DMA Engine API Guide

Vinod Koul <vinod dot koul at intel.com>

> **Note**
>
> For DMA Engine usage in async_tx please see: `Documentation/crypto/async-tx-api.rst`

Below is a guide to device driver writers on how to use the Slave-DMA API of the DMA Engine. This is applicable only for slave DMA usage only.

DMA usage

The slave DMA usage consists of following steps:

- Allocate a DMA slave channel
- Set slave and controller specific parameters
- Get a descriptor for transaction
- Submit the transaction
- Issue pending requests and wait for callback notification

The details of these operations are:

1. Allocate a DMA slave channel

   Channel allocation is slightly different in the slave DMA context, client drivers typically need a channel from a particular DMA controller only and even in some cases a specific channel is desired. To request a channel dma_request_chan() API is used.

   Interface:

   ```
   struct dma_chan *dma_request_chan(struct device *dev, const char *name);
   ```

   Which will find and return the `name` DMA channel associated with the 'dev' device. The association is done via DT, ACPI or board file based dma_slave_map matching table.

   A channel allocated via this interface is exclusive to the caller, until dma_release_channel() is called.

2. Set slave and controller specific parameters

   Next step is always to pass some specific information to the DMA driver. Most of the generic information which a slave DMA can use is in struct dma_slave_config. This allows the clients to specify DMA direction, DMA addresses, bus widths, DMA burst lengths etc for the peripheral.

   If some DMA controllers have more parameters to be sent then they should try to embed struct dma_slave_config in their controller specific structure. That gives flexibility to client to pass more parameters, if required.

   Interface:

   ```
   int dmaengine_slave_config(struct dma_chan *chan,
                              struct dma_slave_config *config)
   ```

   Please see the dma_slave_config structure definition in dmaengine.h for a detailed explanation of the struct members. Please note that the 'direction' member will be going away as it duplicates the direction given in the prepare call.

3. Get a descriptor for transaction

   For slave usage the various modes of slave transfers supported by the DMA-engine are:

   - slave_sg: DMA a list of scatter gather buffers from/to a peripheral
   - dma_cyclic: Perform a cyclic DMA operation from/to a peripheral till the operation is explicitly stopped.
   - interleaved_dma: This is common to Slave as well as M2M clients. For slave address of devices' fifo could be already known to the driver. Various types of operations could be expressed by setting appropriate values to the 'dma_interleaved_template' members. Cyclic interleaved DMA transfers are also possible if supported by the channel by setting the DMA_PREP_REPEAT transfer flag.

   A non-NULL return of this transfer API represents a "descriptor" for the given transaction.

   Interface:

   ```
   struct dma_async_tx_descriptor *dmaengine_prep_slave_sg(
              struct dma_chan *chan, struct scatterlist *sgl,
              unsigned int sg_len, enum dma_data_direction direction,
              unsigned long flags);
   ```

```
struct dma_async_tx_descriptor *dmaengine_prep_dma_cyclic(
          struct dma_chan *chan, dma_addr_t buf_addr, size_t buf_len,
          size_t period_len, enum dma_data_direction direction);

struct dma_async_tx_descriptor *dmaengine_prep_interleaved_dma(
          struct dma_chan *chan, struct dma_interleaved_template *xt,
          unsigned long flags);
```

The peripheral driver is expected to have mapped the scatterlist for the DMA operation prior to calling dmaengine_prep_slave_sg(), and must keep the scatterlist mapped until the DMA operation has completed. The scatterlist must be mapped using the DMA struct device. If a mapping needs to be synchronized later, dma_sync_*_for_*() must be called using the DMA struct device, too. So, normal setup should look like this:

```
struct device *dma_dev = dmaengine_get_dma_device(chan);

nr_sg = dma_map_sg(dma_dev, sgl, sg_len);
   if (nr_sg == 0)
          /* error */

   desc = dmaengine_prep_slave_sg(chan, sgl, nr_sg, direction, flags);
```

Once a descriptor has been obtained, the callback information can be added and the descriptor must then be submitted. Some DMA engine drivers may hold a spinlock between a successful preparation and submission so it is important that these two operations are closely paired.

> **Note**
>
> Although the async_tx API specifies that completion callback routines cannot submit any new operations, this is not the case for slave/cyclic DMA.
>
> For slave DMA, the subsequent transaction may not be available for submission prior to callback function being invoked, so slave DMA callbacks are permitted to prepare and submit a new transaction.
>
> For cyclic DMA, a callback function may wish to terminate the DMA via dmaengine_terminate_async().
>
> Therefore, it is important that DMA engine drivers drop any locks before calling the callback function which may cause a deadlock.
>
> Note that callbacks will always be invoked from the DMA engines tasklet, never from interrupt context.

Optional: per descriptor metadata

DMAengine provides two ways for metadata support.

DESC_METADATA_CLIENT

The metadata buffer is allocated/provided by the client driver and it is attached to the descriptor.

```
int dmaengine_desc_attach_metadata(struct dma_async_tx_descriptor *desc,
                                   void *data, size_t len);
```

DESC_METADATA_ENGINE

The metadata buffer is allocated/managed by the DMA driver. The client driver can ask for the pointer, maximum size and the currently used size of the metadata and can directly update or read it.

Becasue the DMA driver manages the memory area containing the metadata, clients must make sure that they do not try to access or get the pointer after their transfer completion callback has run for the descriptor. If no completion callback has been defined for the transfer, then the metadata must not be accessed after issue_pending. In other words: if the aim is to read back metadata after the transfer is completed, then the client must use completion callback.

```
void *dmaengine_desc_get_metadata_ptr(struct dma_async_tx_descriptor *desc,
          size_t *payload_len, size_t *max_len);

int dmaengine_desc_set_metadata_len(struct dma_async_tx_descriptor *desc,
          size_t payload_len);
```

Client drivers can query if a given mode is supported with:

```
bool dmaengine_is_metadata_mode_supported(struct dma_chan *chan,
          enum dma_desc_metadata_mode mode);
```

Depending on the used mode client drivers must follow different flow.

