ChatGPT

Learning Linux Device Drivers on Raspberry Pi 5: DMA, SPI & I²S with ADC/DAC Tutorial

This tutorial will guide you through developing simple Linux device drivers on a **Raspberry Pi 5** to interface an **MCP3008 SPI Analog-to-Digital Converter (ADC)** and a **PCM5102A I²S Digital-to-Analog Converter (DAC)**. We will use **Direct Memory Access (DMA)** for efficient data transfers, **memory-mapped I/O (MMIO)** for register access, and handle **interrupts** in the kernel driver. By the end, you'll understand how analog signals from a potentiometer can be digitized and processed, then converted back to analog audio – a full **analog ↔ digital ↔ analog** pipeline. The tutorial is split into two sections:

(A) *MCP3008 + Potentiometer* – SPI ADC input with DMA (RX)
(B) *PCM5102A + Audio Output* – I²S DAC output with DMA (TX)

Each section covers hardware setup, protocol theory (SPI or I²S), how DMA is used, driver implementation snippets (including DMA setup and interrupt handling), and a simple test plan.

A. MCP3008 ADC Input (SPI + DMA RX)

Goal: Read an analog voltage from a potentiometer using the MCP3008 ADC, via the Pi's SPI interface. Use DMA to transfer ADC data to memory with minimal CPU intervention, and handle interrupt signals for data-ready events.

Hardware Setup: Wiring MCP3008 to Raspberry Pi 5

Wiring diagram: Raspberry Pi (GPIO header) connected to MCP3008 ADC and a potentiometer.

Connect the MCP3008 and potentiometer to the Raspberry Pi 5's 40-pin GPIO header as follows [1] [2]:

- **Power & Reference:** MCP3008 VDD → 3.3 V (Pin 1), VREF → 3.3 V (Pin 17). Ground MCP3008 AGND and DGND to Pi GND (Pin 6, 9, or any ground). This powers the ADC and sets reference voltage to 3.3 V (the ADC's input range).
- **SPI Signals:** MCP3008 CLK → SPI SCLK (GPIO 11, physical pin 23) [3]. MCP3008 DOUT (ADC data out) → SPI MISO (GPIO 9, pin 21). MCP3008 DIN (ADC data in) → SPI MOSI (GPIO 10, pin 19). MCP3008 CS (chip select) → SPI CE0 (GPIO 8, pin 24) on the Pi. *(Note: The diagram shows CS wired to a generic GPIO#22 as in the Adafruit example, but using the hardware CE0 pin is recommended for the SPI controller to toggle CS automatically.)* [3]
- **Potentiometer:** Use a 10 kΩ pot (or similar) as an analog input. Connect one end of the pot to 3.3 V and the other end to GND. Connect the wiper (middle pin) to MCP3008 channel 0 (CH0) [2]. This way, turning the pot changes the voltage on CH0 between 0 V and 3.3 V, which the MCP3008 will measure.

With this wiring, the MCP3008's CH0 will report a 10-bit digital value (0–1023) corresponding to the pot's analog voltage. The Raspberry Pi's SPI0 interface will communicate with the MCP3008 using the MOSI/MISO/CLK/CE0 lines and 3.3 V logic levels.

SPI and ADC Theory (MCP3008 + Raspberry Pi)

The **Serial Peripheral Interface (SPI)** is a synchronous serial bus with a master-slave architecture. The Raspberry Pi acts as the SPI master, clocking data in and out of the MCP3008 ADC. Key SPI signals include: clock (SCLK), Master-Out-Slave-In (MOSI), Master-In-Slave-Out (MISO), and chip-select (CS). The master drives SCLK and CS, and data is exchanged on MOSI/MISO lines. In our setup, the Pi will send commands on MOSI and receive ADC data on MISO.

MCP3008 ADC: This chip is an 8-channel, 10-bit ADC. To start a conversion, the Pi sends a Start bit and channel selection bits via MOSI, and the MCP3008 responds by clocking out the 10-bit measurement on MISO [3]. Each reading occurs over a 3-byte SPI transfer. The ADC uses the analog reference (3.3 V) as the full-scale input; an input at 3.3 V yields a digital value of 1023, and 0 V yields 0. For intermediate voltages, the output is proportional (e.g. ~512 for ~1.65 V).

SPI Mode: MCP3008 communicates in SPI Mode 0 (clock idle low, sample on rising edge). The Raspberry Pi's SPI driver will be configured accordingly (mode 0, max frequency typically up to 1 MHz– stable for MCP3008). We'll use the Pi's SPI0 bus (CE0) to talk to the MCP3008.

Using DMA for SPI RX: Normally, reading from SPI involves the CPU transferring each byte and possibly waiting for each conversion. By using **DMA (Direct Memory Access)**, we can let the DMA controller autonomously handle data movement from the SPI peripheral's receive FIFO into memory. This frees the CPU from byte-by-byte interrupts and allows higher sampling rates. The SPI controller can be set up to trigger a DMA request when its RX FIFO reaches a certain level [4] [5]. The DMA engine then copies incoming data directly into a kernel buffer. With DMA, the ADC can be read *continuously or in large blocks* with precise timing and minimal CPU load – ideal for scenarios like audio sampling or fast sensors [6]. In our driver, we'll initiate an SPI transfer for a block of ADC samples and use DMA to receive the data into a buffer.

