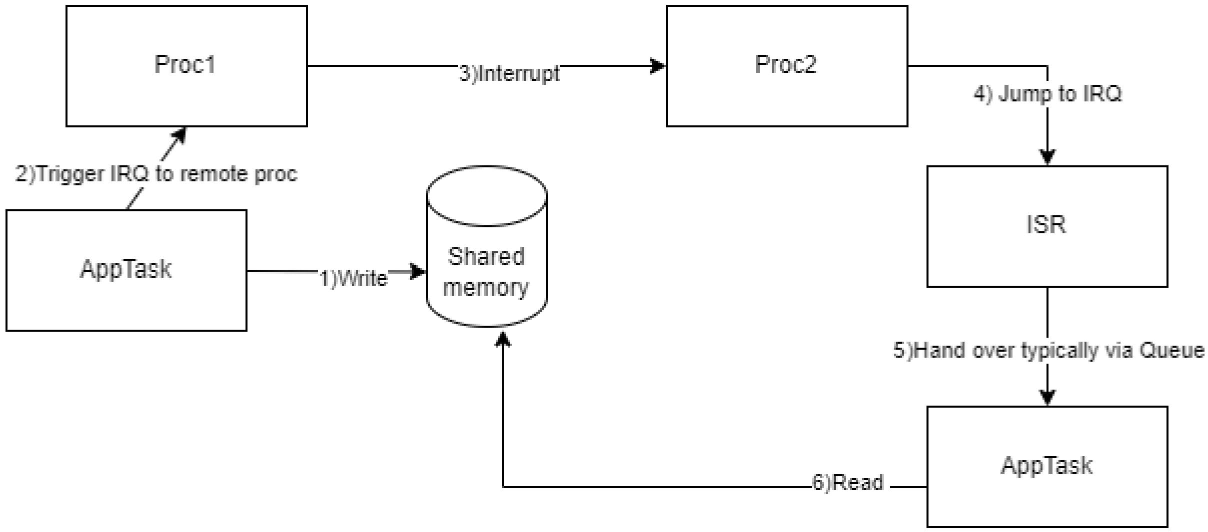
1. https://github.com/nxp-mcuxpresso/rpmsg-lite
2. https://blogs.oracle.com/linux/post/introduction-to-virtio

3. Scope

While rpmsg-lite provides a lot of capabilities like zero-copy operations, different options to receive messages, this document does not attempt to describe all the capabilities of the library. This is intended to be an overview of the library and description of the core data structures and functions that gets used for any of the capabilities the library offers. This can be considered as a supplement to the README file that has exhaustive coverage of all the features.

4. Motivation

A common model by which two applications in two different processors can communicate with each other is illustrated below.



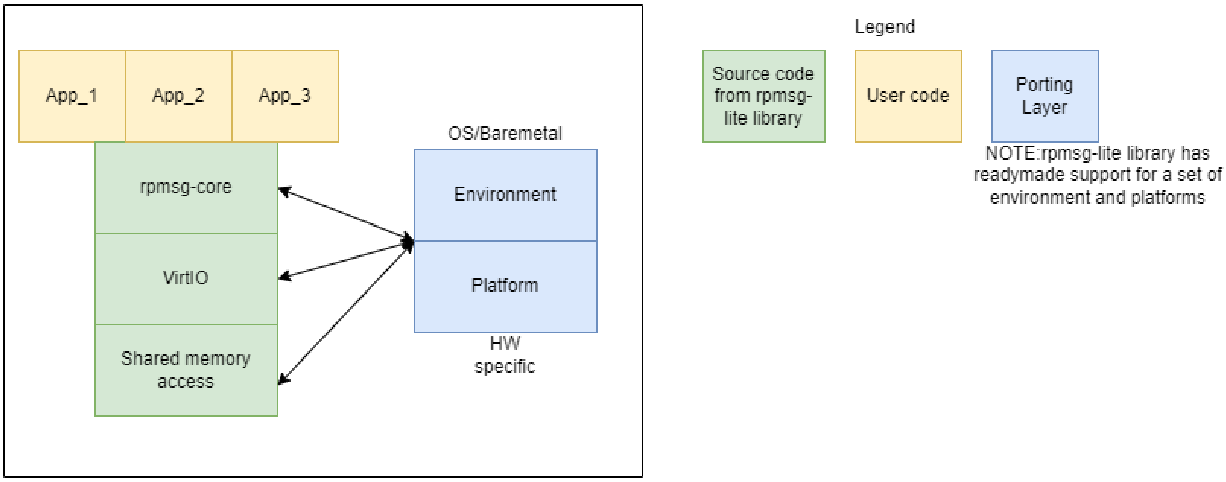
There are a lot of implementation challenges involved in the above model. For example,