DESC_METADATA_CLIENT

- DMA_MEM_TO_DEV / DEV_MEM_TO_MEM:
    1. prepare the descriptor (dmaengine_prep_*) construct the metadata in the client's buffer
    2. use dmaengine_desc_attach_metadata() to attach the buffer to the descriptor
    3. submit the transfer
- DMA_DEV_TO_MEM:
    1. prepare the descriptor (dmaengine_prep_*)
    2. use dmaengine_desc_attach_metadata() to attach the buffer to the descriptor
    3. submit the transfer
    4. when the transfer is completed, the metadata should be available in the attached buffer

DESC_METADATA_ENGINE

- DMA_MEM_TO_DEV / DEV_MEM_TO_MEM:
    1. prepare the descriptor (dmaengine_prep_*)
    2. use dmaengine_desc_get_metadata_ptr() to get the pointer to the engine's metadata area
    3. update the metadata at the pointer
    4. use dmaengine_desc_set_metadata_len() to tell the DMA engine the amount of data the client has placed into the metadata buffer
    5. submit the transfer
- DMA_DEV_TO_MEM:
    1. prepare the descriptor (dmaengine_prep_*)
    2. submit the transfer
    3. on transfer completion, use dmaengine_desc_get_metadata_ptr() to get the pointer to the engine's metadata area
    4. read out the metadata from the pointer

> **Note**
>
> When DESC_METADATA_ENGINE mode is used the metadata area for the descriptor is no longer valid after the transfer has been completed (valid up to the point when the completion callback returns if used).
>
> Mixed use of DESC_METADATA_CLIENT / DESC_METADATA_ENGINE is not allowed, client drivers must use either of the modes per descriptor.

4. Submit the transaction

   Once the descriptor has been prepared and the callback information added, it must be placed on the DMA engine drivers pending queue.

   Interface:

   ```
   dma_cookie_t dmaengine_submit(struct dma_async_tx_descriptor *desc)
   ```

   This returns a cookie can be used to check the progress of DMA engine activity via other DMA engine calls not covered in this document.

   dmaengine_submit() will not start the DMA operation, it merely adds it to the pending queue. For this, see step 5, dma_async_issue_pending.

   > **Note**
   >
   > After calling `dmaengine_submit()` the submitted transfer descriptor (`struct dma_async_tx_descriptor`) belongs to the DMA engine. Consequently, the client must consider invalid the pointer to that descriptor.

5. Issue pending DMA requests and wait for callback notification

   The transactions in the pending queue can be activated by calling the issue_pending API. If channel is idle then the first transaction in queue is started and subsequent ones queued up.

   On completion of each DMA operation, the next in queue is started and a tasklet triggered. The tasklet will then call the client driver completion callback routine for notification, if set.

   Interface:

   ```
   void dma_async_issue_pending(struct dma_chan *chan);
   ```

Further APIs

1. Terminate APIs

```
int dmaengine_terminate_sync(struct dma_chan *chan)
int dmaengine_terminate_async(struct dma_chan *chan)
int dmaengine_terminate_all(struct dma_chan *chan) /* DEPRECATED */
```

This causes all activity for the DMA channel to be stopped, and may discard data in the DMA FIFO which hasn't been fully transferred. No callback functions will be called for any incomplete transfers.

Two variants of this function are available.

dmaengine_terminate_async() might not wait until the DMA has been fully stopped or until any running complete callbacks have finished. But it is possible to call dmaengine_terminate_async() from atomic context or from within a complete callback. dmaengine_synchronize() must be called before it is safe to free the memory accessed by the DMA transfer or free resources accessed from within the complete callback.

dmaengine_terminate_sync() will wait for the transfer and any running complete callbacks to finish before it returns. But the function must not be called from atomic context or from within a complete callback.

dmaengine_terminate_all() is deprecated and should not be used in new code.

2. Pause API

```
int dmaengine_pause(struct dma_chan *chan)
```

This pauses activity on the DMA channel without data loss.

3. Resume API

```
int dmaengine_resume(struct dma_chan *chan)
```

Resume a previously paused DMA channel. It is invalid to resume a channel which is not currently paused.

4. Check Txn complete

```
enum dma_status dma_async_is_tx_complete(struct dma_chan *chan,
          dma_cookie_t cookie, dma_cookie_t *last, dma_cookie_t *used)
```

This can be used to check the status of the channel. Please see the documentation in include/linux/dmaengine.h for a more complete description of this API.

This can be used in conjunction with dma_async_is_complete() and the cookie returned from dmaengine_submit() to check for completion of a specific DMA transaction.

> **Note**
>
> Not all DMA engine drivers can return reliable information for a running DMA channel. It is recommended that DMA engine users pause or stop (via dmaengine_terminate_all()) the channel before using this API.

5. Synchronize termination API

```
void dmaengine_synchronize(struct dma_chan *chan)
```

Synchronize the termination of the DMA channel to the current context.

This function should be used after dmaengine_terminate_async() to synchronize the termination of the DMA channel to the current context. The function will wait for the transfer and any running complete callbacks to finish before it returns.

If dmaengine_terminate_async() is used to stop the DMA channel this function must be called before it is safe to free memory accessed by previously submitted descriptors or to free any resources accessed within the complete callback of previously submitted descriptors.

The behavior of this function is undefined if dma_async_issue_pending() has been called between dmaengine_terminate_async() and this function.