DMAengine controller documentation

Hardware Introduction

Most of the Slave DMA controllers have the same general principles of operations.

They have a given number of channels to use for the DMA transfers, and a given number of requests lines.

Requests and channels are pretty much orthogonal. Channels can be used to serve several to any requests. To simplify, channels are the entities that will be doing the copy, and requests what endpoints are involved.

The request lines actually correspond to physical lines going from the DMA-eligible devices to the controller itself. Whenever the device will want to start a transfer, it will assert a DMA request (DRQ) by asserting that request line.

A very simple DMA controller would only take into account a single parameter: the transfer size. At each clock cycle, it would transfer a byte of data from one buffer to another, until the transfer size has been reached.

That wouldn't work well in the real world, since slave devices might require a specific number of bits to be transferred in a single cycle. For example, we may want to transfer as much data as the physical bus allows to maximize performances when doing a simple memory copy operation, but our audio device could have a narrower FIFO that requires data to be written exactly 16 or 24 bits at a time. This is why most if not all of the DMA controllers can adjust this, using a parameter called the transfer width.

Moreover, some DMA controllers, whenever the RAM is used as a source or destination, can group the reads or writes in memory into a buffer, so instead of having a lot of small memory accesses, which is not really efficient, you'll get several bigger transfers. This is done using a parameter called the burst size, that defines how many single reads/writes it's allowed to do without the controller splitting the transfer into smaller sub-transfers.

Our theoretical DMA controller would then only be able to do transfers that involve a single contiguous block of data. However, some of the transfers we usually have are not, and want to copy data from non-contiguous buffers to a contiguous buffer, which is called scatter-gather.

DMAEngine, at least for mem2dev transfers, require support for scatter-gather. So we're left with two cases here: either we have a quite simple DMA controller that doesn't support it, and we'll have to implement it in software, or we have a more advanced DMA controller, that implements in hardware scatter-gather.

The latter are usually programmed using a collection of chunks to transfer, and whenever the transfer is started, the controller will go over that collection, doing whatever we programmed there.

This collection is usually either a table or a linked list. You will then push either the address of the table and its number of elements, or the first item of the list to one channel of the DMA controller, and whenever a DRQ will be asserted, it will go through the collection to know where to fetch the data from.

Either way, the format of this collection is completely dependent on your hardware. Each DMA controller will require a different structure, but all of them will require, for every chunk, at least the source and destination addresses, whether it should increment these addresses or not and the three parameters we saw earlier: the burst size, the transfer width and the transfer size.

The one last thing is that usually, slave devices won't issue DRQ by default, and you have to enable this in your slave device driver first whenever you're willing to use DMA.

These were just the general memory-to-memory (also called mem2mem) or memory-to-device (mem2dev) kind of transfers. Most devices often support other kind of transfers or memory operations that dmaengine support and will be detailed later in this document.

DMA Support in Linux

Historically, DMA controller drivers have been implemented using the async TX API, to offload operations such as memory copy, XOR, cryptography, etc., basically any memory to memory operation.

Over time, the need for memory to device transfers arose, and dmaengine was extended. Nowadays, the async TX API is written as a layer on top of dmaengine, and acts as a client. Still, dmaengine accommodates that API in some cases, and made some design choices to ensure that it stayed compatible.

For more information on the Async TX API, please look the relevant documentation file in Documentation/crypto/async-tx-api.rst.

DMAEngine APIs

`struct dma_device` Initialization

Just like any other kernel framework, the whole DMAEngine registration relies on the driver filling a structure and registering against the framework. In our case, that structure is dma_device.

The first thing you need to do in your driver is to allocate this structure. Any of the usual memory allocators will do, but you'll also

need to initialize a few fields in there:

- `channels`: should be initialized as a list using the INIT_LIST_HEAD macro for example
- `src_addr_widths`: should contain a bitmask of the supported source transfer width
- `dst_addr_widths`: should contain a bitmask of the supported destination transfer width
- `directions`: should contain a bitmask of the supported slave directions (i.e. excluding mem2mem transfers)
- `residue_granularity`: granularity of the transfer residue reported to dma_set_residue. This can be either:
    - Descriptor: your device doesn't support any kind of residue reporting. The framework will only know that a particular transaction descriptor is done.
    - Segment: your device is able to report which chunks have been transferred
    - Burst: your device is able to report which burst have been transferred
- `dev`: should hold the pointer to the `struct device` associated to your current driver instance.

Supported transaction types

The next thing you need is to set which transaction types your device (and driver) supports.

Our `dma_device structure` has a field called cap_mask that holds the various types of transaction supported, and you need to modify this mask using the dma_cap_set function, with various flags depending on transaction types you support as an argument.