1) Synchronization between Sender and Receiver such that message integrity is guaranteed. How can the Sender know if the Receiver has read the message, and it can send a subsequent message. Can it be done without explicit locking mechanism?
2) How to orchestrate between multiple application tasks that need to use the same shared memory. How to partition the shared memory such that the applications can independently communicate with a peer application on the other end.
3) How can the application be designed such that it can be ported easily to another hardware platform or a different OS running on the same hardware platform.

rpmsg-lite library provides a set software abstractions which the applications can use to do "send/receive" messages to peer applications and the library handles all the communication related challenges listed above.

5. Software Layers

The layering of software components is illustrated in the below diagram.



Environment defines the abstraction for actions like memory allocation, mutual exclusion (this is needed if rpmsg is used in a multi-threaded application) etc. It is driven by mainly the operating system that is used. (Or the bare-metal setup when OS is not used).

Platform defines H/W specific operations like memory address translations, interrupt handling etc.

The library supports, out-of-the-box, certain combinations of environment and platform. If the use case requires support for a new environment (a new OS for example) or a new hardware platform (a new SoC for example), then it must be implemented and bundled along with the rpmsg-lite library.

App_1, App_2 denote distinct applications using the same underlying rpmsg-lite instance (described more in sections below) to communicate with the peer applications in the remote processor. In this diagram this represents a logical functionality that is achieved by means of a rpmsg communication. It doesn't define a "OS Process" in the conventional sense. In fact, all the apps and the rpmsg code to be thought of belonging to one "OS Process" with rpmsg-core multiplexing the requests from each of the application on one processor and demultiplexing the requests on the remote processor to hand it over to the corresponding application.

The rpmsg-lite library doesn't create any "threads" or "tasks" internally. All actions are driven by the threads/tasks of the user applications or an Interrupt service routine (ISR).

5.1. rpmsg-core

This encapsulates the APIs provided by rpmsg-lite library to the applications. Following are the main APIs that the applications will need to use

1) rpmsg_lite_master_init() or rpmsg_lite_remote_init()
2) rpmsg_lite_create_ept()
3) rpmsg_lite_send()

NOTE: There is no separate API provided for recv() ! When a message is received for an endpoint, rpmsg-lite invokes a Callback function provided by the application when creating that endpoint. Typically, the Callback runs in an ISR context. The library also supports options for a blocking receive on a queue. It is not described here in this document.

