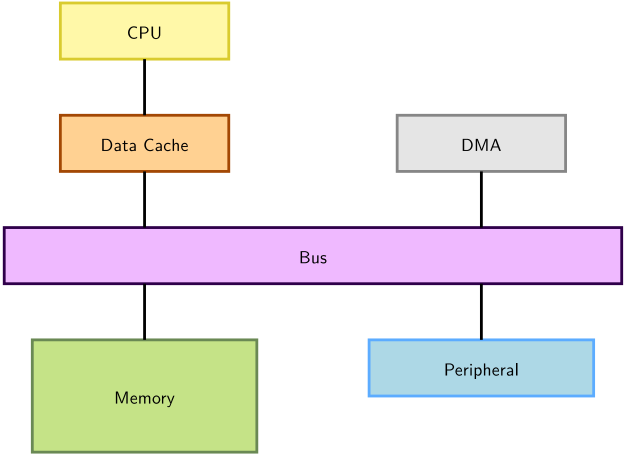
Direct Memory Access

DMA main principles

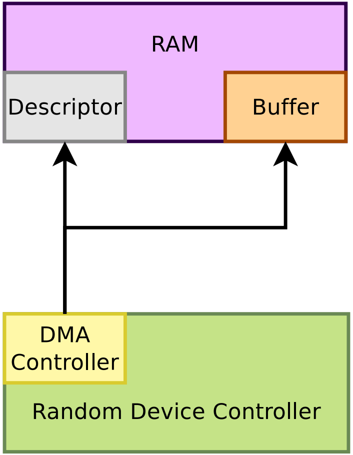
DMA integration

DMA (*Direct Memory Access*) is used to copy data directly between devices and RAM, without going through the CPU.



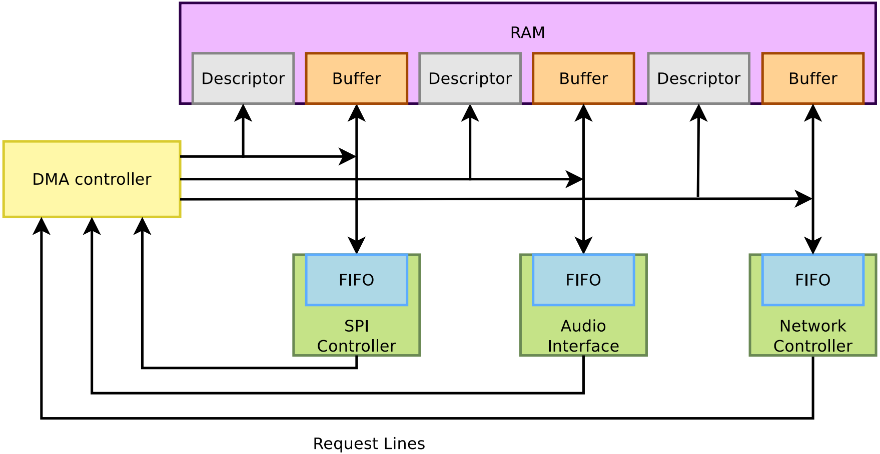
Peripheral DMA

Some device controllers embedded their own DMA controller and therefore can do DMA on their own.



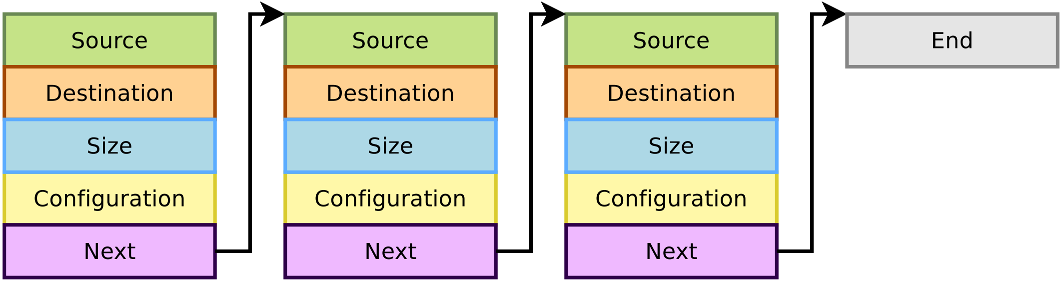
DMA controllers

Other device controllers rely on an external DMA controller (on the SoC). Their drivers need to submit DMA descriptors to this controller.



