June 2025

Skip the wait

Maximizing SPI throughput in the Linux kernel with SPI offloading

About the speaker

- David Lechner (dlech)
- Embedded software engineer, BayLibre
  - Embedded software consultancy based near Nice, France, with ~60 engineers around the world contributing to open source projects like Linux, U-Boot, Android, Yocto Project, Zephyr and the GNU toolchain.
- LEGO MINDSTORMS robotics expert
  - ev3dev - Debian Linux on LEGO MINDSTORMS EV3
  - Pybricks - MicroPython on LEGO Powered Up smart hubs

Table of contents

- What is SPI “offloading”?
- The goal
- The journey
- The result
- What’s next?

What is SPI “offloading”?

What is SPI “offloading”?

What is SPI “offloading”?

Generally speaking, “offloading” is having the Linux CPUs do less work and have other dedicated hardware do more work.

Why is SPI offloading?

Any time we can avoid waiting on the CPU, we can get higher throughput and/or lower latency. Also, more accurate timing.

What does SPI offloading look like?

It can take many forms…

What does SPI offloading look like?

A *specialized* hardware with a SPI execution engine that can **record and play back** SPI messages.

Examples:

- Freescale QuadSPI and NXP FlexSPI (lookup tables)
- Analog Devices AXI SPI Engine

… or do something similar using DMA to write to registers on a “conventional” SPI controller.

https://github.com/msperl/spi-bcm2835/blob/refactor_dmachain_for_prepared_messages/spi-bcm2835dma.c

… or use a dedicated microcontroller core for SPI operations.

What does SPI offloading look like?

Program a set of instructions.

For example:

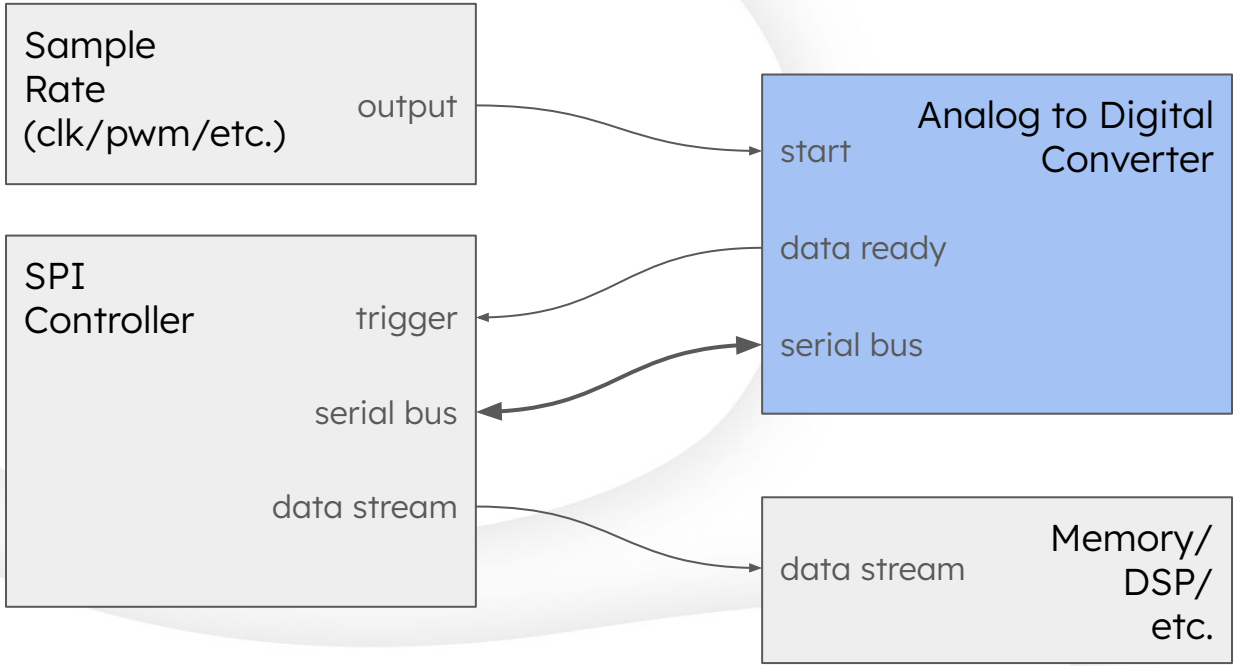
1. Assert chip select
2. Read one word
3. Deassert chip select

Trigger the execution of those instructions.

