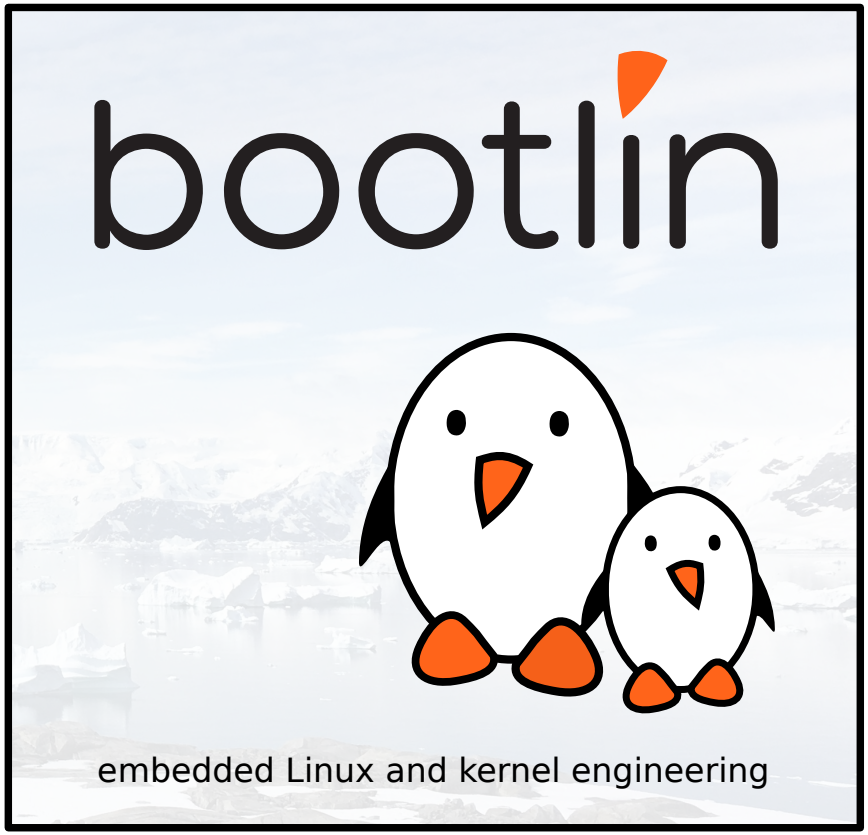
An Overview of the DMAEngine Subsystem

Maxime Ripard
maxime@bootlin.com

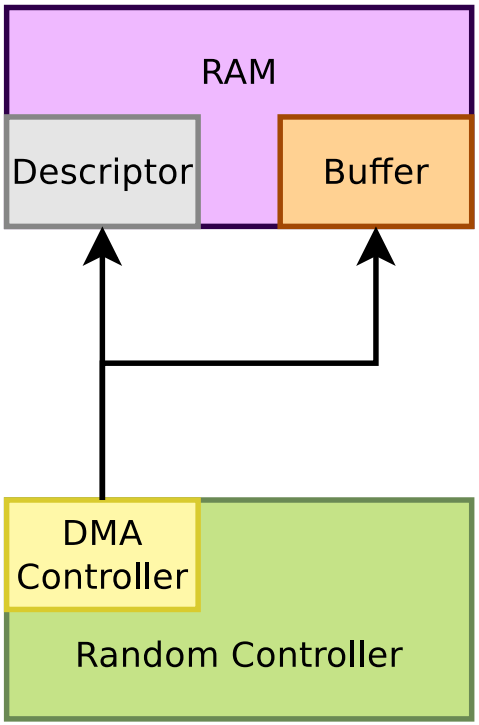


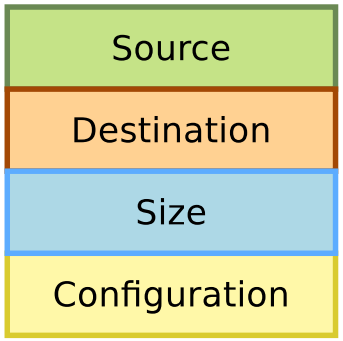
Maxime Ripard

- Embedded Linux engineer and trainer at Bootlin
  - Embedded Linux **development**: kernel and driver development, system integration, boot time and power consumption optimization, consulting, etc.
  - Embedded Linux **training**, Linux driver development training and Android system development training, with materials freely available under a Creative Commons license.
  - `http://bootlin.com`
- Contributions
  - **Kernel support for the sunXi SoCs** from Allwinner
  - Contributor to few open-source projects, **Buildroot**, an open-source, simple and fast embedded Linux build system, **Barebox**, a modern bootloader.
- Living in **Toulouse**, south west of France