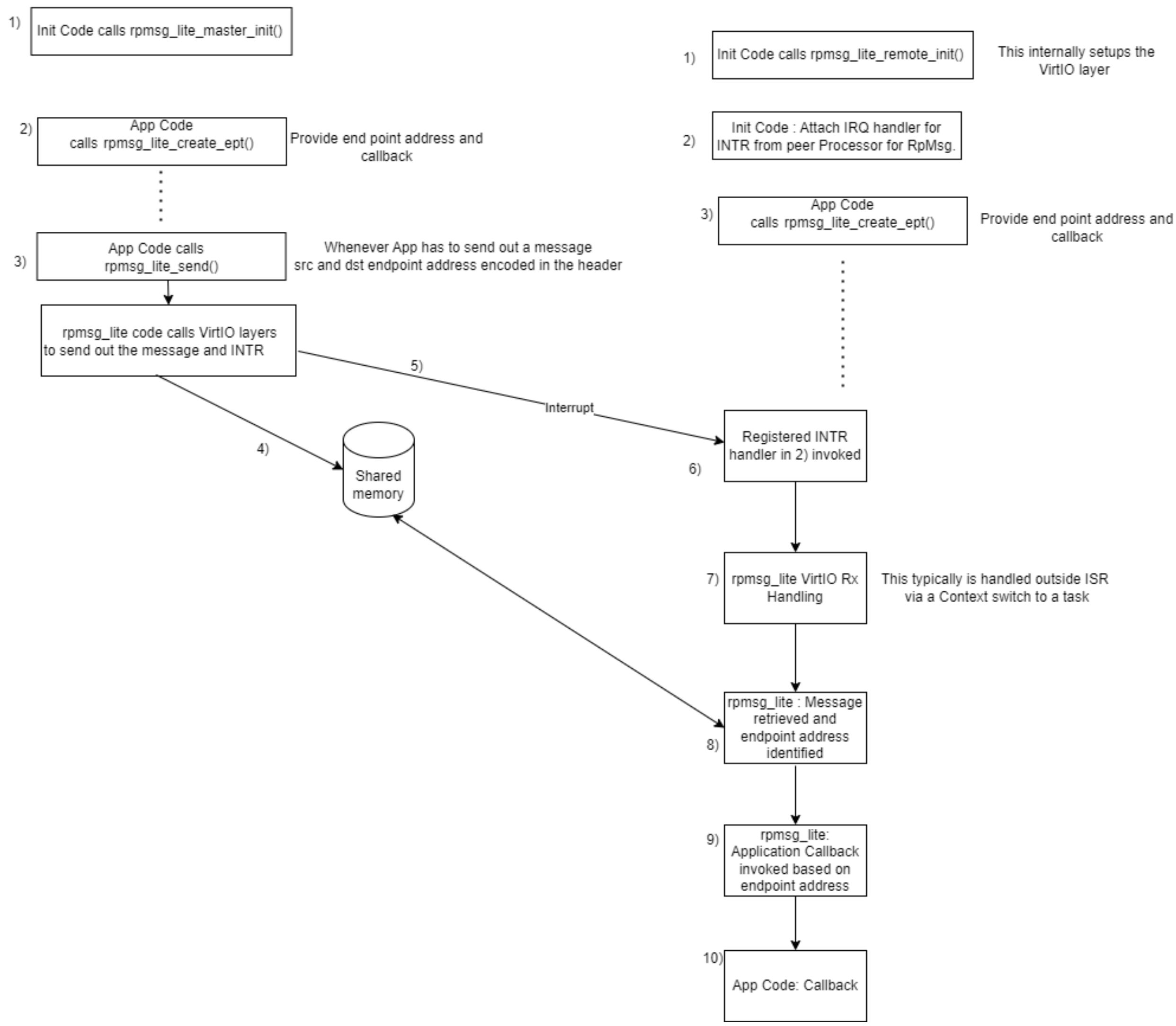
Terminologies

Link or Channel : rpmsg_lite_xxx_init() functions create an rpmsg_lite_instance object which denotes one end of a link. As the name indicates, rpmsg_lite_master_init() creates the object as the "master" end of the link and rpmsg_lite_remote_init() creates the object as the "remote" end of the link.

Master/Remote: This distinction arises due to usage of VirtIO as the access layer to manage the sharing of memory between multiple applications using the same link. Apart from that, there is no difference in terms of the ability of the master or remote instance to send or receive messages. A "Remote" can initiate a "send" to the Master apart from "receive". The same is true for the "Master". The distinction is made within the rpmsg-lite library and is described in detail in the VirtIO section (Ref [VirtIO]).

EndPoint: A uint32 number representing a specific instance that is using the Link for communication. There can be multiple end points on a given link. When sending a message, the value of destination end point must be provided. On receiving the message, the far end rpmsg-lite library decodes the end point to invoke the corresponding callback.

