IPC over ring buffer

Overview

Currently, there are several ways for C-API and Firmware to communicate:

- Hardware Mailbox
- REQ/ACK mechanism
- Spare registers
- Using PIF to access share DRAM buffer directly

Typically, C-API writes configuration information (rules) to Spare registers and signal Firmware by one of two ways:

- Send message over Hardware Mailbox, or
- Using REQ/ACK

For transfer large data from Firmware to API such as Histogram, a global buffer (1KB) residing in DRAM is used:

1. C-API makes request to Firmware via Mailbox or REQ/ACK
2. Firmware put large data into global buffer and response to C-API via Mailbox or REQ/ACK

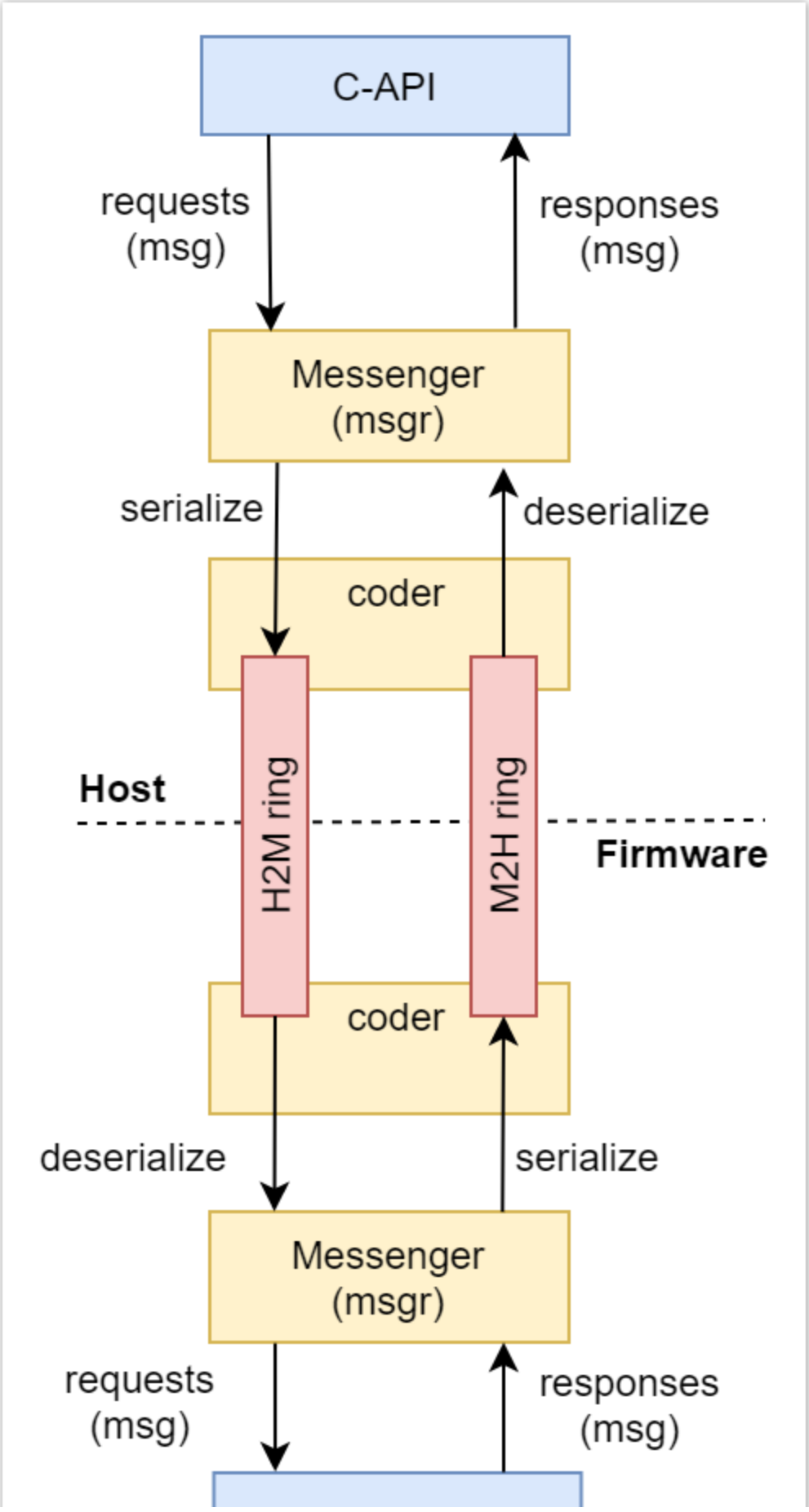
Due to Security reason, Hardware Mailbox and the ability to access to hardware registers may be cut-off, only access to shared memory is allowed from the C-API layer. Spare registers may be still allowed but we may not have enough spare registers for all of channels and we have to manage all of overlay registers to make sure that no fields are conflict/overlapped, and this may become messy when we added more debug features.

This page is to introduce new communication between API and Firmware in an Object Oriented way via Shared memory. The Shared memory can be:

- Scratch Memory
- Buffer in DRAM

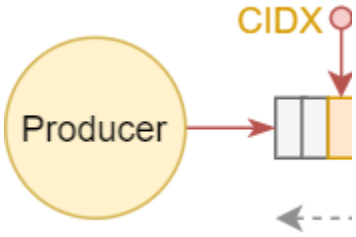
And the below figure shows all of new components (C-API, FW are existing ones), each component is corresponding to an abstract data type (can call **class**).