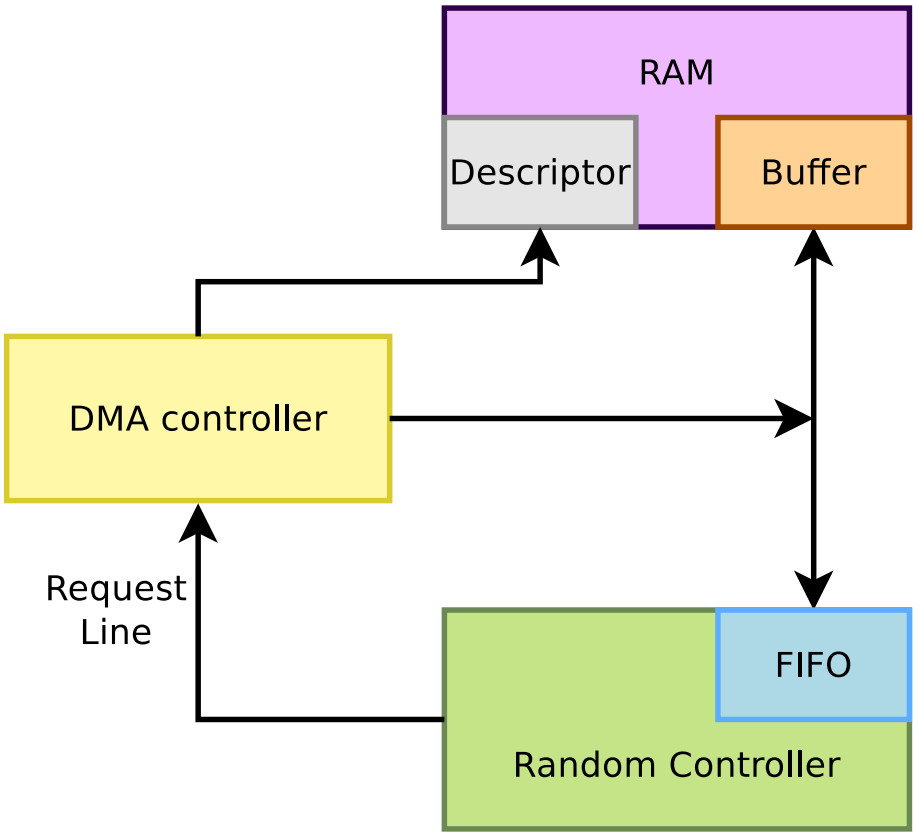
Introduction

Device DMA vs...