Below diagram illustrates a typical flow of events along with references to the functions listed above.

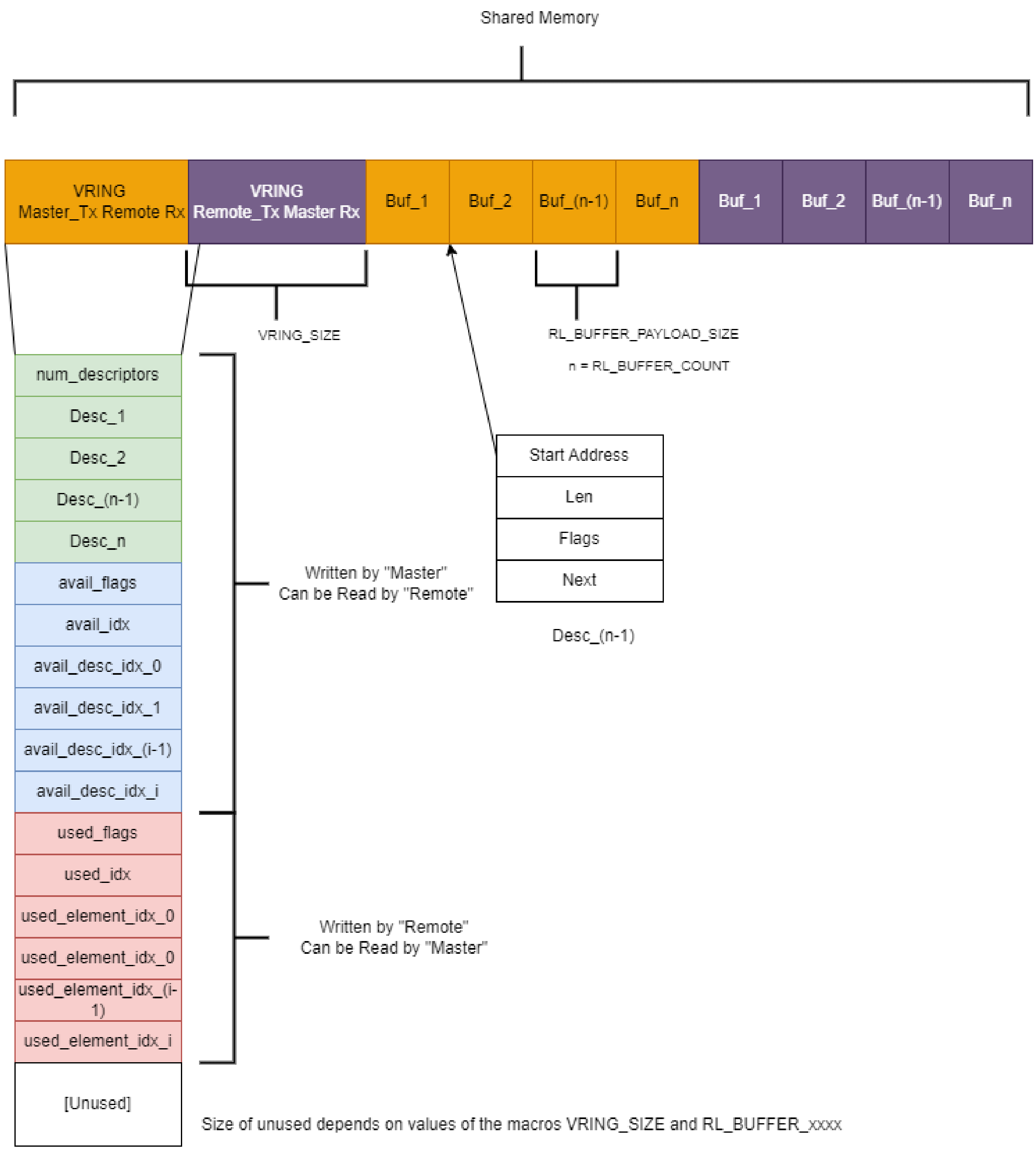


5.2. VirtIO

VirtIO is used as the “media access layer” by the rpmsg library. The shared memory to be used for the Link/Channel can be shared by multiple endpoints using the channel. VirtIO facilitates this process of communication between the endpoints without needing any “explicit locking” when reading or writing into the shared memory. It is adapted from the VirtIO framework used in virtualization environments (Ref [2])

The shared memory that is to be used between a “master” and “remote” rpmsg_lite_instance is **structured** into a set of circular buffers which enables the lockless communication mechanism.