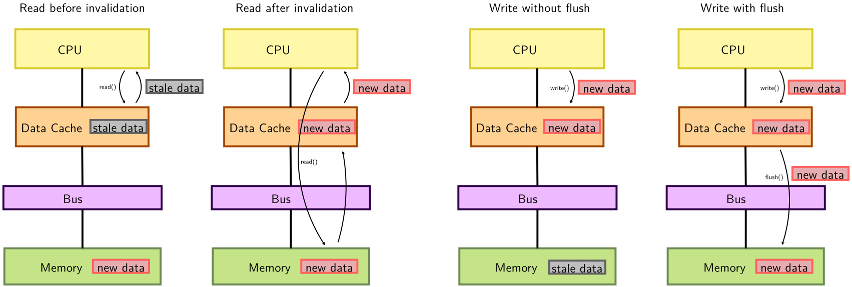
DMA descriptors

DMA descriptors describe the various attributes of a DMA transfer, and are chained.



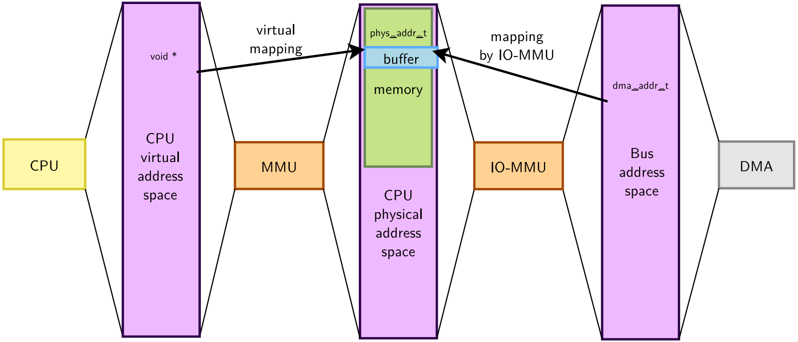
Cache constraints

- The CPU can access memory through a data cache
  - Using the cache can be more efficient (faster accesses to the cache than the bus)
- But the DMA does not access the CPU cache, so one needs to take care of cache coherency (cache content vs. memory content):
  - When the CPU reads from memory accessed by DMA, the relevant cache lines must be invalidated to force reading from memory again
  - When the CPU writes to memory before starting DMA transfers, the cache lines must be flushed/cleaned in order to force the data to reach the memory



DMA addressing constraints

- Memory and devices have physical addresses: `phys_addr_t`
- CPUs usually access memory through an MMU, using virtual pointers: `void *`
- DMA controllers do not access memory through the MMU and thus cannot manipulate virtual addresses, instead they access a `dma_addr_t` through either:
  - physical addresses directly
  - an IOMMU, in which case a specific mapping must be created



DMA memory allocation constraints

The APIs must remain generic and handle all cases transparently, hence:

- Each memory chunk accessed by the DMA shall be physically contiguous, which means one can use:
  - any memory allocated by `kmalloc()` (up to 128 KB)
  - any memory allocated by `__get_free_pages()` (up to 8MB)
  - block I/O and networking buffers, designed to support DMA
- Unless the buffer is smaller than one page, one cannot use:
  - kernel memory allocated with `vmalloc()`
  - user memory allocated with `malloc()`
    - Almost all the time userspace relies on the kernel to allocate the buffers and `mmap()` them to be usable from userspace (requires a dedicated user API)

Kernel APIs for DMA

dma-mapping vs. dmaengine vs. dma-buf

The dma-mapping API:

- Allocates and manages DMA buffers
- Offers generic interfaces to handle coherency
- Manages IO-MMU DMA mappings when relevant
- See core-api/dma-api and core-api/dma-api-howto

The dmaengine API:

- Abstracts the DMA controller
- Offers generic functions to configure, queue, trigger, stop transfers
- Unused when dealing with peripheral DMA
- See driver-api/dmaengine/client and

The dma-buf API:

- Enables sharing DMA buffers between devices within the kernel
- Not covered in this training

dma-mapping: Coherent or streaming DMA mappings

- Coherent mappings
  - The kernel allocates a suitable buffer and sets the mapping for the driver
  - Can simultaneously be accessed by the CPU and device
  - So, has to be in a cache coherent memory area
  - Usually allocated for the whole time the module is loaded
    - Can be expensive to setup and use on some platforms
    - Typically implemented by disabling cache on ARM
- Streaming mappings
  - Use an already allocated buffer
  - The driver provides a buffer, the kernel just sets the mapping
  - Mapping set up for each transfer (keeps DMA registers free on the hardware)

dma-mapping: memory addressing constraints

- The default addressing capability of the DMA controllers is assumed to be 32-bit.
- If the platform supports it, the DMA addressing capability can be:
  - increased (eg. need to access highmem)
  - decreased (eg. ISA devices, where kmalloc() buffers can also be allocated in the first part of the RAM with GFP_DMA)
- Linux stores this capability in a per-device mask, DMA mappings can fail because a buffer is out of reach
- In all cases, the DMA mask shall be consistent before allocating buffers

```
int dma_set_mask_and_coherent(struct device *dev, u64 mask)
```

