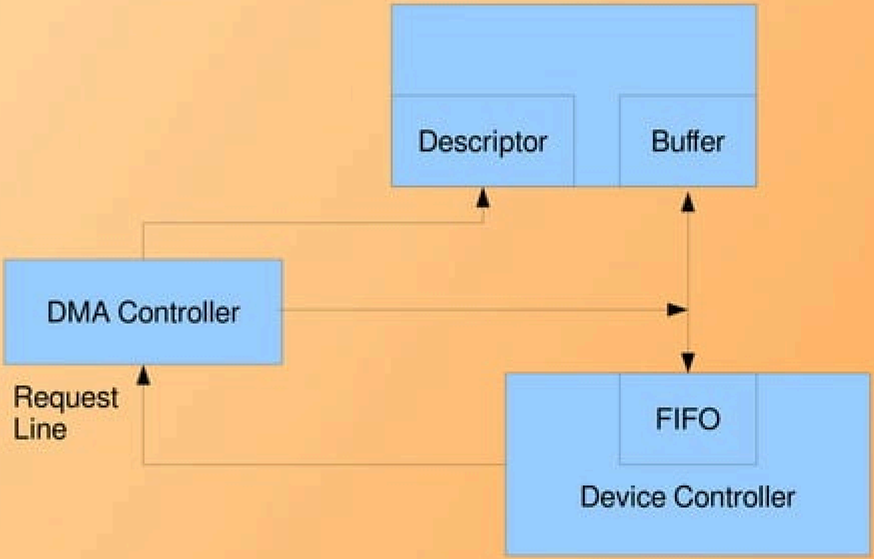
Linux DMA Engine

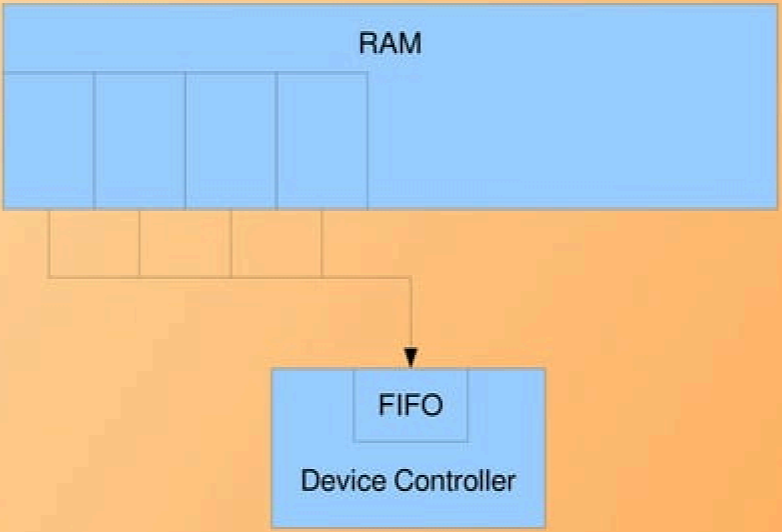
What to Expect?

- DMA Controllers
- Types of DMA Transfers
- Linux DMA Engine API
- Steps for DMA Transfer