Class	Brief	Purpose
`msg`	Message	Information is encapsulated in request and response messages
`msgr`	Messenger	Responsible to message delivering. It works with the coder
`coder`	Encoder/decoder	Support functions to encode/decode primitive data types over `ring` buffer. Deal with data alignment.
`ring`	Circular Buffer	A byte-stream implemented by circular buffers



Ring buffer

In general, a `ring` just likes below figure.



A `ring` has:

- Buffer to store data. This buffer is able to hold N bytes, N is called ring size. There may be a requirement that number of bytes put to the ring must be multiple of 2-bytes or 4-bytes. This is called alignment and is usually power of 2.

- Producer index - **`PIDX`**: to hold the index in the buffer where new data has just put in. This pointer is updated by the producer.
- Consumer index - **`CIDX`**: to hold the index in the buffer where data had been consumed. This pointer is updated by the consumer.

Data alignment can be different depending on access spaces (MCU or MDIO), memory types. As below table.

Access space	MCU	MDIO
DRAM	1-byte	4-bytes
Scratch Memory	2-byte	2-bytes

Accessing to **`PIDX`** and **`CIDX`** will help:

- Producer to know how many free spaces that it can enqueue (put)
- Consumer to know how many bytes are available in buffer to dequeue (pop)

Ring data buffer is own by Firmware and it can access this buffer directly. The C-API layer can access this buffer in a burst way:

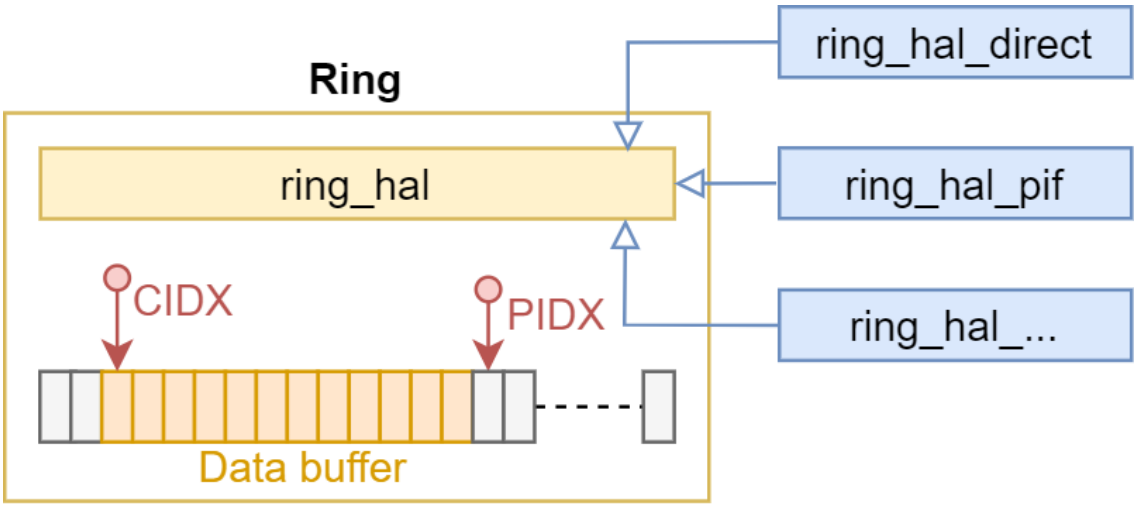
- Over PIF interface, if accessing to IRAM/DRAM is allowed.
- Scratch Mem burst access. This is a basic requirement to HW team.

Accessing to the PIDX/CIDX/Buffer are different, direct, indirect, MDIO, PIF, ... To abstract this, the `ring_hal` (Ring Hardware Abstraction Layer) is defined to abstract single/burst read/write. The `ring_hal` just defines a set of function pointers, there are two default implementations of this HAL:

- `ring_hal_direct`: used by FW to directly access ring data structure
- `ring_hal_pif`: used by API to access ring data structure over PIF/MDIO

Each specific project can have different implementation of ring HAL.

The below figure shows what are owned by the `ring`.



That means, to construct a ring, a ring metadata is required along with ring HAL. Ring metadata includes:

- Data buffer address
- Size of data buffer (ring size)
- PIDX address
- CIDX address

The API below is to construct a ring.

```
struct ring *ring(struct ring *self,
                  const struct ring_metadata *metadata,
                  struct ring_hal *hal);
```

Rings are owned by Firmware, so after constructing rings, depend on each project, FW can advertise ring meta data to C-API over:

- Spare registers
- Firmware information data structure residing in DRAM (e.g. `plr_firmware_info_t`)
- Scratch Memory
- Whatever …

Below is an example how to construct a H2M ring in C-API context/

```
/* Assume that the ring metadata is fetched by some ways, depend one each specific products */
struct ring_metadata h2m_metadata;
/* Codes to fetch ring meta data ... */
/* ... */

/* Ring HAL */
struct ring_hal_pif r_hal;
ring_hal_pif(&r_hal, plr_mcu_pif_read, plr_mcu_pif_write);

/* Construct ring with metadata and HAL */
struct ring h2m_ring;
ring(&h2m_ring, &h2m_metadata, &r_hal);

/* Tell the internal implementation that we are producer
 * to optimize pointer fetching */
ring_mode_set(&h2m_ring, RING_MODE_PRODUCER);
```