All those capabilities are defined in the `dma_transaction_type enum`, in `include/linux/dmaengine.h`

Currently, the types available are:

- DMA_MEMCPY
    - The device is able to do memory to memory copies
-
    - DMA_MEMCPY_SG
    - The device supports memory to memory scatter-gather transfers.
    - Even though a plain memcpy can look like a particular case of a scatter-gather transfer, with a single chunk to copy, it's a distinct transaction type in the mem2mem transfer case. This is because some very simple devices might be able to do contiguous single-chunk memory copies, but have no support for more complex SG transfers.
    - No matter what the overall size of the combined chunks for source and destination is, only as many bytes as the smallest of the two will be transmitted. That means the number and size of the scatter-gather buffers in both lists need not be the same, and that the operation functionally is equivalent to a `strncpy` where the `count` argument equals the smallest total size of the two scatter-gather list buffers.
    - It's usually used for copying pixel data between host memory and memory-mapped GPU device memory, such as found on modern PCI video graphics cards. The most immediate example is the OpenGL API function `glReadPielx()`, which might require a verbatim copy of a huge framebuffer from local device memory onto host memory.
- DMA_XOR
    - The device is able to perform XOR operations on memory areas
    - Used to accelerate XOR intensive tasks, such as RAID5
- DMA_XOR_VAL
    - The device is able to perform parity check using the XOR algorithm against a memory buffer.
- DMA_PQ
    - The device is able to perform RAID6 P+Q computations, P being a simple XOR, and Q being a Reed-Solomon algorithm.
- DMA_PQ_VAL
    - The device is able to perform parity check using RAID6 P+Q algorithm against a memory buffer.
- DMA_INTERRUPT
    - The device is able to trigger a dummy transfer that will generate periodic interrupts
    - Used by the client drivers to register a callback that will be called on a regular basis through the DMA controller interrupt
- DMA_PRIVATE
    - The devices only supports slave transfers, and as such isn't available for async transfers.
- DMA_ASYNC_TX
    - Must not be set by the device, and will be set by the framework if needed
    - TODO: What is it about?
- DMA_SLAVE
    - The device can handle device to memory transfers, including scatter-gather transfers.
    - While in the mem2mem case we were having two distinct types to deal with a single chunk to copy or a collection of them, here, we just have a single transaction type that is supposed to handle both.
    - If you want to transfer a single contiguous memory buffer, simply build a scatter list with only one item.
- DMA_CYCLIC
    - The device can handle cyclic transfers.
    - A cyclic transfer is a transfer where the chunk collection will loop over itself, with the last item pointing to the first.
    - It's usually used for audio transfers, where you want to operate on a single ring buffer that you will fill with your audio data.
- DMA_INTERLEAVE

    - The device supports interleaved transfer.
    - These transfers can transfer data from a non-contiguous buffer to a non-contiguous buffer, opposed to DMA_SLAVE that can transfer data from a non-contiguous data set to a continuous destination buffer.
    - It's usually used for 2d content transfers, in which case you want to transfer a portion of uncompressed data directly to the display to print it
- DMA_COMPLETION_NO_ORDER
    - The device does not support in order completion.
    - The driver should return DMA_OUT_OF_ORDER for device_tx_status if the device is setting this capability.
    - All cookie tracking and checking API should be treated as invalid if the device exports this capability.
    - At this point, this is incompatible with polling option for dmatest.
    - If this cap is set, the user is recommended to provide an unique identifier for each descriptor sent to the DMA device in order to properly track the completion.
- DMA_REPEAT
    - The device supports repeated transfers. A repeated transfer, indicated by the DMA_PREP_REPEAT transfer flag, is similar to a cyclic transfer in that it gets automatically repeated when it ends, but can additionally be replaced by the client.
    - This feature is limited to interleaved transfers, this flag should thus not be set if the DMA_INTERLEAVE flag isn't set. This limitation is based on the current needs of DMA clients, support for additional transfer types should be added in the future if and when the need arises.
- DMA_LOAD_EOT
    - The device supports replacing repeated transfers at end of transfer (EOT) by queuing a new transfer with the DMA_PREP_LOAD_EOT flag set.
    - Support for replacing a currently running transfer at another point (such as end of burst instead of end of transfer) will be added in the future based on DMA clients needs, if and when the need arises.

These various types will also affect how the source and destination addresses change over time.

Addresses pointing to RAM are typically incremented (or decremented) after each transfer. In case of a ring buffer, they may loop (DMA_CYCLIC). Addresses pointing to a device's register (e.g. a FIFO) are typically fixed.