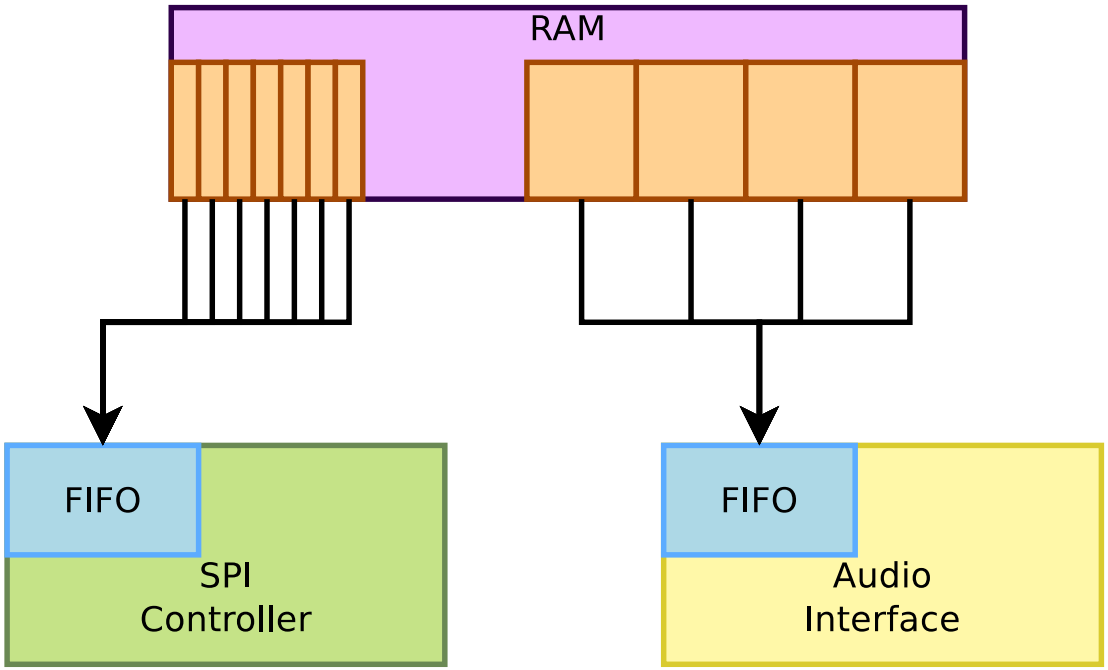




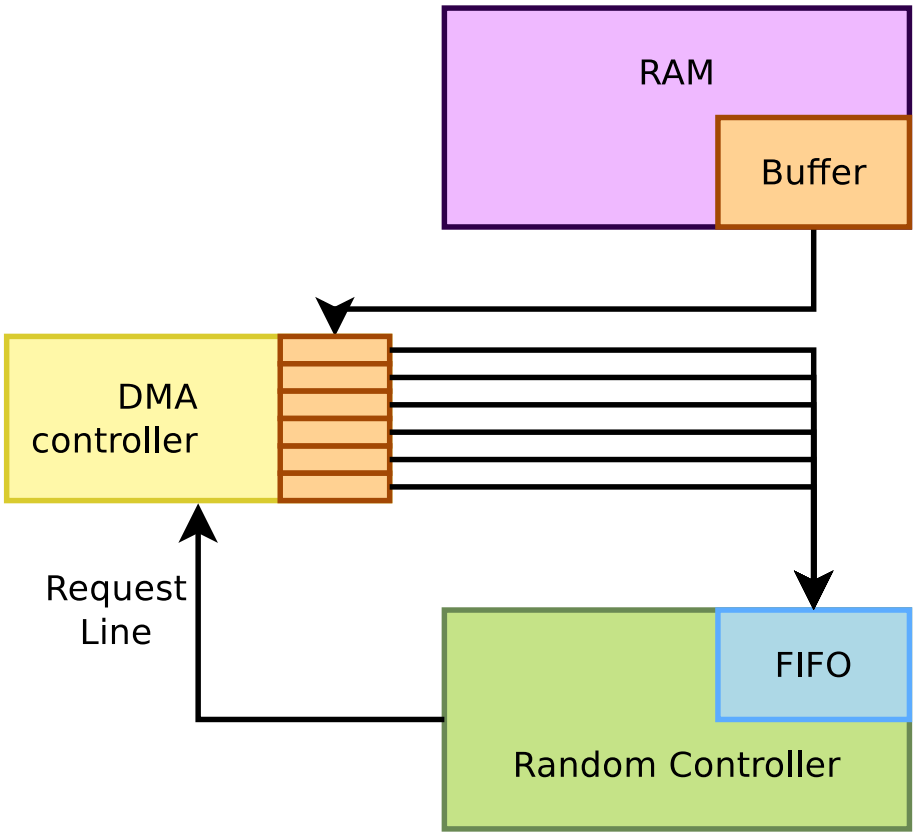
... DMA Controllers



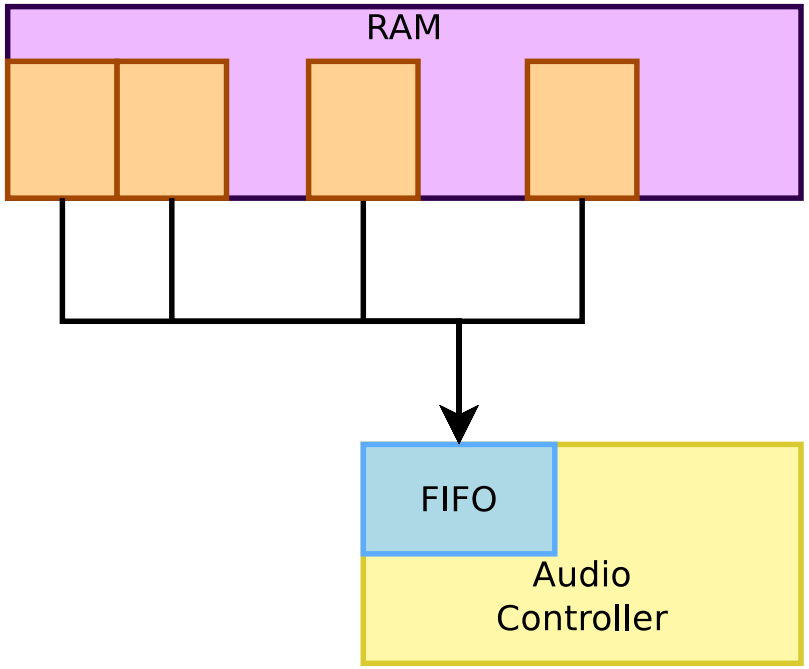
Transfer Width



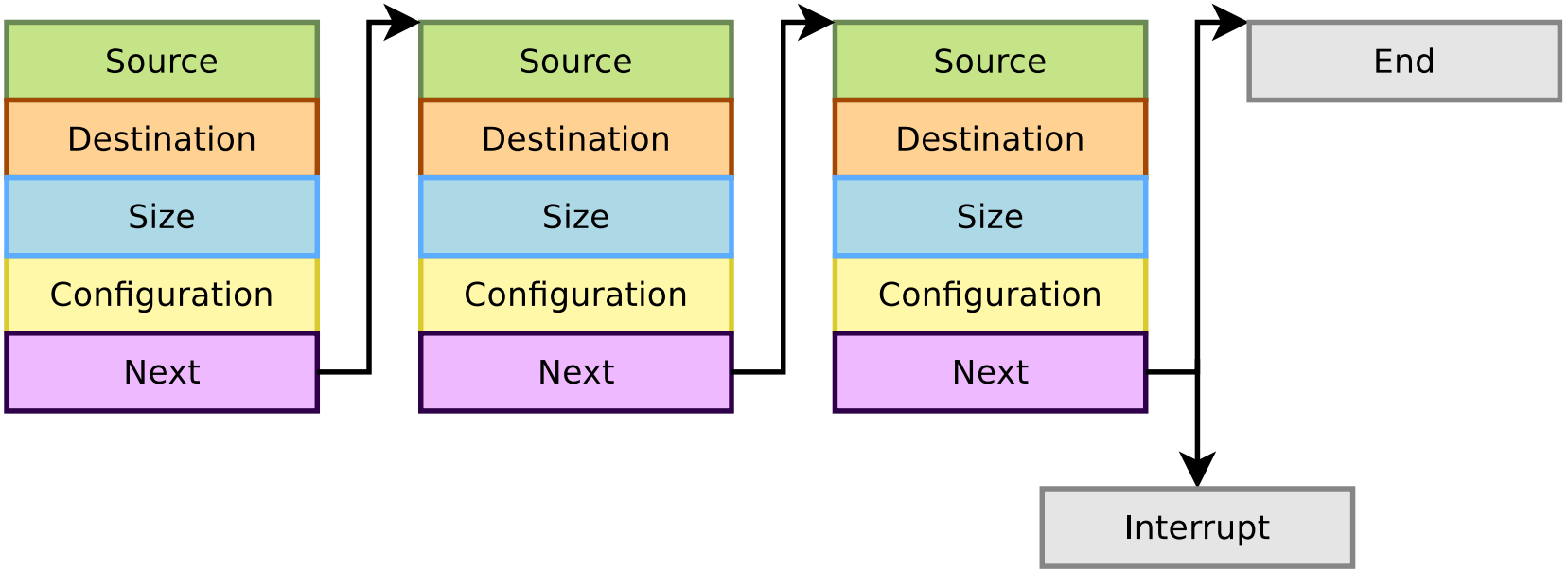
Burst Size


RAM
Buffer
DMA
controller
Request
Line
FIFO
Random Controller

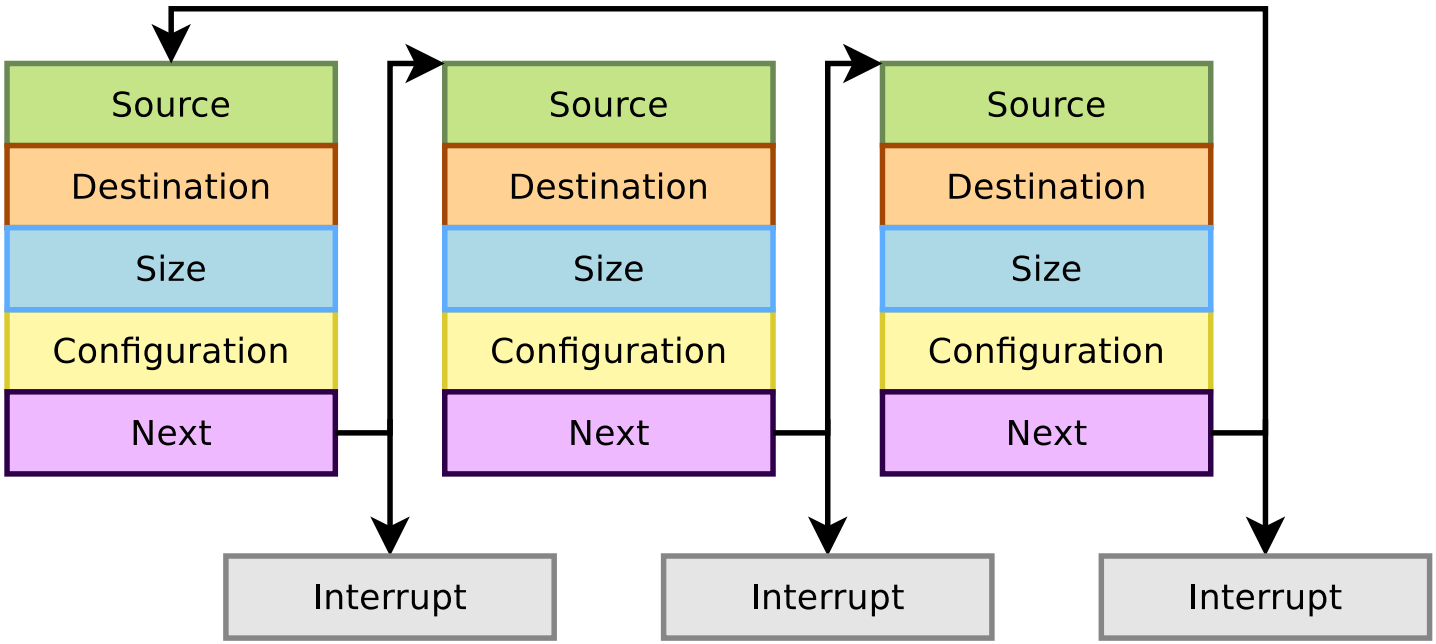
Scatter Gather



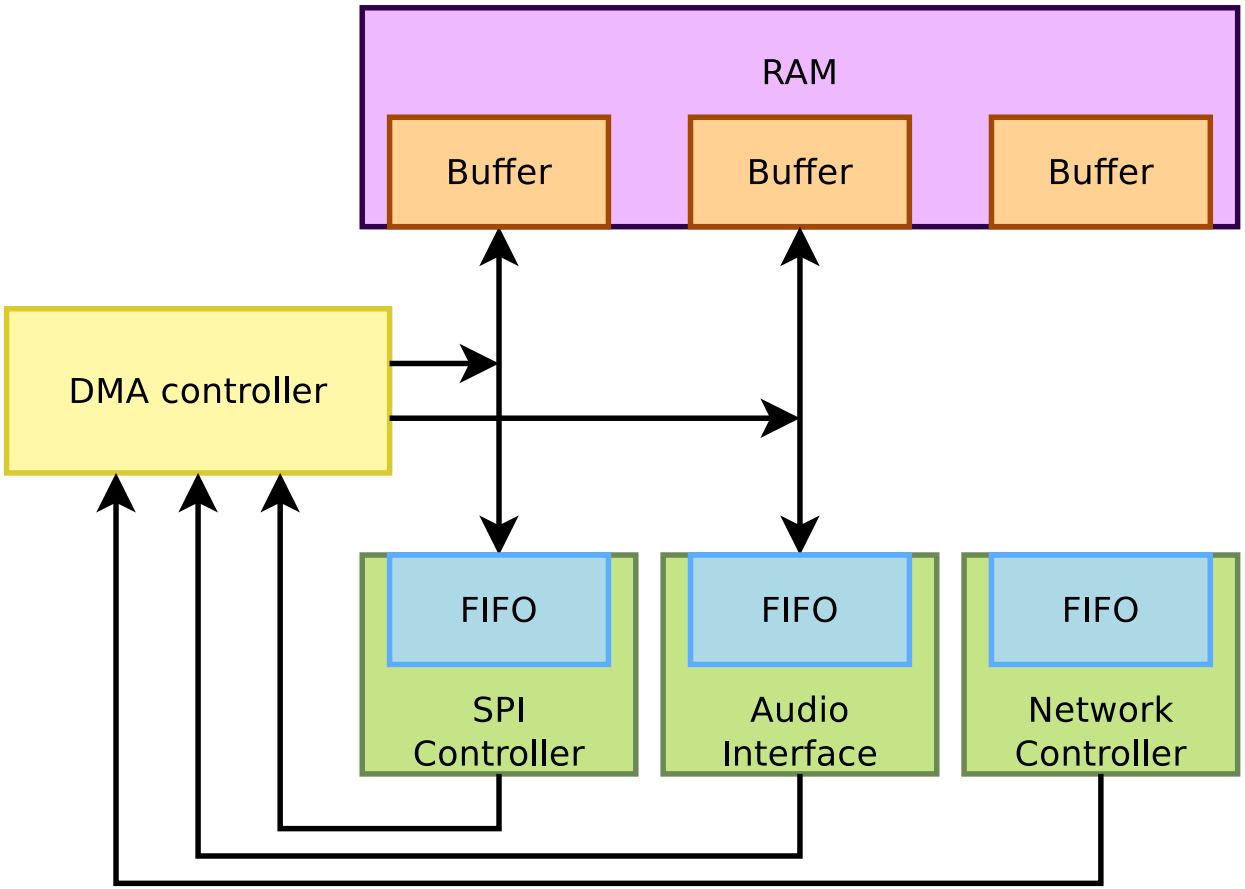
Scatter Gather Descriptors



Cyclic Transfers



Realistic DMA Controller



Linux Support

APIs

- DMAEngine
  - Merged in 2006, in 2.6.18
  - Subsystem to handle memory-to-device transfers
- Async TX
  - Merged in 2007, in 2.6.23
  - Initially part of the raid5 code to support the XScale offload engines
  - Subsystem to handle memory to memory operations (memcpy, XOR, etc.)
  - Implemented on top of dmaengine, but takes many shortcuts, instead of being a real client.

Slave consumer API

1. Request a channel: `dma_request_channel`, or one of its variants
2. Configure the channel for our use: `dmaengine_slave_config`
3. Get a transaction descriptor for our transfer: `dmaengine_prep_*`
4. Put the transaction in the driver pending queue: `dmaengine_submit`
5. Issue pending requests (blocks and calls back your driver to give an update on the transfer status): `dmaengine_issue_pending`

Slave Controller Drivers

`struct dma_device` and its Fields

- DMAEngine, like any framework, relies on a structure you have to fill with various pieces of information in order to do its job properly.
- Mostly:

  channels Initialized list of the channels supported by the driver, of the size of the number of channels supported by your driver