This could be:

- Software trigger
- Hardware trigger

Example hardware setup with ADC

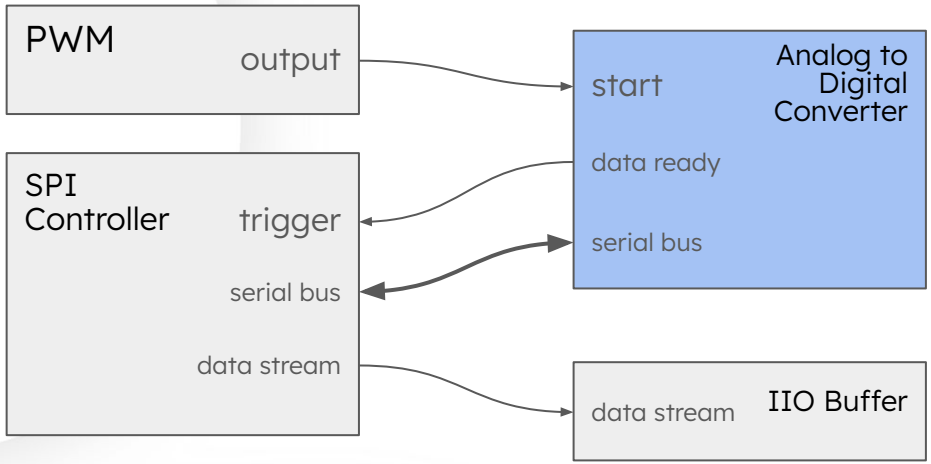


The goal

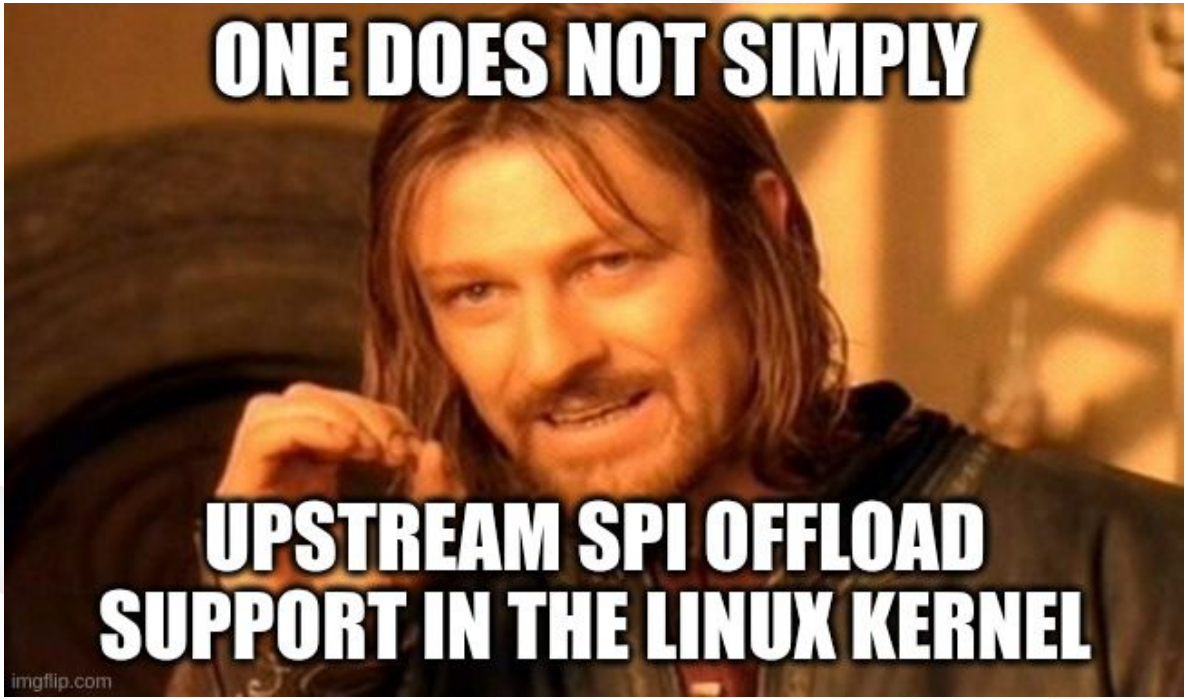
The goal...