Per descriptor metadata support

Some data movement architecture (DMA controller and peripherals) uses metadata associated with a transaction. The DMA controller role is to transfer the payload and the metadata alongside. The metadata itself is not used by the DMA engine itself, but it contains parameters, keys, vectors, etc for peripheral or from the peripheral.

The DMAengine framework provides a generic ways to facilitate the metadata for descriptors. Depending on the architecture the DMA driver can implement either or both of the methods and it is up to the client driver to choose which one to use.

- DESC_METADATA_CLIENT

  The metadata buffer is allocated/provided by the client driver and it is attached (via the dmaengine_desc_attach_metadata() helper to the descriptor.

  From the DMA driver the following is expected for this mode:

    - DMA_MEM_TO_DEV / DEV_MEM_TO_MEM

      The data from the provided metadata buffer should be prepared for the DMA controller to be sent alongside of the payload data. Either by copying to a hardware descriptor, or highly coupled packet.

    - DMA_DEV_TO_MEM

      On transfer completion the DMA driver must copy the metadata to the client provided metadata buffer before notifying the client about the completion. After the transfer completion, DMA drivers must not touch the metadata buffer provided by the client.

- 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. dmaengine_desc_get_metadata_ptr() and dmaengine_desc_set_metadata_len() is provided as helper functions.

  From the DMA driver the following is expected for this mode:

    - get_metadata_ptr()

      Should return a pointer for the metadata buffer, the maximum size of the metadata buffer and the currently used / valid (if any) bytes in the buffer.

    - set_metadata_len()

      It is called by the clients after it have placed the metadata to the buffer to let the DMA driver know the number of valid bytes provided.

  Note: since the client will ask for the metadata pointer in the completion callback (in DMA_DEV_TO_MEM case) the DMA

driver must ensure that the descriptor is not freed up prior the callback is called.

Device operations

Our dma_device structure also requires a few function pointers in order to implement the actual logic, now that we described what operations we were able to perform.

The functions that we have to fill in there, and hence have to implement, obviously depend on the transaction types you reported as supported.

- `device_alloc_chan_resources`
- `device_free_chan_resources`
  - These functions will be called whenever a driver will call `dma_request_channel` or `dma_release_channel` for the first/last time on the channel associated to that driver.
  - They are in charge of allocating/freeing all the needed resources in order for that channel to be useful for your driver.
  - These functions can sleep.
- `device_prep_dma_*`
  - These functions are matching the capabilities you registered previously.
  - These functions all take the buffer or the scatterlist relevant for the transfer being prepared, and should create a hardware descriptor or a list of hardware descriptors from it
  - These functions can be called from an interrupt context
  - Any allocation you might do should be using the GFP_NOWAIT flag, in order not to potentially sleep, but without depleting the emergency pool either.
  - Drivers should try to pre-allocate any memory they might need during the transfer setup at probe time to avoid putting to much pressure on the nowait allocator.
  - It should return a unique instance of the `dma_async_tx_descriptor structure`, that further represents this particular transfer.
  - This structure can be initialized using the function `dma_async_tx_descriptor_init`.
  - You'll also need to set two fields in this structure:
    - flags: TODO: Can it be modified by the driver itself, or should it be always the flags passed in the arguments
    - tx_submit: A pointer to a function you have to implement, that is supposed to push the current transaction descriptor to a pending queue, waiting for issue_pending to be called.
  - In this structure the function pointer callback_result can be initialized in order for the submitter to be notified that a transaction has completed. In the earlier code the function pointer callback has been used. However it does not provide any status to the transaction and will be deprecated. The result structure defined as `dmaengine_result` that is passed in to callback_result has two fields:
    - result: This provides the transfer result defined by `dmaengine_tx_result`. Either success or some error condition.
    - residue: Provides the residue bytes of the transfer for those that support residue.
- `device_issue_pending`
  - Takes the first transaction descriptor in the pending queue, and starts the transfer. Whenever that transfer is done, it should move to the next transaction in the list.
  - This function can be called in an interrupt context
- `device_tx_status`
  - Should report the bytes left to go over on the given channel
  - Should only care about the transaction descriptor passed as argument, not the currently active one on a given channel
  - The tx_state argument might be NULL
  - Should use dma_set_residue to report it
  - In the case of a cyclic transfer, it should only take into account the current period.
  - Should return DMA_OUT_OF_ORDER if the device does not support in order completion and is completing the operation out of order.
  - This function can be called in an interrupt context.
- device_config
  - Reconfigures the channel with the configuration given as argument
  - This command should NOT perform synchronously, or on any currently queued transfers, but only on subsequent ones
  - In this case, the function will receive a `dma_slave_config` structure pointer as an argument, that will detail which configuration to use.
  - Even though that structure contains a direction field, this field is deprecated in favor of the direction argument given to the prep_* functions
  - This call is mandatory for slave operations only. This should NOT be set or expected to be set for memcpy operations. If a driver support both, it should use this call for slave operations only and not for memcpy ones.
- device_pause
  - Pauses a transfer on the channel
  - This command should operate synchronously on the channel, pausing right away the work of the given channel
- device_resume
  - Resumes a transfer on the channel
  - This command should operate synchronously on the channel, resuming right away the work of the given channel