  ***_align** Alignment in bytes for the Async TX buffers

DMA Transfer Types 1/2

- The next step is to set which transfer types your driver supports
- This is done through the function `dma_cap_set`, which takes various flags as an argument:
  - `DMA_MEMCPY`
    - Memory to memory copy
  - `DMA_SG`
    - Memory to memory scatter gather
  - `DMA_INTERLEAVE`
    - Memory to memory interleaved transfer
  - `DMA_XOR`
    - Memory to memory XOR
  - `DMA_XOR_VAL`
    - Memory buffer parity check using XOR

DMA Transfer Types 2/2

- `DMA_PQ`
  - Memory to memory P+Q computation
- `DMA_PQ_VAL`
  - Memory buffer parity check using P+Q
- `DMA_INTERRUPT`
  - The device is able to generate a dummy transfer that will generate interrupts
- `DMA_SLAVE`
  - Memory to device transfers
- `DMA_CYCLIC`
  - The device is able to handle cyclic transfers

Weird Transfer Types

- `DMA_PRIVATE`
  - Async TX doesn't go through `dma_request_channel` but circumvents it, and just starts using any random channel it can.
  - It does so unless you set this flag
- `DMA_ASYNC_TX`
  - Set by the core when you support all Async TX transfer types
  - Used only if `ASYNC_TX_ENABLE_CHANNEL_SWITCH` is enabled
  - Used by `dma_find_channel`, which is a non-exclusive equivalent of `dma_request_channel`, used only by Async TX

Channels Resources Allocation

- `device_alloc_chan_resources` and `device_free_chan_resources`
- Called by the framework when your channel is first requested
- Allows to allocate custom resources for your channel, and free them when you're done
- Optional (since 3.20)

Transaction Descriptor Retrieval Functions

- `device_prep_dma_*`
- Optional, but have to match the transfer types you declared
- Should create both the software descriptor, for Linux and clients to identify the transfer, and the hardware descriptor matching it for the dma controller.
- Should also ensure that the parameters of this transfer match what the driver supports

Submitting Pending Jobs

- `device_issue_pending`
- Should take the first descriptor of the transaction and start it
- Should go through all the descriptors in the list, notifying the client using an optional callback of the status of the transfer

Transfer Status Reporting

- `device_tx_status`
- Reports the current state of a given transaction descriptor
- Does so using the `dma_set_residue` function, and returns only a flag saying whether it's done or in progress.
- This is where the granularity we used earlier comes into action.

Channel configuration

- `device_control`
- Takes an additional flag, to give the type of control to do on the channel
  - `DMA_PAUSE`
    - Pause a given channel
  - `DMA_RESUME`
    - Resume a given channel
  - `DMA_TERMINATE_ALL`
    - Aborts all transfers on a given channel
  - `DMA_SLAVE_CONFIG`
    - Configures a given channel with new parameters

Capabilities

- `device_slave_caps`
- Returns various pieces of information about the controller
  - Can the transfer be paused? terminated?
  - Which transfer widths are supported?
  - Which slave directions are supported?
- Used by generic layers to get an idea of what the device they're going to use is capable of (only ASoC so far)

Recent Developments

Generic Capabilities (3.20)

- Removal of `device_slave_caps`, and moved the logic in the framework
- Introduction of new variables in `struct dma_device`

  `*_width` Bitmask of supported transfer width, both as source and destination

  `directions` Bitmask of the supported slave directions (memory to device, device to memory, device to device)

  `granularity` Granularity of the transfer residue your controller can provide: bursts, chunks or descriptors
- Split of the `device_control` function in four independent functions: `device_config`, `device_pause`, `device_resume`, `device_terminate_all`

Scheduled DMA

- Many DMA controllers have more requests than channels
- These drivers usually have all the scheduling code
- Plus, every driver has a lot of administrative code, that is not trivial to get right (callback deferral, allocation of the descriptors, etc.), yet similar from one driver to another
- Scheduled DMA framework abstracts away most of it, and only a few things remain in the drivers:
  - Interrupt management
  - LLI related functions (iterators, configuration, etc.)
  - Scheduling hints
  - Channel management (pause, resume, residues, etc.)

Questions? Suggestions? Comments?

Maxime Ripard
maxime@bootlin.com

`http://bootlin.com/pub/conferences/2015/elc/ripard-dmaengine`