> ***NOTE**: The `ring_hal_pif` always assume all of addresses are MCU space addresses (IRAM/DRAM). For performance purpose, the rings constructed by FW can use PFL registers (accessible via MDIO) for PIDX/CIDX since these pointers frequently accessed (to check for data available), the `ring_hal_pif` will NOT work in this case. It is recommended to have each specific project provides concrete implementation for ring HAL. For single read/write, it should check if addresses are MDIO addresses, then MDIO read/write functions will be used. See this section for an example implementation.*

After a ring is constructed, data can be sent/received over it by below APIs.

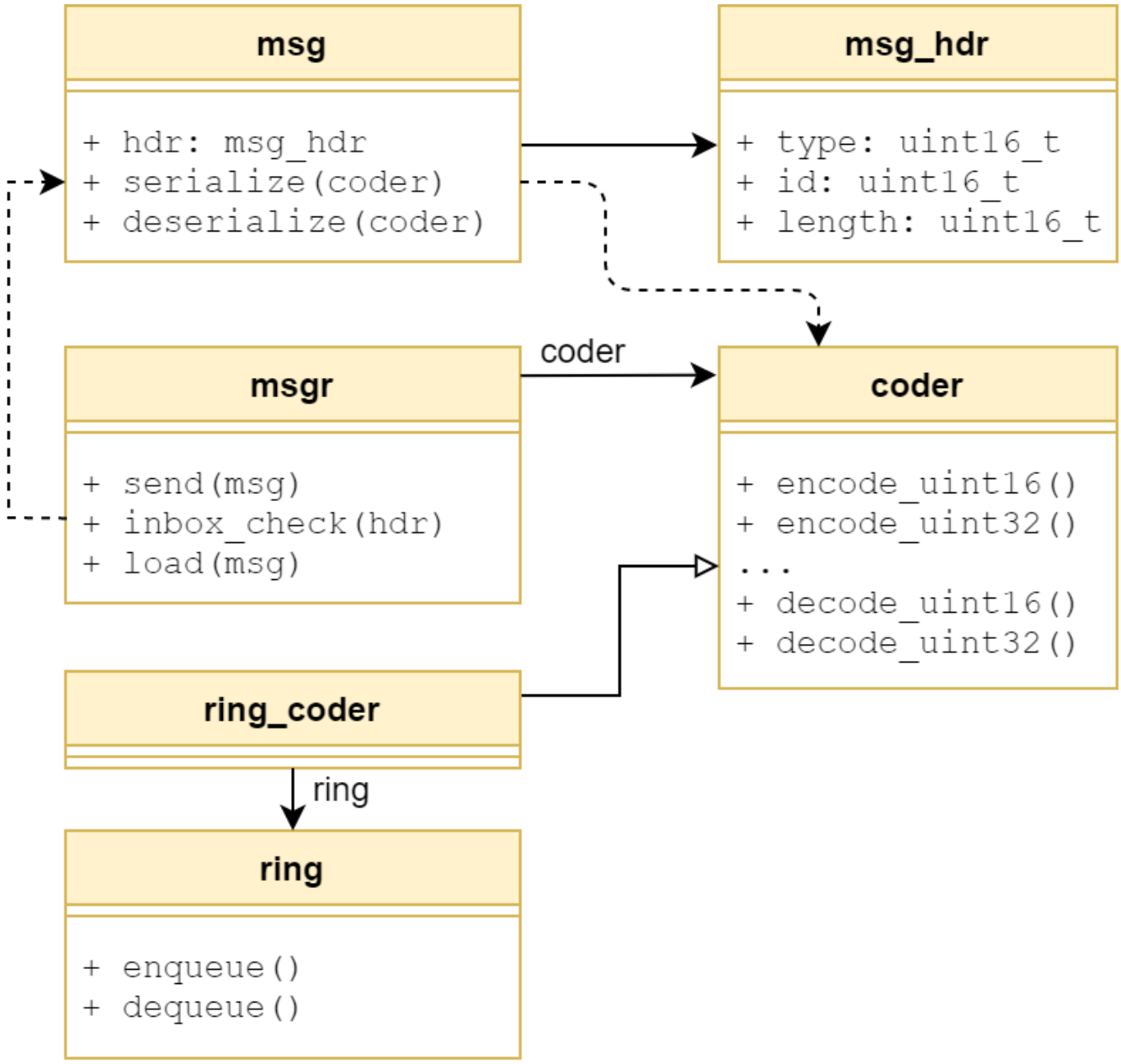
```
/* Send */
uint16_t ring_enqueue(struct ring *self, uint8_t *buf, uint16_t len);

/* Receive */
uint16_t ring_dequeue(struct ring *self, uint8_t *buf, uint16_t len);
```

For full ring APIs, please refer **`ring.h`**

Messaging

This new messaging is designed in an Object Oriented way. Below class diagram show all of basic classes and their relationship.