- device_terminate_all
  - Aborts all the pending and ongoing transfers on the channel
  - For aborted transfers the complete callback should not be called
  - Can be called from atomic context or from within a complete callback of a descriptor. Must not sleep. Drivers must be able to handle this correctly.
  - Termination may be asynchronous. The driver does not have to wait until the currently active transfer has completely stopped. See device_synchronize.
- device_synchronize
  - Must synchronize the termination of a channel to the current context.
  - Must make sure that memory for previously submitted descriptors is no longer accessed by the DMA controller.
  - Must make sure that all complete callbacks for previously submitted descriptors have finished running and none are scheduled to run.
  - May sleep.

Misc notes

(stuff that should be documented, but don't really know where to put them)

`dma_run_dependencies`

- Should be called at the end of an async TX transfer, and can be ignored in the slave transfers case.
- Makes sure that dependent operations are run before marking it as complete.

dma_cookie_t

- it's a DMA transaction ID that will increment over time.
- Not really relevant any more since the introduction of `virt-dma` that abstracts it away.

DMA_CTRL_ACK

- If clear, the descriptor cannot be reused by provider until the client acknowledges receipt, i.e. has a chance to establish any dependency chains
- This can be acked by invoking async_tx_ack()
- If set, does not mean descriptor can be reused

DMA_CTRL_REUSE

- If set, the descriptor can be reused after being completed. It should not be freed by provider if this flag is set.
- The descriptor should be prepared for reuse by invoking `dmaengine_desc_set_reuse()` which will set DMA_CTRL_REUSE.
- `dmaengine_desc_set_reuse()` will succeed only when channel support reusable descriptor as exhibited by capabilities
- As a consequence, if a device driver wants to skip the `dma_map_sg()` and `dma_unmap_sg()` in between 2 transfers, because the DMA'd data wasn't used, it can resubmit the transfer right after its completion.
- Descriptor can be freed in few ways
  - Clearing DMA_CTRL_REUSE by invoking `dmaengine_desc_clear_reuse()` and submitting for last txn
  - Explicitly invoking `dmaengine_desc_free()`, this can succeed only when DMA_CTRL_REUSE is already set
  - Terminating the channel
- DMA_PREP_CMD
  - If set, the client driver tells DMA controller that passed data in DMA API is command data.
  - Interpretation of command data is DMA controller specific. It can be used for issuing commands to other peripherals/register reads/register writes for which the descriptor should be in different format from normal data descriptors.
- DMA_PREP_REPEAT
  - If set, the transfer will be automatically repeated when it ends until a new transfer is queued on the same channel with the DMA_PREP_LOAD_EOT flag. If the next transfer to be queued on the channel does not have the DMA_PREP_LOAD_EOT flag set, the current transfer will be repeated until the client terminates all transfers.
  - This flag is only supported if the channel reports the DMA_REPEAT capability.
- DMA_PREP_LOAD_EOT
  - If set, the transfer will replace the transfer currently being executed at the end of the transfer.
  - This is the default behaviour for non-repeated transfers, specifying DMA_PREP_LOAD_EOT for non-repeated transfers will thus make no difference.
  - When using repeated transfers, DMA clients will usually need to set the DMA_PREP_LOAD_EOT flag on all transfers, otherwise the channel will keep repeating the last repeated transfer and ignore the new transfers being queued. Failure to set DMA_PREP_LOAD_EOT will appear as if the channel was stuck on the previous transfer.
  - This flag is only supported if the channel reports the DMA_LOAD_EOT capability.

General Design Notes

Most of the DMAEngine drivers you'll see are based on a similar design that handles the end of transfer interrupts in the handler, but defer most work to a tasklet, including the start of a new transfer whenever the previous transfer ended.

This is a rather inefficient design though, because the inter-transfer latency will be not only the interrupt latency, but also the scheduling latency of the tasklet, which will leave the channel idle in between, which will slow down the global transfer rate.

You should avoid this kind of practice, and instead of electing a new transfer in your tasklet, move that part to the interrupt handler in order to have a shorter idle window (that we can't really avoid anyway).

Glossary

- Burst: A number of consecutive read or write operations that can be queued to buffers before being flushed to memory.
- Chunk: A contiguous collection of bursts
- Transfer: A collection of chunks (be it contiguous or not)