Driver Implementation Snippets: MCP3008 SPI ADC with DMA

Below are simplified kernel driver snippets (in C) highlighting the key steps: setting up a DMA-safe buffer, configuring an SPI transfer, and handling interrupts for DMA completion. This is a **char device** or **SPI driver** running in kernel space. (For brevity, error checking is omitted.)

```
#define NSAMPLES 1000  // e.g. number of ADC samples per batch
#define TRANSFER_LEN (NSAMPLES * 3)  // 3 bytes per sample transaction

// 1. Allocate a DMA-coherent buffer for SPI RX data (ADC readings)
void *rx_buf;
dma_addr_t rx_buf_dma;
rx_buf = dma_alloc_coherent(&pdev->dev, TRANSFER_LEN, &rx_buf_dma, GFP_KERNEL);
if (!rx_buf) { /* handle allocation failure */ }
```

```
// 2. Prepare SPI transfer structure to read NSAMPLES from MCP3008
uint8_t *tx_cmd = kzalloc(TRANSFER_LEN, GFP_KERNEL);
// (Populate tx_cmd with the ADC channel read command repeated NSAMPLES times)
// For MCP3008, each 3-byte sequence: [start+single+channel_bits], [0], [0]

// Use SPI transfer with DMA (the SPI controller will use DMA if available)
struct spi_transfer xfer = {
    .tx_buf        = tx_cmd,
    .rx_buf        = rx_buf,
    .len           = TRANSFER_LEN,
    .tx_dma        = rx_buf_dma,   // use DMA addresses
    .rx_dma        = rx_buf_dma,
    .cs_change     = 0,            // deassert CS after transfer
    .bits_per_word = 8,
    .speed_hz      = 1000000,      // 1 MHz SPI clock
};
struct spi_message msg;
spi_message_init(&msg);
spi_message_add_tail(&xfer, &msg);

// 3. Submit SPI message asynchronously
int ret = spi_async(spi_device, &msg);
```

When we call `spi_async`, the SPI core will start the transfer. Because the length is relatively large (3000 bytes in this example), the controller's driver (e.g., `spi-bcm2835.c`) will likely use DMA mode if possible [7]. The BCM2835/2712 SPI controller supports DMA and has a dedicated DREQ line for pacing the transfers. The DMA engine will move data from the SPI RX FIFO to `rx_buf` in memory as each ADC sample arrives, without CPU intervention on each byte.

```
// 4. Handle completion in interrupt (called when DMA+SPI transfer finishes)
static irqreturn_t spi_dma_irq(int irq, void *dev_id) {
    struct my_adc_dev *dev = dev_id;
    // Check and clear DMA interrupt status (device-specific register)
    // ... (clear DMA channel interrupt flag) ...

    // Signal that ADC data is ready
    dev->adc_done = true;
    wake_up_interruptible(&dev->wait_queue);
    return IRQ_HANDLED;
}

// 5. In driver probe or init, request the DMA completion IRQ
// (Assume we know the DMA channel's IRQ number, e.g., from DTS or platform
```

```
data)
request_irq(dev->dma_irq, spi_dma_irq, 0, "mcp3008_dma", dev);
```

In this snippet, we assume the DMA controller raises an interrupt when the transfer is complete. We register an interrupt handler with `request_irq()` for the DMA channel's IRQ. In the handler, we would acknowledge the interrupt in the DMA controller's registers and then wake up any process or thread waiting for the data. The use of `request_irq` follows the standard Linux pattern – it returns 0 on success or `-EBUSY` if the IRQ line is already in use [8].

Note: On Raspberry Pi, DMA controllers are a shared resource. Instead of manually manipulating DMA registers, real drivers often use the Linux **DMA Engine API** to request a DMA channel and submit "slave" DMA transfers (configured to read from the SPI FIFO register). For clarity, our example treats DMA in a simplified way. Proper implementation would involve mapping the SPI FIFO physical address, setting up a DMA descriptor with the SPI's DREQ signal, etc., or simply relying on the SPI master driver to handle DMA internally [9] [10].

Also note the use of `dma_alloc_coherent` for the buffer. This ensures the memory is not cached (or is properly synchronized for DMA) and is aligned as required [11] [12]. This prevents cache coherency bugs when the DMA engine writes to memory without CPU involvement.

After the SPI+DMA transfer, our buffer `rx_buf` contains an interleaved stream of command/response bytes. We would parse out the 10-bit ADC values from the response bytes (the MCP3008 outputs 2 null bits, then 10 data bits). The driver can then copy these values to user space or kernel logic (e.g., via a `read()` on a char device or by storing in a sysfs attribute).