- Maximum and optimal buffer sizes can also be retrieved to optimize allocations/buffer handling

```
size_t dma_max_mapping_size(struct device *dev);
size_t dma_opt_mapping_size(struct device *dev);
```

The kernel takes care of both buffer allocation and mapping:

```
#include <linux/dma-mapping.h>

void *                        /* Output: buffer address */
    dma_alloc_coherent(
         struct device *dev, /* device structure */
         size_t size,        /* Needed buffer size in bytes */
         dma_addr_t *handle, /* Output: DMA bus address */
         gfp_t gfp           /* Standard GFP flags */
);

void dma_free_coherent(struct device *dev,
    size_t size, void *cpu_addr, dma_addr_t handle);
```

Note: called *consistent mappings* on PCI
(`pci_alloc_consistent()` and `pci_free_consistent()`)

Works on already allocated buffers:

```
#include <linux/dma-mapping.h>

dma_addr_t dma_map_single(
      struct device *,        /* device structure */
      void *,                 /* input: buffer to use */
      size_t,                 /* buffer size */
      enum dma_data_direction /* Either DMA_BIDIRECTIONAL,
                               * DMA_TO_DEVICE or
                               * DMA_FROM_DEVICE */
);

void dma_unmap_single(struct device *dev, dma_addr_t handle,
    size_t size, enum dma_data_direction dir);
```

dma-mapping: Setting up streaming memory mappings (multiples)

A scatterlist using the scatter-gather library can be used to map several buffers and link them together

```
#include <linux/dma-mapping.h>
#include <linux/scatterlist.h>

struct scatterlist sglist[NENTS], *sg;
int i, count;

sg_init_table(sglist, NENTS);
sg_set_buf(&sglist[0], buf0, len0);
sg_set_buf(&sglist[1], buf1, len1);

count = dma_map_sg(dev, sglist, NENTS, DMA_TO_DEVICE);
for_each_sg(sglist, sg, count, i) {
        dma_address[i] = sg_dma_address(sg);
        dma_len[i] = sg_dma_len(sg);
}
...
dma_unmap_sg(dev sglist, count, DMA_TO_DEVICE);
```

dma-mapping: Setting up streaming I/O mappings

Physical addresses with MMIO registers shall always be remapped (otherwise it would not work when they are accessed through an IO-MMU)

```
#include <linux/dma-mapping.h>

dma_addr_t dma_map_resource(
      struct device *,          /* device structure */
      phys_addr_t,              /* input: resource to use */
      size_t,                   /* buffer size */
      enum dma_data_direction, /* Either DMA_BIDIRECTIONAL,
                                 * DMA_TO_DEVICE or
                                 * DMA_FROM_DEVICE */
      unsigned long attrs,      /* optional attributes */
);

void dma_unmap_resource(struct device *dev, dma_addr_t handle,
    size_t size, enum dma_data_direction dir, unsigned long attrs);
```

- All mapping helpers can fail and return errors
- The right way to check the validity of the returned dma_addr_t is to call:

```
int dma_mapping_error(struct device *dev, dma_addr_t dma_addr)
```

  - May give additional clues if CONFIG_DMA_API_DEBUG is enabled.

dma-mapping: Syncing streaming DMA mappings

- In general streaming mappings are:
  - mapped right before use with DMA
    - `MEM_TO_DEV`: caches are flushed
  - unmapped right after
    - `DEV_TO_MEM`: cache lines are invalidated
- The CPU shall only access the buffer after unmapping!
- If however the same memory region has to be used for several DMA transfers, the same mapping can be kept in place. In this case the data must be synchronized before CPU access:
  - The CPU needs to access the data:

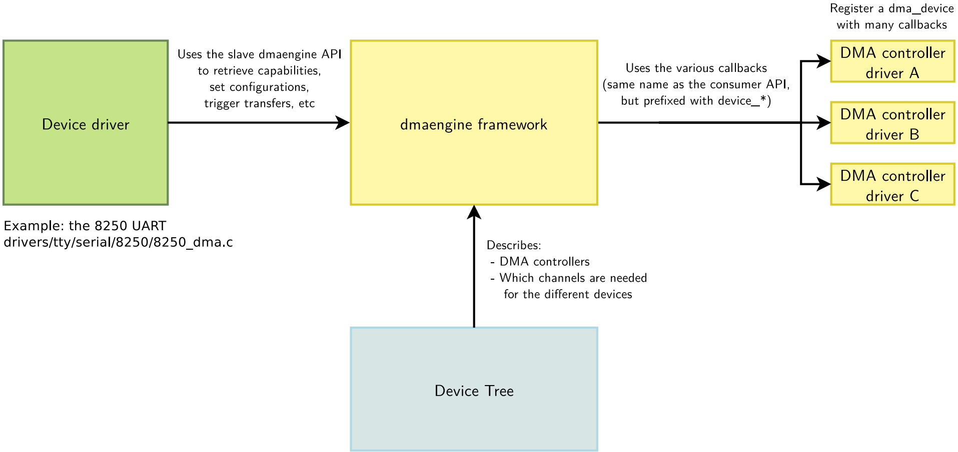
    ```
    dma_sync_single_for_cpu(dev, dma_handle, size, direction);
    dma_sync_sg_for_cpu(dev, sglist, nents, direction);
    ```
  - The device needs to access the data:

    ```
    dma_sync_single_for_device(dev, dma_handle, size, direction);
    dma_sync_sg_for_device(dev, sglist, nents, direction);
    ```

Starting DMA transfers

- If the device you're writing a driver for is doing peripheral DMA, no external API is involved.
- If it relies on an external DMA controller, you'll need to
  1. Ask the hardware to use DMA, so that it will drive its request line
  2. Use Linux `dmaengine` framework, especially its slave API

The `dmaengine` framework



dmaengine: Slave API: Initial configuration

Steps to start a DMA transfer with dmaengine:

1. Request a channel for exclusive use with dma_request_chan(), or one of its variants
   - This channel pointer will be used all along
   - Returns a pointer over a struct dma_chan which can also be an error pointer
2. Configure the engine by filling a struct dma_slave_config structure and passing it to dmaengine_slave_config():

```
struct dma_slave_config txconf = {};

/* Tell the engine what configuration we want on a given channel:
 * direction, access size, burst length, source and destination).
 * Source being memory, there is no buswidth or maxburst limitation
 * and each buffer will be different. */
txconf.direction = DMA_MEM_TO_DEV;
txconf.dst_addr_width = DMA_SLAVE_BUSWIDTH_1_BYTE;
txconf.dst_maxburst = TX_TRIGGER;
txconf.dst_addr = fifo_dma_addr;
ret = dmaengine_slave_config(dma->txchan, &txconf);
```

dmaengine: Slave API: Per-transfer configuration (1/2)

1. Create a descriptor with all the required configuration for the next transfer with:

```
struct dma_async_tx_descriptor *
dmaengine_prep_slave_single(struct dma_chan *chan, dma_addr_t buf,
                            size_t len, enum dma_transfer_direction dir,
                            unsigned long flags);
struct dma_async_tx_descriptor *
dmaengine_prep_slave_sg(struct dma_chan *chan, struct scatterlist *sgl,
                        unsigned int sg_len, enum dma_transfer_direction dir,
                        unsigned long flags);
struct dma_async_tx_descriptor *
dmaengine_prep_dma_cyclic(struct dma_chan *chan, dma_addr_t buf, size_t buf_len,
                          size_t period_len, enum dma_data_direction dir);
```

- Common flags are:
  - DMA_PREP_INTERRUPT: Generates an interrupt once done
  - DMA_CTRL_ACK: No need for a manual ack of the transaction
- The descriptor returned can be used to fill-in a callback:

```
desc->callback = foo_dma_complete;
desc->callback_param = foo_dev;
```

2. Queue the next operation:

```
dma_cookie_t cookie;

cookie = dmaengine_submit(desc);
ret = dma_submit_error(cookie);
if (ret)
    ...
```

3. Trigger the queued transfers

```
dma_async_issue_pending(chan);
```

3bis. In case anything went wrong or the device should stop being used, it is possible to terminate all ongoing transactions with:

```
dmaengine_terminate_sync(chan);
```

Examples

- Commented network driver, whith both streaming and coherent mappings:
  `https://bootlin.com/pub/drivers/r6040-network-driver-with-comments.c`
- Example of usage of the slave API: look at the code for `stm32_i2c_prep_dma_xfer()`.