- All of information sent/received between C-API and Firmware are encapsulated in messages.
- The messenger is responsible to delivering messages. So, it supports methods to send message, check/load if any new messages come.
- The coder provide different methods for encode/decode primitive data types such as `uint8_t/uint16_t/uint32_t/uint64_t` over byte stream like `ring` or hardware mailbox (not supported yet). This is also intended to have data be encrypted/decrypted at this layer if security gets involve.

Message

As mentioned earlier, message is used to encapsulate information to be sent/received between C-API and Firmware. A message contains:

- Header which has
  - Message type
  - Message ID.
  - Data length.
- `serialize` method is to work with the `coder` to encode message header and additional fields to coder's byte stream (which is `ring`)
- `deserialize` method is a reversed process of `serialize`. This works with the `coder` to decode (extract) its additional fields from the `coder` byte stream.

All of messages must implement `serialize/deserialize` methods. These methods will be used by the messenger to send/receive messages, it does not care the internal data structures of a message, just simply call those methods.

Basically, we can implement messages for all of possible attributes and operations, but doing so may require much of codes. So, we'd better have messages to:

- SET/GET channel attributes.
- Execute device-level operation.
- Other general purpose messages …

And below is one recommendation on general purpose messages.

Message type	Purpose	Associate fields

`PLR_IPC_MSG_TYPE_CHN_ATTR_SET_REQ`
 	
`PLR_IPC_MSG_TYPE_CHN_ATTR_SET_RES`
	Request/response to SET channel attribute	• Channel attribute ID, like:
		  • `PLR_IPC_CHN_ATTR_ID_HISTOGRAM`
		  • `PLR_IPC_CHN_ATTR_ID_TX_FIR`
		  • …
		• Channel ID
		• Attribute value: like `plr_ipc_tx_fir_t`, `plr_ipc_fmon_cfg_t`, …
		• Status: OK/ERROR

`PLR_IPC_MSG_TYPE_CHN_ATTR_GET_REQ`
 	
`PLR_IPC_MSG_TYPE_CHN_ATTR_GET_RES`
	Request/response to GET channel attribute	

`PLR_IPC_MSG_TYPE_OP_REQ`
 	
`PLR_IPC_MSG_TYPE_OP_RES`
	Request/response to execute an operation at device level	• Operation ID, like:
		  • `PLR_IPC_OP_ID_FEC_STATS_CFG`
		  • `PLR_IPC_OP_ID_FEC_STATS_GET`
		  • …

Each message instance should have unique ID, for now, this ID is hardcode to a specific value (e.g. `0xABCD`), of course, it is not a right usage of message ID. The requestor (C-API) should have some way to generate unique IDs (atomic counter, for example).

Just an example, the message to SET/GET a channel attribute can be defined as below. There is a `process()` function pointer (callback) provided by the receiver (Firmware) for further processing on the received message depending on what the attribute ID is.

```
typedef struct plr_ipc_chn_attr_req_s
{
    struct msg msg;    /**< Generic message */

    uint16_t attr_id;   /**< Attribute ID */
    uint8_t channel;    /**< FW Channel ID */
    uint8_t ip_block;   /**< Channel's IP block */

    /* Associated data (optional) */
    uint16_t length;    /**< Associated data length */
    uint8_t *data;      /**< Associated data to send to API */

    /** Process and return number of decoded data */
    uint32_t (*process)(struct plr_ipc_chn_attr_req_s *self, struct coder *coder);
}plr_ipc_chn_attr_req_t;
```

The `serialize` method just calculate the data length, update the header then encode it along with attribute ID, channel, IP block. Associated data will be encoded if provided too. Like below code.

```
static enum msg_ret plr_ipc_chn_attr_req__serialize(struct msg *msg, struct coder *coder)
{
    enum msg_ret msg_ret;
    struct plr_ipc_chn_attr_req_s *self = (struct plr_ipc_chn_attr_req_s *)msg;
    struct msg_hdr *hdr = msg_hdr_get(msg);
    uint32_t nb_bytes = 0;

    /* Encode header */
    hdr->length = sizeof(self->attr_id)  +
                  sizeof(self->channel)  +
                  sizeof(self->ip_block) +
                  self->length;
    msg_ret = msg_hdr_encode(hdr, coder);
    if (msg_ret != MSG_OK)
        return msg_ret;

    /* Encode payload */
    nb_bytes += coder_uint16_encode(coder, self->attr_id);
    nb_bytes += coder_uint8_encode(coder, self->channel);
    nb_bytes += coder_uint8_encode(coder, self->ip_block);

    if (self->length)
        nb_bytes += coder_uint8s_encode(coder, self->data, self->length);

    return (nb_bytes == hdr->length) ? MSG_OK : MSG_LENGTH_MISMATCH;
}
```

The `deserialize` method is just a reversed of `serialize`.