Upstream out-of-tree offload support for the Analog Device Inc. (ADI) AXI SPI Engine using a hardware setup as shown.

Operate at 1 to 4 million samples per second and pass noise and harmonic distortion tests.



So we clean it up and send it to the mailing list and...



The journey

(just the highlights)

Break it down and research past efforts

The initial feedback was positive but came with some suggestions to rethink the approach.

> *This is a lot to do in one go, and I think it's a bit too off on the side and unintegrated with the core. There's two very high level bits here, there's the pre-cooking a message for offloading to be executed by a hardware engine and there's the bit where that's triggered by some hardware event rather than by software.*
>
> Mark Brown, https://lore.kernel.org/linux-spi/2c74aad9-3cb9-4222-8072-e72120c2658e@sirena.org.uk/

And also some hints to past discussions on similar topics.

- [RFC] A new SPI API for fast, low-latency regmap peripheral access, From: David Jander
- Depreciated spi_master.transfer and "prepared spi messages" for an optimized pipelined-SPI-DMA-driver, From: Martin Sperl

Step 2: pre-cooking SPI messages

- Was discussed extensively and had a few basic patches proposed in [PATCH] added API: spi_message_compile/optimize/hint/..., From: Martin Sperl
- Basic idea is that this is a “best effort” optimization for **messages that will be reused**.
- No special “offloading” hardware required. **All SPI hardware benefits**, e.g. by avoiding repeating expensive `__spi_validate()` function call.
- Opportunities for more optimization, e.g. avoid repeating DMA mapping.

Step 2: pre-cooking SPI messages

- Now we have spi_optimize_message(), introduced in kernel v6.9.
  https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=786115655f4d9167bd855872bd55b0a37321806e
- How do we know it improves performance?
- On older 32-bit ARM hardware, in an ADC driver we observed:
  With these changes, CPU usage of the threaded interrupt handler, which calls spi_sync(), was reduced from 83% to 73% while at the same time the sample rate (frequency of SPI xfers) was increased from 20kHz to 25kHz.

Step 3: didn't happen

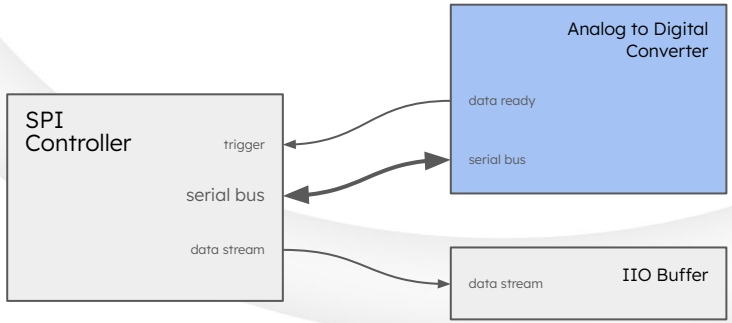
- Wanted to break down remaining work even more, e.g. implement some sort of offloading that required some dedicated hardware, like Martin Sperl's Raspberry Pi DMA offloading proof of concept.
  https://github.com/msperl/spi-bcm2835/blob/refactor_dmachain_for_prepared_messages/spi-bcm2835dma.c
- This did not help advance towards the goal and would have taken a lot of time, so we opted to skip this sidequest.
- Would likely be a useful thing to do someday if anyone wants to dust that off and try it out.