The below diagram illustrates the partitioning of the shared memory. Between a "master" and "remote" there are two directions of transfer. "master tx to remote rx" and "remote tx to master rx". A given direction has one VRING and a set of buffers. VRING is the area in shared memory where the VirtIO layer keeps track of the availability and usage of buffers.



This partitioning of memory is done only by the “master” instance. While the “remote” can read the same shared memory area, until the “master” has setup the descriptor and avail areas in the VRING, the “remote” can’t start any send actions.

The rpmsg_lite_master_init() call is provided with start address of the shared memory and then a set of “C” macros that define this partitioning. The macros are

RL_BUFFER_PAYLOAD_SIZE : Size of 1 buffer. This is the max size the rpmsg_lite_send() can send.

RL_BUFFER_COUNT :   Number of buffers.

VRING_SIZE : Size of the overhead area that is allocated to store and manage the descriptors, available and used buffer information.

VirtIO layer manages the allocation, read/write, freeing of the buffers internally and the application end point can issue rpmsg_lite_send() when it wants to send and have its “Callback” invoked anytime there is a message addressed to it.

VirtIO provides two main abstractions to accomplish the partitioning and management.

1) vring – **struct vring** from lib/include/virtio_ring.h from rpmsg-lite repo
    a. This structure contains three circular buffers
        i. Descriptor Ring – refers to Desc_1 … through Desc_n in the diagram
        ii. Available Ring – refers to avail_xxx data section in the diagram
        iii. Used Ring – refers to used_xxx data section in the diagram