> ***IMPORTANT:*** *the `deserialize` must not decode the header since the header is already decoded by the messenger, will be discussed later.*

```
static enum msg_ret plr_ipc_chn_attr_req__deserialize(struct msg *msg, struct coder *coder)
{
    plr_ipc_chn_attr_req_t *self = (plr_ipc_chn_attr_req_t *)msg;
    struct msg_hdr *hdr = msg_hdr_get(msg);
    uint32_t nb_bytes = 0;

    /* Decode payload */
    nb_bytes += coder_uint16_decode(coder, &self->attr_id);
    nb_bytes += coder_uint8_decode(coder, &self->channel);
    nb_bytes += coder_uint8_decode(coder, &self->ip_block);

    if (self->length)
        nb_bytes += coder_uint8s_decode(coder, self->data, self->length);

    /* Ask the callback for further processing */
    nb_bytes += self->process(self, coder);

    return (nb_bytes == hdr->length) ? MSG_OK : MSG_LENGTH_MISMATCH;
}
```

Having these two methods implemented, the constructor of this message is just as below.

```
static struct msg_ops plr_ipc_chn_attr_msg_ops =
{
    .serialize   = plr_ipc_chn_attr_req__serialize,
    .deserialize = plr_ipc_chn_attr_req__deserialize
};


struct msg *plr_ipc_chn_attr_req(plr_ipc_chn_attr_req_t *self, uint32_t msg_type)
{
    memset(self, 0, sizeof(plr_ipc_chn_attr_req_t));

    self->msg.ops = &plr_ipc_chn_attr_msg_ops;
    self->msg.hdr.id = 0xABCD;
    self->msg.hdr.type = msg_type;

    return &self->msg;
}
```

The implementation of `serialize` and `deserialize` should reside on the same source file so that we can easily see their symmetric. For saving code space, we'd better have a compiler flag __FW__ to determine what code should be compiled on Firmware and C-API.

> *INFO: The idea of __FW__ is just like __KERNEL__ compiler flag when we work with Linux Kernel module.*

Below is an example.

```
#ifdef __FW__

static enum msg_ret plr_ipc_chn_attr_req__deserialize(struct msg *msg, struct coder *coder)
{
    // ...
}

static struct msg_ops plr_ipc_chn_attr_msg_ops =
{
    .serialize   = NULL,
    .deserialize = plr_ipc_chn_attr_req__deserialize
};

#else

static enum msg_ret plr_ipc_chn_attr_req__serialize(struct msg *msg, struct coder *coder)
{
    // ...
}

static struct msg_ops plr_ipc_chn_attr_msg_ops =
{
    .serialize   = plr_ipc_chn_attr_req__serialize,
    .deserialize = NULL
};

#endif

struct msg *plr_ipc_chn_attr_req(plr_ipc_chn_attr_req_t *self, uint32_t msg_type)
{
    memset(self, 0, sizeof(plr_ipc_chn_attr_req_t));

    self->msg.ops = &plr_ipc_chn_attr_msg_ops;
    self->msg.hdr.id = 0xABCD;
    self->msg.hdr.type = msg_type;

    return &self->msg;
}
```

The public APIs to serialize/deserialize are below and they are used by the messenger. Notice that the `coder` is input to both APIs.

```
enum msg_ret msg_serialize(struct msg *self, struct coder *coder);
enum msg_ret msg_deserialize(struct msg *self, struct coder *coder);
```

Messenger

The messenger is quite simple, its job is to send/receive messages. The messenger needs `coder` object but it does not actually own that coder but application does. So to have this coder, it gives the application `delegate`, application needs to implement this delegate to return appropriate coder `object`. The `delegate` is defined for several purposes.

1. To ask application return appropriate coder for sending/receiving messages
2. To notify application if messages are sent
3. To notify application if new messages are received

Below are related APIs to construct a messenger.

```
/* Construct messenger */
struct msgr *msgr(struct msgr *self);

/* Messenger delegate to return the coder object */
void msgr_delegate_set(struct msgr *self, void *ctx, struct msgr_delegate *delegate);
```

After constructing messenger, application can send messages by calling below API. The internal implementation of the send API is pretty simple, it just simply as the `delegate` to return `coder` object, then pass this coder object along with the message to be sent to the API `msg_serialize()`, the message `serialize` will know how to serialize its fields onto the `coder`.