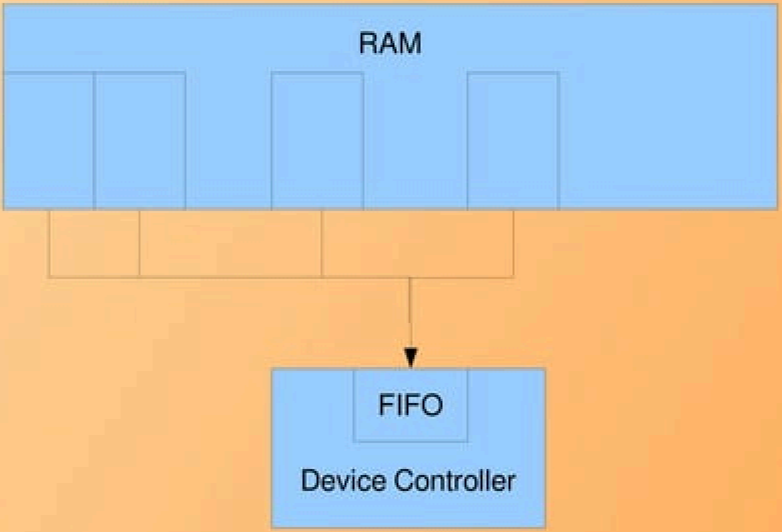
DMA Controllers



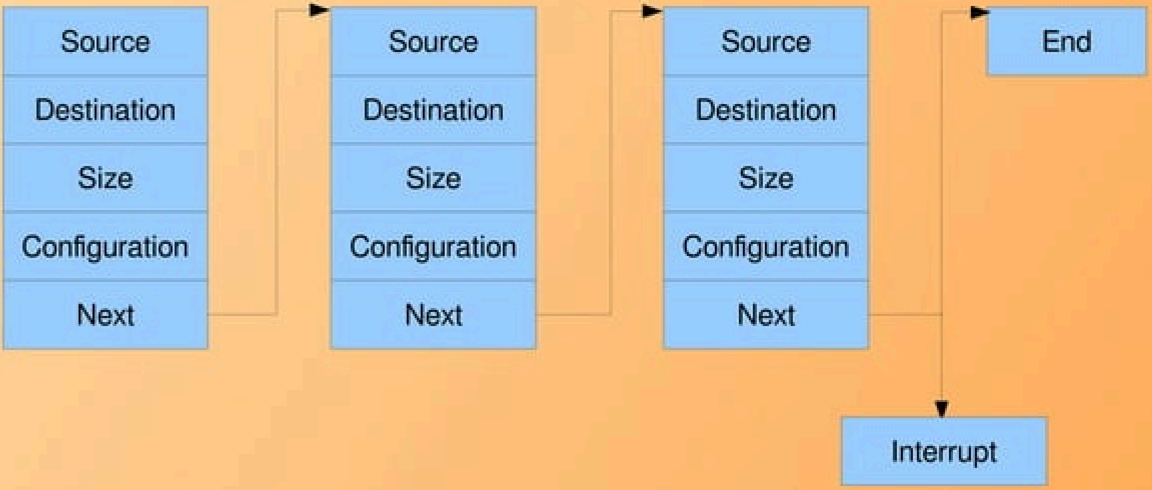
Contiguous Memory DMA



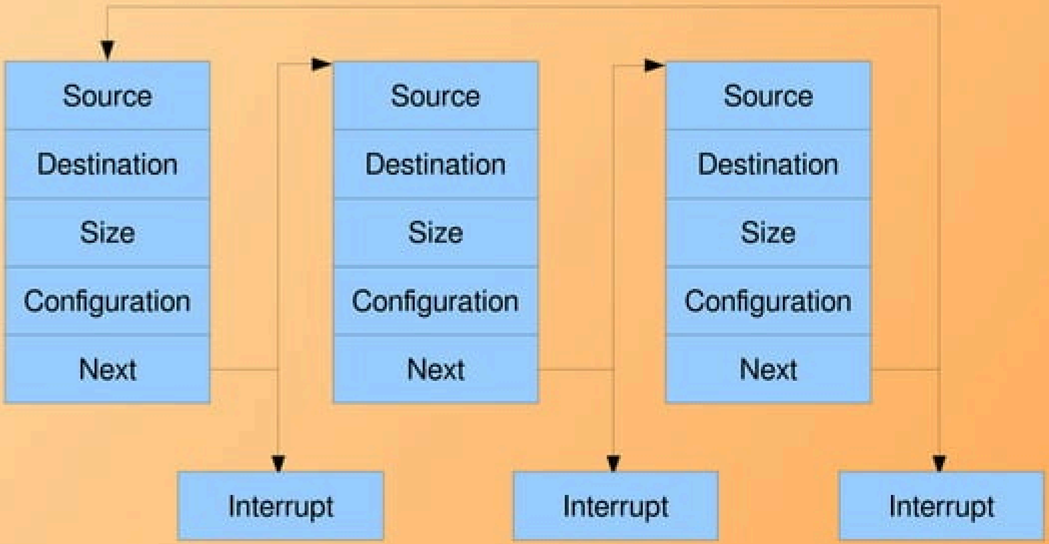
Scatter Gather DMA



Scatter Gather Descriptors



Cyclic Transfers



Source
Destination
Size
Configuration
Next

Source
Destination
Size
Configuration
Next

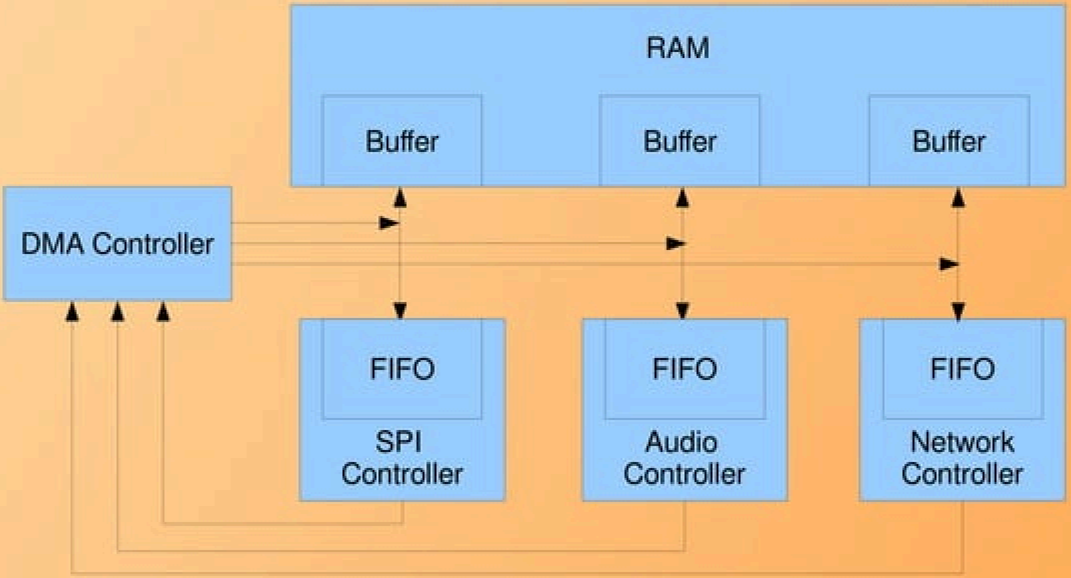
Source
Destination
Size
Configuration
Next

Interrupt

Interrupt

Interrupt

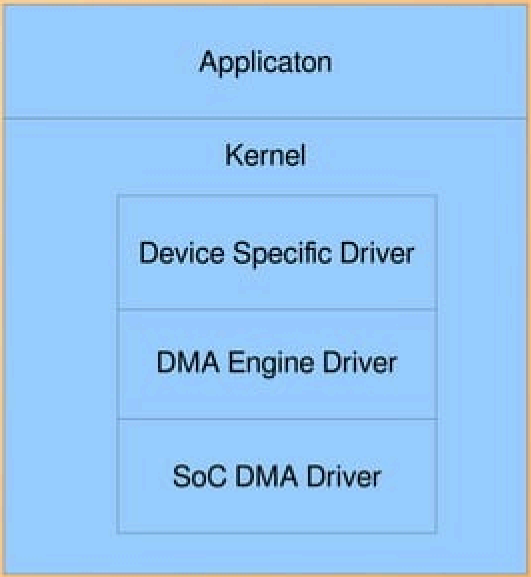
DMA Controller with Peripherals



Linux DMA Engine

- The DMA Engine driver works as a layer on the top of SoC (e.g. TI) specific DMA driver using the slave DMA API