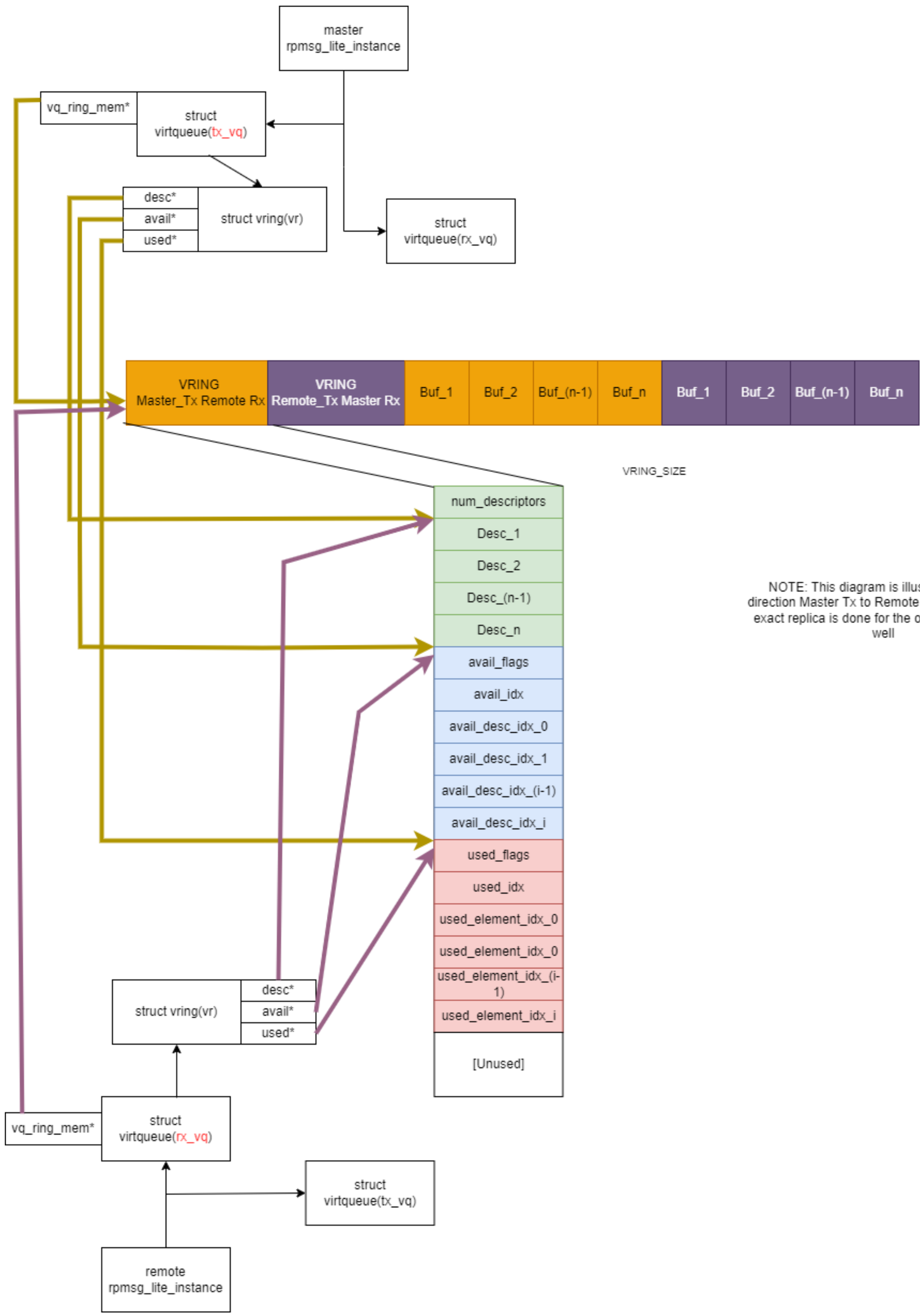
    The area pointed to by the Descriptor Ring and Available rings can be written *only* by the Master. It can be read by the Remote. Similarly, the area pointed to by the Used Ring can be written *only* by the Remote and can be read by the Master.

    These are called “rings” as the indexes wrap around when they reach the limit of the array size. Refer to [2] for more details.

2) virtqueue – **struct virtqueue** from lib/include/virtqueue.h from rpmsg-lite repo
    a. This has one struct vring object and other helper methods.

An rpmsg_lite_instance creates two virtqueue objects. One for Tx and the other for Rx.

NOTE: The struct vring and struct virtqueue objects reside in the “main memory” of the corresponding processor. Based on the configuration of rpmsg-lite, they are either allocated in heap or statically defined (for ex, in bare metal without any memory management library). struct vring has pointers to the actual vring location in the shared memory. Tx of one end (for ex, master) and Rx of the other end(for ex, remote) point to the same VRING. After rpmsg_lite_master_init() and rpmsg_lite_remote_init() is called on both ends, the data structures at both ends will be as depicted in the below diagram.



6. Error Handling

Once the "master" sets up the descriptors as part of its init, it is unchanged till the rpmsg_lite_instance is alive. So, it is possible under certain circumstances the communication to fail. Some of them are listed below.

1) rpmsg_lite_send() is invoked with data larger the size of 1 buffer as defined by RL_BUFFER_PAYLOAD_SIZE. In this case the method returns an error code RL_ERR_BUFF_SIZE.
2) rpmsg_lite_send() is invoked with data within the size of 1 buffer, but there are no free buffers, this is possible in cases where the remote processor is just not responsive enough – it could be busy doing some other tasks. In this case, the method returns an error code RL_ERR_NO_MEM

NOTE: The successful return of rpmsg_lite_send() only guarantees that the message is transferred to the buffers and the remote processor is notified. It doesn't have any explicit acknowledgement mechanism to say if the buffer is consumed. Applications running on top of this layer, must take care of such checks if required.