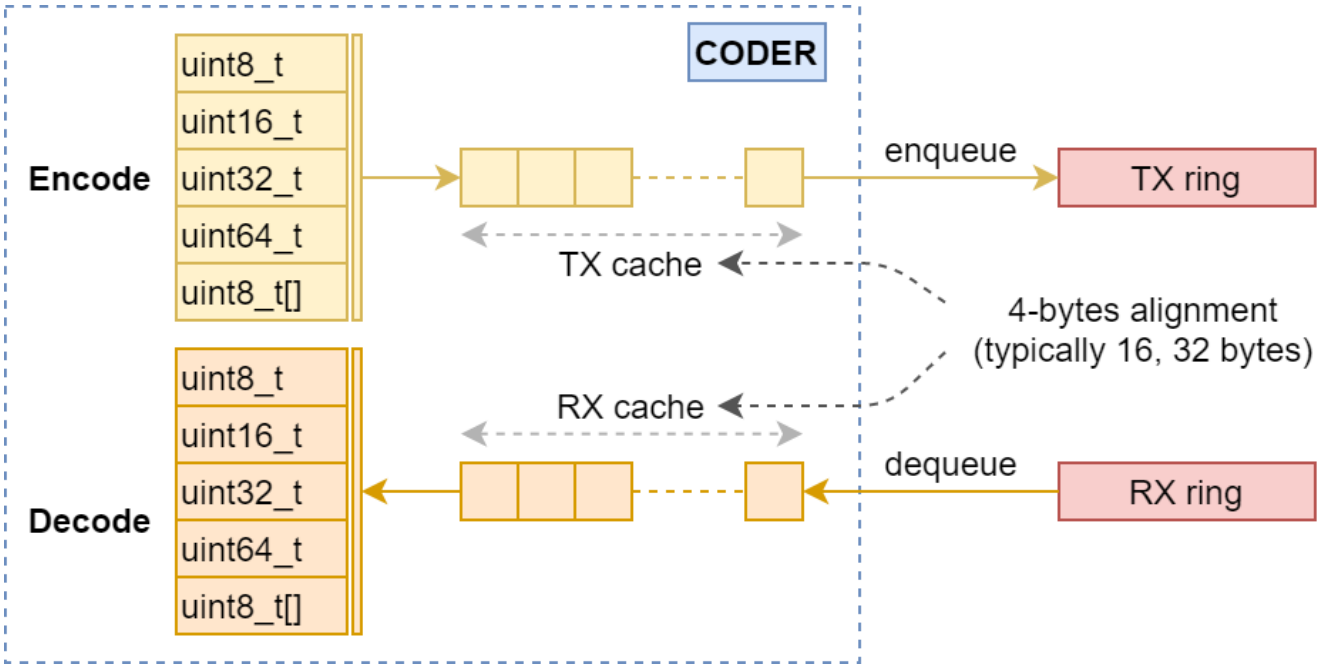
```
enum msg_ret msgr_send(struct msgr * self, struct msg *msg);
```

At the receive side, the Firmware, it can receive any different kind of messages. To load the new incoming message, the message header must be examined to know what type it is so that concrete message can be instantiated to fully load it. Two APIs below are used to deal with message receiving.

```
/* Check if any new message and pop header from data stream */
enum msg_ret msgr_inbox_check(struct msgr * self, struct msg_hdr *hdr);

/* Fully load message */
enum msg_ret msgr_load(struct msgr * self, struct msg *msg);
```

Coder



The coder is to work with any data streams which has different data alignment. Currently, only `ring` stream is supported but we can add Hardware Mailbox stream if we'd like. Ring data alignment is 4-bytes, which mean number of bytes to send and receive must be multiple of 4-bytes.

It is the job of the coder implementation to deal with data alignment and it supports set of basic APIs to encode/decode primitive data types such as `uint8_t`, `uint16_t`, `uint64_t` and `uint8_t[]` array. To make sure the data alignment, the coder internally has TX/RX cache.

Below are APIs provided by the `coder` is an abstract class.

API	Purpose
`coder_uint8_encode()/coder_uint8_decode()`	Encode/decode uint8_t
`coder_uint16_encode()/coder_uint16_decode()`	Encode/decode uint16_t
`coder_uint32_encode()/coder_uint32_decode()`	Encode/decode uint32_t
`coder_uint64_encode()/coder_uint64_decode()`	Encode/decode uint64_t
`coder_uint8s_encode()/coder_uint8s_decode()`	Encode/decode buffer with specified length.

Currently, only the `ring_coder` is supported to encode/decode data over `ring` buffer. To utilize burst transfer:

- At the encoding direction: data are encoded into the TX cache, when the cache is full, the coder will call `ring_enqueue()` API to burst send the whole cache buffer.
- At the decoding direction: if the RX cache is empty or it does not have enough data for decoding a specific data type (e.g. want to decode `uint64_t` but only 4 bytes are available in the cache), then the API `ring_dequeue()` is called to burst read and fill this cache. Then decoding can continue to take places.

The API below is used to flush the TX cache. When the cache is flushed, if number of bytes in the cache is not multiple of data alignment of the underline stream, then the flush function will pad. For example, if data alignment is 4-bytes and only 1 byte are still in cache, then 3 bytes will be padded.

```
uint16_t coder_flush(struct coder *self);
```

Benchmark

This new IPC is basically integrated into Mira project. So, below is what we have with Mira.

Code size

Replacing Hardware Mailbox with this IPC, below is code size changes.

	text	data	bss	dec
Mailbox	172162	3216	19840	195218
ring/IPC	172206	3272	19592	195070
Diff	**44**	**56**	**-248**	**-148**

Basically, code size of this new IPC is similar to Mailbox, it even reduces a little bit on DRAM.

Ring data transfer rate

Below result is almost close to PIF bandwidth and we can optimize more. The H2M is always faster than M2H because we have MDIO burst write, there is no MDIO burst read supported.

Direction	rate
H2M	12.0 kBps
M2H	10.8 kBps

APIs speed

Histogram is used for this benchmark.