    - The reason its called as slave is due to the fact that software initiates the DMA transactions to the DMA controller hardware rather than a hardware device with a integrated DMA initiating the transfer
- Drivers using the DMA Engine are referred to as a clients

Linux DMA Engine...



Applicaton

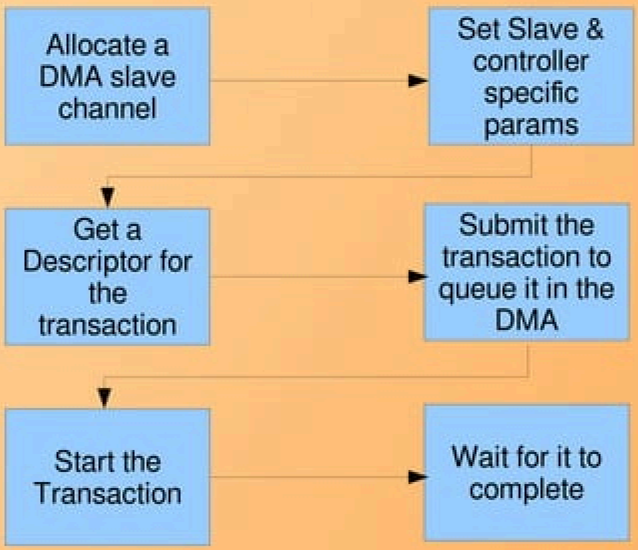
Kernel

Device Specific Driver

DMA Engine Driver

SoC DMA Driver

Steps in DMA Transfer



Allocate a DMA slave channel

Set Slave & controller specific params

Get a Descriptor for the transaction

Submit the transaction to queue it in the DMA

Start the Transaction

Wait for it to complete

DMA channel

- Header: include/linux/dmaengine.h
- Data Structure:

```
struct dma_chan {
    struct dma_device;
    dma_cookie_t cookie;
    chan_id;
    client_count;
    table_count;
}
```