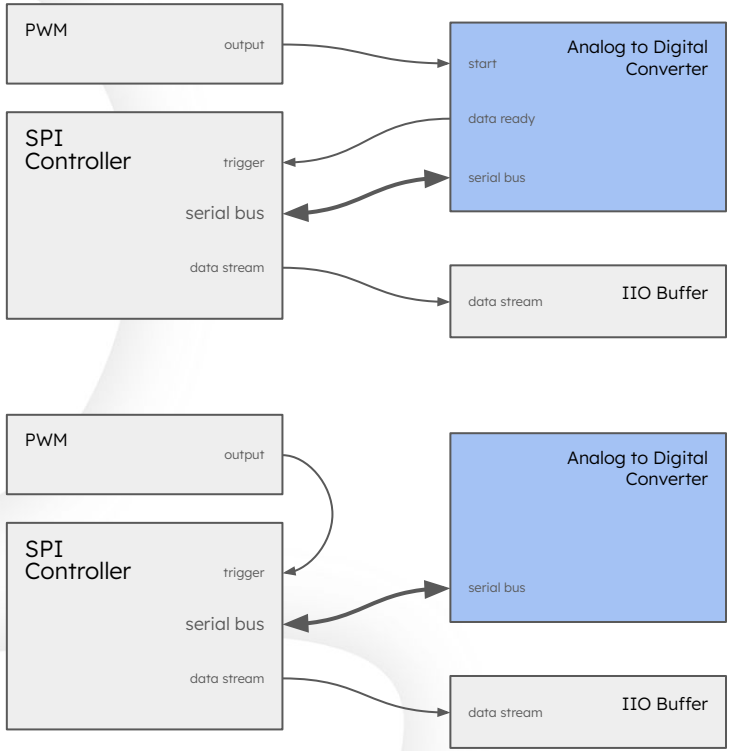
Devicetree bindings for triggers and DMA buffers

Devicetree bindings for triggers and DMA buffers

We need to describe our hardware.

But also make it generic enough to support any possible future hardware.



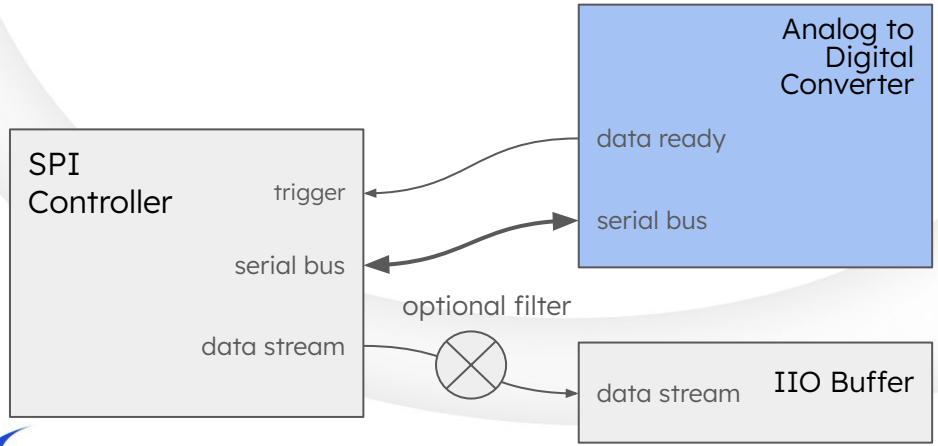


Examples of various ADC wiring requirements

Step 5: streaming data

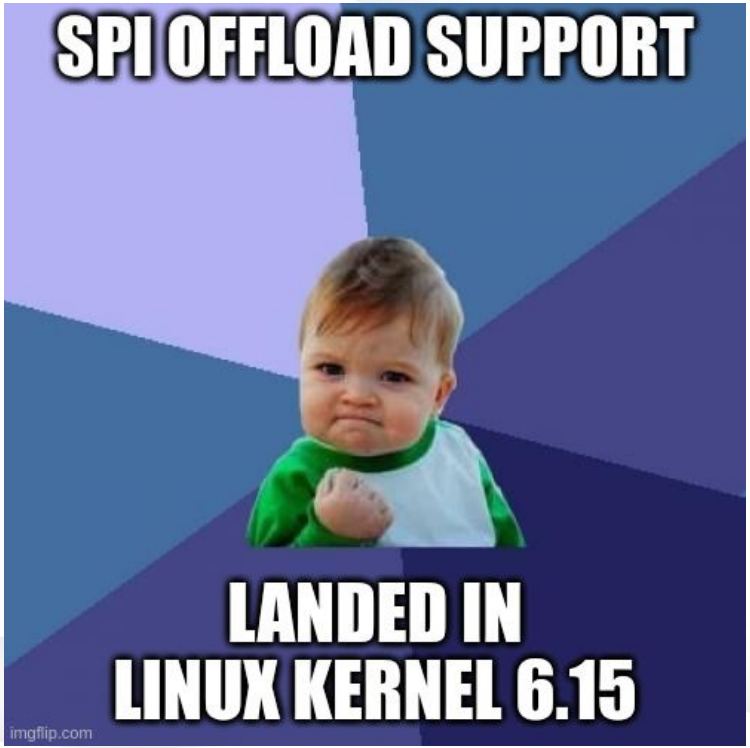
Q: When is a DMA connected to a SPI controller not a SPI controller DMA?