API	Communication method	speed
`plr_rx_dsp_get_histogram`	Hardware Mailbox	**248.2** ms/channel
`plr_rx_dsp_get_histogram`	ring/IPC	**68.2** ms/channel
`plr_hist_querier`	ring/IPC	**32.1** ms/channel

The `plr_hist_querier` are new APIs to query histogram of a bunch of channels. Below is C example.

```
/* Histogram done callback, here we just simple plot the result histogram */
static void hist_on_done(
    struct plr_hist_querier_s *querier,
    uint32_t die, uint8_t channel, e_plr_intf intf,
    uint32_t *hist_data, uint16_t data_len)
{
    plr_rx_hist_ascii_plot(die, channel, hist_data, data_len);
}

/* Buffer to receive histogram data */
uint32_t hist[160];

/* Construct querier */
plr_hist_querier_t querier;
plr_hist_querier_cb_t callback = {
    .on_done = hist_on_done;
};
plr_hist_querier(&querier, hist, 160, &callback);

/* Add all channels to query */
for (uint8_t chn_id = 1; chn_id <= 8; chn_id++)
    plr_hist_querier_add_channel(&querier, die, chn_id, PLR_INTF_LRX);

/* Now start querying */
plr_hist_querier_run(&querier);
```

Appendix

SET/GET channel attributes

By implementing general purpose messages `plr_ipc_chn_attr_req/plr_ipc_chn_attr_req` to encapsulate information of desired channel attributes, we can then define a general purpose APIs to set/get channel attributes over this ring/IPC channel.

```
inphi_status_t plr_ipc_chn_attr_set(int32_t pdie, uint32_t channel, e_plr_intf intf, plr_ipc_attr_id_t attr_id,
uint8_t *data, uint16_t length);
inphi_status_t plr_ipc_chn_attr_get(int32_t pdie, uint32_t channel, e_plr_intf intf, plr_ipc_attr_id_t attr_id,
uint8_t *data, uint16_t length);
```

An attribute can be primitive data type or it is a data structure. If attribute is a data structure such as FIR, both API and Firmware must see the same definition. Let's take FIR as an example, we can have a common data structure as below:

```
struct plr_ipc_tx_fir_s
{
    uint8_t lut_mode;
    uint8_t swing;
    int16_t fir_tap[7];
    uint16_t inner_eye1;
    uint16_t inner_eye2;
};
```

And the calling flow on API is just simple as below:

```
inphi_status_t plr_tx_fir_set(
    uint32_t pdie,
    uint32_t channel,
    enum e_plr_intf intf,
    struct plr_tx_fir_s *fir)
{
    plr_ipc_tx_fir_t fw_fir;

    fw_fir.lut_mode = fir->lut_mode;
    fw_fir.swing = fir->swing;
    INPHI_MEMCPY(fw_fir.fir_tap, fir->fir_tap, sizeof(fir->fir_tap));
    fw_fir.inner_eye1 = fir->inner_eye1;
    fw_fir.inner_eye2 = fir->inner_eye2;

    return plr_ipc_chn_attr_set(pdie, channel, intf, PLR_IPC_CHN_ATTR_ID_TX_FIR, (uint8_t *)&fw_fir, sizeof
(plr_ipc_tx_fir_t));
}

inphi_status_t plr_tx_fir_get(
    uint32_t pdie,
    uint32_t channel,
    enum e_plr_intf intf,
    struct plr_tx_fir_s *fir)
{
    inphi_status_t status;
    plr_ipc_tx_fir_t fw_fir;

    status = plr_ipc_chn_attr_get(pdie, channel, intf, PLR_IPC_CHN_ATTR_ID_TX_FIR, (uint8_t *)&fw_fir, sizeof
(plr_ipc_tx_fir_t));
    if (status != INPHI_OK)
        return status;

    /* FIR returned from FW */
    fir->lut_mode = fw_fir.lut_mode;
    fir->swing = fw_fir.swing;
    INPHI_MEMCPY(fir->fir_tap, fw_fir.fir_tap, sizeof(fir->fir_tap));
    fir->inner_eye1 = fw_fir.inner_eye1;
    fir->inner_eye2 = fw_fir.inner_eye2;

    return INPHI_OK;
}
```