Allocate a DMA Channel

- Header: include/linux/dmaengine.h
- APIs:
  - dma_caps_set(transaction_type, mask);
  - struct dma_chan *dma_request_slave_channel(struct device *dev, const char *name);

Allocate a DMA Channel Example

- Set up the capabilities for the channel that will be requested
  - dma_cap_mask_t mask;
  - dma_cap_zero(mask);
  - dma_cap_set(DMA_SLAVE | DMA_PRIVATE, mask);
- Request the DMA channel from the DMA engine
  - chan = dma_request_channel(mask, NULL, NULL);
- Release the DMA channel
  - dma_release_channel(chan);

Set DMA Slave & Controller Params

- DMA slave channel runtime configuration
- Data Structure:

```
struct dma_slave_config {
    enum dma_transfer_direction direction;
    dma_addr_t src_addr;
    dma_addr_t dst_addr;
    enum dma_slave_buswidth src_addr_width;
    enum dma_slave_buswidth dst_addr_width;
    u32 src_maxburst;
    u32 dst_maxburst;
    bool device_fc;
    unsigned int slave_id;
};
```

- API: dmaengine_slave_config(dma_channel, dma_slave_config *);

Set Controller Params Example

- struct dma_slave_config cfg;
- cfg.src_addr = OMAP2_MCSPI_RX0;
- cfg.dst_addr = OMAP2_MCSPI_TX0;
- cfg.src_addr_width = width;
- cfg.dst_addr_width = width;
- cfg.src_maxburst = burst;
- cfg.dst_maxburst = burst;
- dmaengine_slave_config(dma_channel, cfg);

Prepare the Descriptor

- Descriptor is used to describe a DMA transaction
- struct dma_async_tx_descriptor *dmaengine_prep_slave_single(parameters);
- Parameters:
  - struct dma_chan *chan
  - dma_addr_t buff
  - size_t len
  - enum dma_transfer_direction dir
  - unsigned long flags
    - DMA_PREPARE_INTERRUPT, DMA_CTRL_ACK

Prepare a Descriptor & Setting the Callback Example

- Allocate a 1K buffer of cached contiguous memory
  - char *buf = kmalloc(1024, GFP_KERNEL | GFP_DMA);
- Get the physical address for the buffer
  - dma_buf = dma_map_single(device, buf, 1024, DMA_TO_DEVICE);
- Prepare the descriptor for the DMA transaction
  - Enum dma_ctrl_flags flags = DMA_CTRL_ACK | DMA_PREP_INTERRUPT;
  - chan_des = dma_engine_prep_slave_single(chan, dma_buf, 1024, DMA_MEM_TO_DEV, flags);
- Set up the callback function for the descriptor
  - chan_des->callback = <callback when the transfer completes>;
  - chan_des->callback_param = cmp;

Submit & Start the DMA Transaction

- The dmaengine_submit() function submits the descriptor to the DMA engine to be put into the pending queue
  - The returned cookie can be used to check the progress
- The dma_sync_issue_pending() function is used to start the DMA transaction that was previously put in pending queue
  - If the channel is idle, then the first transaction in queue is started and the subsequent transactions are queued up
  - On completion of each in queue, the next in queue is started and a tasklet is triggered. The tasklet will then call the client driver completion callback routine for notification, if set

Submit & Start the Transaction

- Submit the descriptor to DMA engine
  - dma_cookie_t dmaengine_submit(struct dma_async_tx_descriptor *desc);
- Start the transaction
  - dma_async_issue_pending(struct dma_chan *chan);

References

- <kernel src tree>/Documentation/dmaengine/
  - client.txt
- External Video Reference
  - https://www.xilinx.com/video/soc/linux-dma-in-device-drivers.html

What all have we learnt?

- DMA Controllers
- Types of DMA Transfers
- Linux DMA Engine API
- Steps for DMA Transfer

Any Queries?