A: When it is an IIO buffer.



We made the case that having a DMA channel dedicated to the offload directly connected to the SPI controller is just a special case of a generic streaming data input or output.

The result


SPI OFFLOAD SUPPORT
LANDED IN
LINUX KERNEL 6.15
imgflip.com

New SPI APIs

General:

- **`spi_optimize_message()`**
  - Includes core SPI optimizations that benefit all use cases
  - Add controller-specific optimizations with **`struct spi_controller.optimize_message`**

Offloading:

- include/linux/spi/offload/
  - **consumer.h**: for SPI device drivers
  - **provider.h**: for SPI controller drivers (and trigger provider drivers)
  - **types.h**: shared types

SPI offload provider APIs

- SPI offload provider
  - **devm_spi_offload_alloc()** - call this for each offload instance (e.g. for each lookup table slot)
  - Must also implement **`struct spi_controller.optimize_message`** to handle messages with **`struct spi_message.offload`** set.
- Trigger provider
  - **devm_spi_offload_trigger_register()** - register a new trigger provider
  - This includes **`struct spi_offload_trigger_ops`**:
    - **`match`** - matches provider to consumer, e.g. from devicetree **`trigger-sources`**
    - **`validate`** - ensures trigger will work before actually enabling it (fail early)

SPI offload consumer APIs

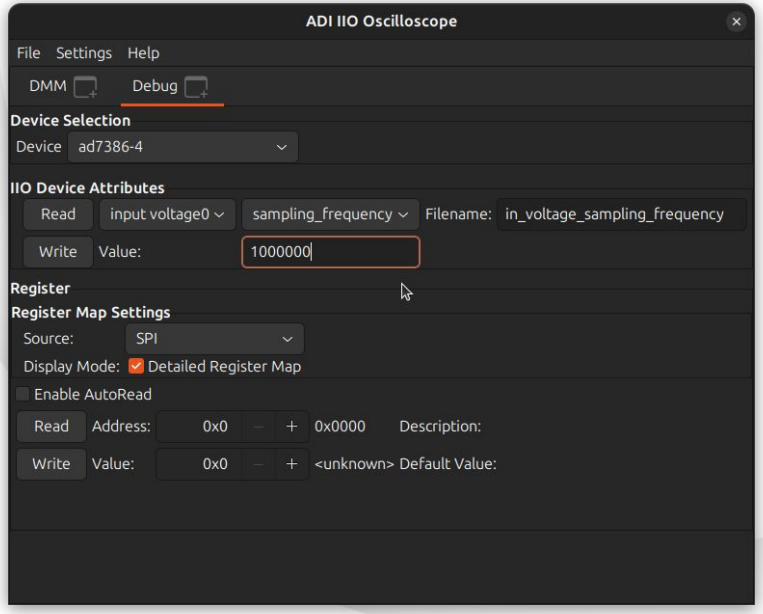
- SPI device APIs
  - **devm_spi_offload_get()** - only succeeds if SPI controller has offload support
  - **devm_spi_offload_trigger_get()** - plus `spi_offload_trigger_{validate,enable,disable}()`
  - **devm_spi_offload_{tx,rx}_stream_request_dma_chan()** - for DMA connected to data input or output stream
- SPI message extensions
  - **`spi_optimize_message()`** - with **`struct spi_message.offload`** set.
  - **`struct spi_transfer.offload_flags`** - used to select tx or rx data stream - omit **`tx_buf`** or **`rx_buf`** in this case but still include **`len`**.