Ring HAL implementation

Below is an example of ring HAL implementation taken from Mira.

```
/* Ring HAL */
typedef struct plr_ring_hal_s
{
    struct ring_hal hal; /* Generic ring HAL */
    uint32_t die;        /* Die to work on */
}plr_ring_hal_t;

static uint16_t plr_ring_hal__alignment(struct ring_hal *self)
{
    return 4;
}

static uint32_t plr_ring_hal__read(struct ring_hal *hal, uint32_t address)
{
    struct plr_ring_hal_s *self = (struct plr_ring_hal_s *)hal;
    uint32_t value;

    if (RING_HAL_IS_MDD30_ADDR(address))
        return plr_reg_read(self->die, RING_HAL_MMD30_ADDR(address));

    if (plr_mcu_pif_read(self->die, address, &value, 1) == INPHI_OK)
```

```
        return value;

    return 0xDEADCAFE;
}

static void plr_ring_hal__write(struct ring_hal *hal, uint32_t address, uint32_t value)
{
    struct plr_ring_hal_s *self = (struct plr_ring_hal_s *)hal;

    if (RING_HAL_IS_MDD30_ADDR(address))
        plr_reg_write(self->die, RING_HAL_MMD30_ADDR(address), value);

    else
        plr_mcu_pif_write(self->die, address, &value, 1);
}

static uint16_t plr_ring_hal__block_read(struct ring_hal *hal, uint32_t start_addr, uint8_t *buffer, uint16_t
len)
{
    struct plr_ring_hal_s *self = (struct plr_ring_hal_s *)hal;
    uint16_t num_bytes = RING_ALIGN_FLOOR(len, RING_HAL_PIF_ALIGNMENT);
    uint16_t num_words = num_bytes / RING_HAL_PIF_ALIGNMENT;
    inphi_status_t status = plr_mcu_pif_read(self->die, start_addr, (uint32_t*)buffer, num_words);
    return (status == INPHI_OK) ? num_bytes : 0;
}

static uint16_t plr_ring_hal__block_write(struct ring_hal *hal, uint32_t start_addr, uint8_t *buffer, uint16_t
len)
{
    struct plr_ring_hal_s *self = (struct plr_ring_hal_s *)hal;
    uint16_t num_bytes = RING_ALIGN_FLOOR(len, RING_HAL_PIF_ALIGNMENT);
    uint16_t num_words = num_bytes / RING_HAL_PIF_ALIGNMENT;
    inphi_status_t status =  plr_mcu_pif_write(self->die, start_addr, (uint32_t*)buffer, num_words);
    return (status == INPHI_OK) ? num_bytes : 0;
}

static struct ring_hal_ops plr_ring_hal_ops =
{
    .alignment = plr_ring_hal__alignment,
    .read = plr_ring_hal__read,
    .write = plr_ring_hal__write,
    .block_read = plr_ring_hal__block_read,
    .block_write = plr_ring_hal__block_write
};

struct ring_hal *plr_ring_hal(struct plr_ring_hal_s *self, uint32_t die)
{
    /* Super constructor */
    memset(self, 0, sizeof(struct plr_ring_hal_s));
    ring_hal((struct ring_hal *)self);

    /* Die and access operations */
    self->die = die;
    self->hal.ops = &plr_ring_hal_ops;

    return (struct ring_hal *) self;
}
```

IPC repository

IPC library can be found at http://las-gitlab/hsc/libfw_api

Mira integration example

This IPC is integrated to Mira project on the branch **`users/nanguyen/ipc`**. Below are steps to fetch code (by MSYS or Cygwin console)

```
# Assume the WORKING_DIR points to your top working directory
# e.g: export WORKING_DIR=/c/usr/nanguyen/git/
export WORKING_DIR=<your working directory>

# 3rdparty and bin repos
git clone git@las-gitlab:hsc/3rdparty.git $WORKING_DIR/3rdparty
git clone git@las-gitlab:hsc/bin.git $WORKING_DIR/bin

# Regression repo to have SRC C-API (such as polaris, porrima, ...)
git clone git@las-gitlab:hsc/regression.git $WORKING_DIR/hsc/regression

# Mira repo
git clone git@las-gitlab:hsc/mira.git $WORKING_DIR/hsc/mira

# LAB scripts
git clone git@las-gitlab:hsc/lab.git $WORKING_DIR/hsc/lab

# Standing on right branches
cd $WORKING_DIR/bin && git checkout users/nanguyen/ipc
cd $WORKING_DIR/hsc/mira && git checkout users/nanguyen/ipc && git submodule update --init --recursive
cd $WORKING_DIR/hsc/regression && git checkout mira_regression && git submodule update --init
```

The SCL3-5814 can be used to build C-API locally by Windows CMD prompt, as below example:

```
# Setup environment variables to make sure that we have all tools
# e.g. set WORKING_DIR=c:\usr\nanguyen\git
set WORKING_DIR=<your working directory>
set PATH_VISUAL_STUDIO=C:\Program Files (x86)\Microsoft Visual Studio 12.0
set PATH_VCVARS=C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\bin\vcvars32.bat
set PATH=%WORKING%\3rdparty\win32;%PATH%
set PATH_SWIG=C:\usr\tools\swig\swig.exe

# Build C-API for PLR2 (as we will work on PLR2 bench, the SCL3-5814)
make -C %WORKING_DIR%/hsc/mira/api clean
make -C %WORKING_DIR%/hsc/mira/api asic=plr2_11x12 python
```

FW can be built by:

- Using Linux. Then use `scp` command to transfer the image to the bench. And use GUI to load it.
- Use Xtensa Xplorer to build and run FW directly on this IDE.

The follow sequence is to run the test script to see how it works, and its benchmark. Run below on MSYS or Git Bash console.

```
# Assume the WORKING_DIR points to your top working directory
# e.g: export WORKING_DIR=/c/usr/nanguyen/git/
export WORKING_DIR=<your working directory>

# Have all of SRC C-APIs
cd $WORKING_DIR/hsc/regression/mira && python -u make.py get_all --asics polaris porrima mira_plr2_11x12_dd \
    --plr_fw_version=1.11.971       --plr_api_version=1.11.865 \
    --por_fw_version=1.12.1343      --por_api_version=1.12.1177 \
    --mira_fw_version=0.12.0.1194   --mira_api_version=0.12.0.1270

# Setup PYTHONPATH to use C-API that we have just built
export PYTHONPATH=$WORKING_DIR/hsc/mira/api/build-output/python:$WORKING_DIR/hsc/regression/mira:$WORKING_DIR
/hsc/lab/users/nanguyen/mira:/c/INPHI/Olympus/Python

# Run the test script
python $WORKING_DIR/hsc/lab/users/nanguyen/mira/tests/test_ipc.py
```

The other nice option is to run the `test_ipc.py` script on Pycharm IDE.