Real-world example

We can get **1 to 2.5 million** samples per second in `several upstream drivers now.`

- `drivers/iio/adc/{ad4000,ad4695,ad7380,ad7606,ad7944}`
- `drivers/iio/dac/ad5791`

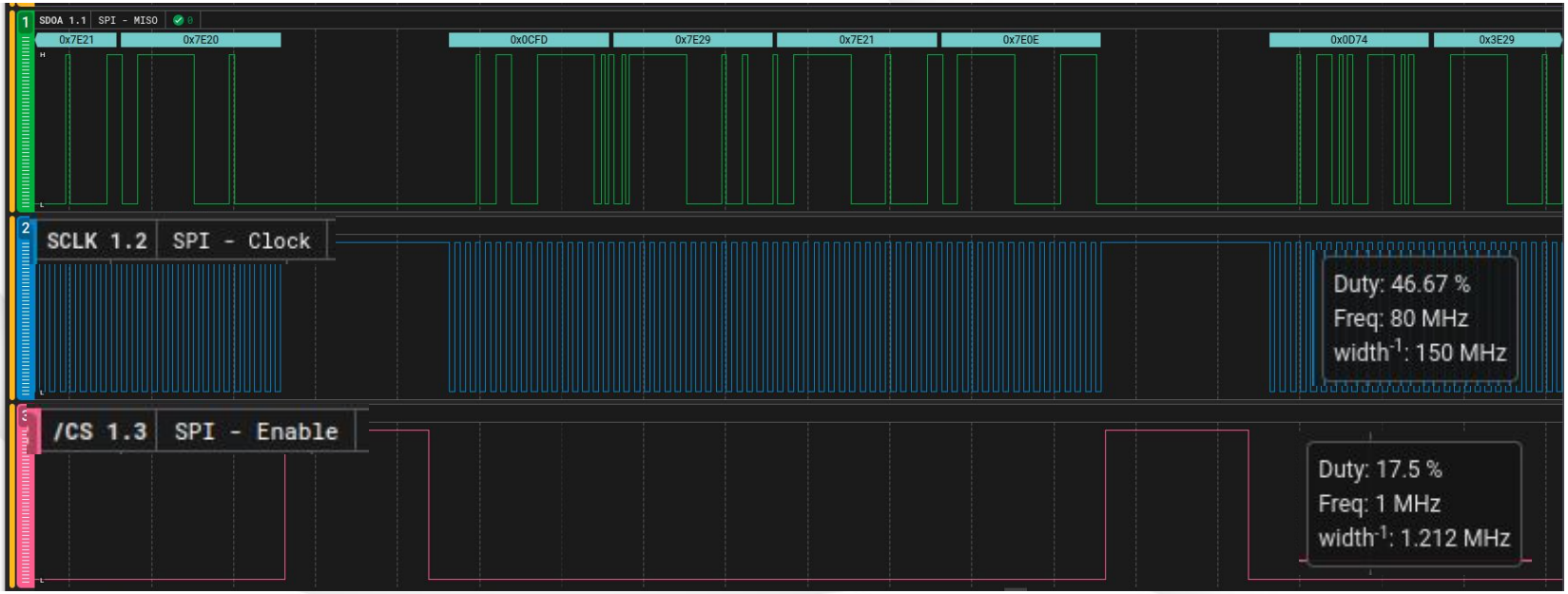
Real-world example



Using IIO Oscilloscope to capture 1 kHZ signal at 1 million samples per second.

▸ The result

Real-world example



Verifying with logic analyzer.

- The result

What’s next?

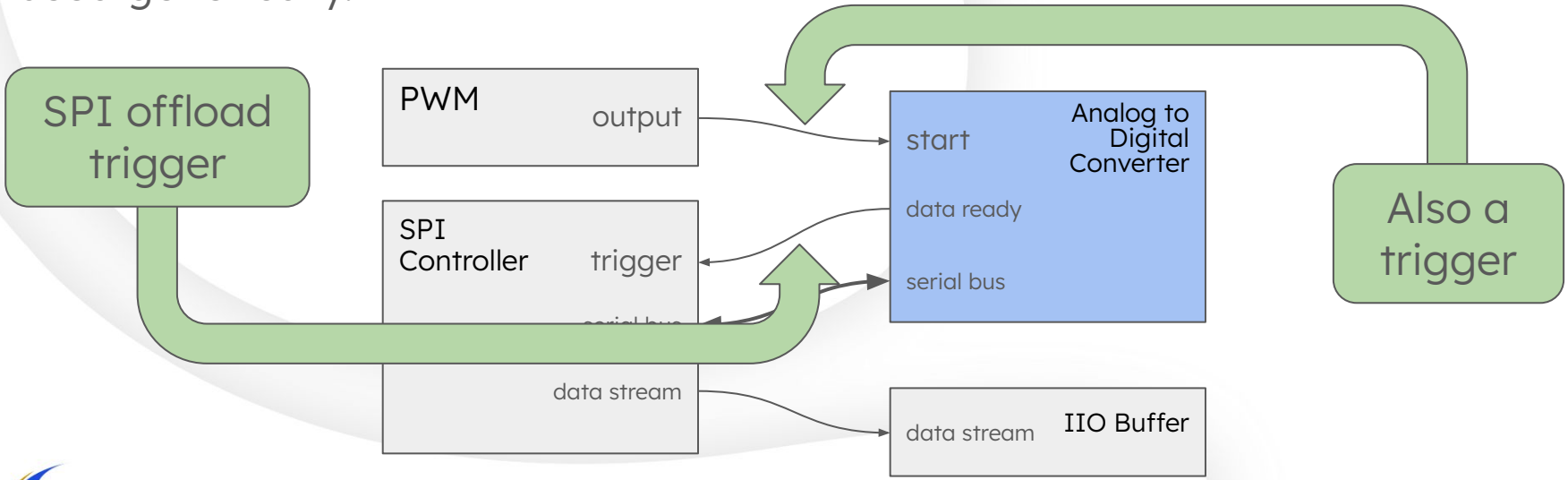
Idea: more SPI offload applications

- CAN controllers - this came up the most in discussion where current performance is not good enough
- SPI flash memory - actually make use of lookup tables
- Quoting Mark Brown's suggestions:

  Something like pre-cooking the commands to read the interrupt status registers from a device, send a network transfer, or to download firmware and settings if you power the device off frequently. Basically anything with more than one operation that you might want to run repeatedly and care about the performance of.

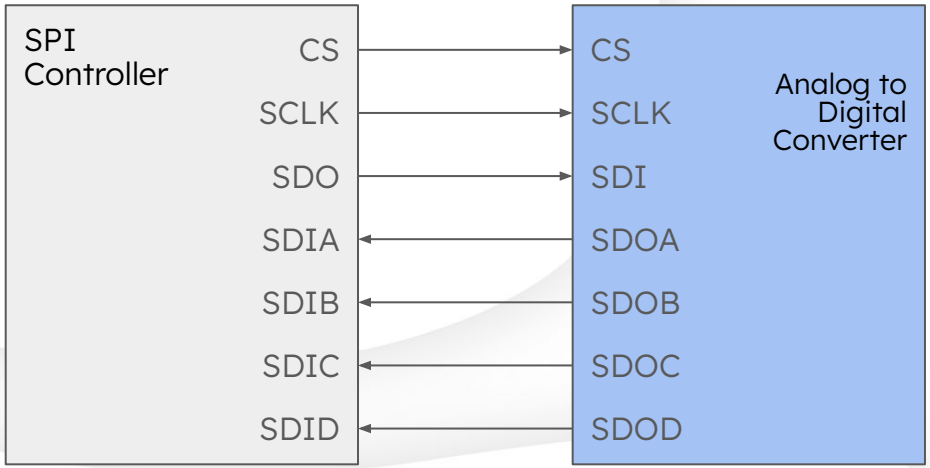
Idea: trigger subsystem

There is no reason that the same abstraction for offload triggers could not be used generically.



Idea: SPI controller with multiple buses

Simultaneous sampling ADCs have multiple data lines - one for each channel.



Thank you

Appendix

Example device tree:

```
spi@30000 {
    compatible = "adi,axi-spi-engine-1.00.a";
    reg = <0x30000 0x1000>;
    ...

    dmas = <&rx_dma 0>;
    dma-names = "offload0-rx";
    trigger-sources = <&adc_trigger>;

    eval_adc: adc@0 {
        compatible = “adi,ad4112”;
        reg = <0>;
        ...
    };
};

adc_trigger: trigger@40000 {
    compatible = "adi,util-sigma-delta-spi";
    reg = <0x40000 0x100>;
    clocks = <&sys_dma_clk>;
    #trigger-source-cells = <0>;
};
```