Testing the MCP3008 ADC DMA Driver

To test this part:

1. **Build and Load the Module:** Compile the driver as a kernel module and insert it (e.g., `sudo insmod mcp3008_dma.ko`). Ensure SPI is enabled on the Pi (add `dtparam=spi=on` in **/boot/config.txt** if not already). The driver may register a device like `/dev/myadc0` for reading ADC values.
2. **Observe ADC Readings:** With the module running, read from the device. For example, use `cat /dev/myadc0` or a small test program to initiate an ADC capture. The driver will trigger a DMA transfer of (say) 1000 samples.
3. **Adjust the Potentiometer:** Turn the pot knob while the capture is running. You should see the reported values change accordingly (e.g., near 0 when pot is at GND, near 1023 when at 3.3 V). The data might be printed to the kernel log or buffered for user space reading, depending on the driver design. Each batch of samples is transferred via DMA – you can verify in `/proc/interrupts` that CPU interrupts during sampling are minimal (just one per buffer, from the DMA completion IRQ).

This demonstrates efficient analog input sampling. The use of DMA allows consistent timing (e.g., reading at fixed intervals if triggered by a timer or continuous clock) without bogging down the CPU with per-sample interrupts [13] [14].

B. PCM5102A DAC Output (I²S + DMA TX)

Goal: Output a sound waveform (e.g., a tone) using the PCM5102A DAC connected via the Raspberry Pi's I²S (PCM) interface. We'll stream audio samples from memory to the DAC using DMA (TX), and handle interrupts for buffer refilling or transfer completion.

Hardware Setup: Wiring PCM5102A to Raspberry Pi 5

Wiring diagram: Raspberry Pi connected to a PCM5102A I²S DAC module (audio output).

The PCM5102A DAC board connects to the Pi's GPIO header as follows (for the common PCM5102 modules) [15]:

- **Power:** PCM5102A VIN → 5 V (Pin 2 or 4). *(The module has an onboard regulator, since the chip runs on 3.3 V logic but can be powered from 5 V.)* [16] [17]. Connect PCM5102A GND → Pi GND (Pin 6, 14, or any ground).
- **I²S Signals:** The Raspberry Pi's PCM (I²S) interface uses specific GPIO pins [18] [19]:
- **BCK (Bit Clock):** Connect PCM5102A **BCK** pin to Pi **PCM_CLK** (GPIO 18, Pin 12) [20].
- **LCK (Left/Right Clock):** Connect PCM5102A **LCK** (LRCLK) to Pi **PCM_FS** (GPIO 19, Pin 35) [20].
- **DIN (Data In to DAC):** Connect PCM5102A **DIN** to Pi **PCM_DOUT** (GPIO 21, Pin 40) [20].
- **Master Clock (SCK):** Tie the PCM5102A **SCK** pin to **GND** [21]. This tells the PCM5102A to internally generate its master clock (using its PLL) instead of expecting an external MCLK. *This is crucial – if SCK is left floating, audio output will be distorted* [22].
- *(Optional)* **XSMT / FMT pins:** Some PCM5102A boards have mute/control pins. For basic playback in default I²S mode, XSMT can be tied to enable output (often high or to VIN) and FMT (format select) tied to ground (for standard I²S format). Many modules have these pre-soldered or configured by jumpers. Verify your module's instructions if audio is muted or format is wrong (e.g., as noted by users bridging XSMT and FMT pins) [23] [24].

Double-check the pin connections using a GPIO pinout reference [25]. The Raspberry Pi 5 uses the same PCM pin functions on GPIO 18, 19, 20, 21 as previous models (and also offers an alternate PCM on GPIO 28–31, not used here). Our wiring is effectively the same as for a Pi 3/4 with a DAC HAT [26] [27].

After wiring, connect the DAC module's audio output to an amplifier or powered speakers (the PCM5102A outputs line-level analog audio on its L and R outputs, typically via an on-board 3.5 mm jack or breakout pins).

I²S and DAC Theory (PCM Audio Output)

I²S (Inter-IC Sound) is a serial bus specifically designed for digital audio data. On the Raspberry Pi, the I²S interface is referred to as PCM (Pulse Code Modulation) in the datasheet [18] [19]. Key signals: - **Bit Clock (BCK or PCM_CLK):** Clock that ticks for each data bit. For example, at 44.1 kHz stereo, 16-bit audio, BCK runs at 44,100 * 16 * 2 ≈ 1.4112 MHz (for 16-bit stereo). - **Left/Right Clock (LRCLK or PCM_FS):** Also called Frame Sync, toggles to indicate Left vs Right channel data. It runs at the sample rate (e.g., 44.1 kHz). A cycle of LRCLK defines a frame (left channel data when LRCLK=Low, right channel when High, for standard I²S). - **Data (PCM_DOUT):** Serial audio data bits, synced to BCK. The data is typically in two's complement, MSB-

first. In I²S format, the data line is valid one bit-clock after the LRCLK transition (one BCK delay from frame start).

The Raspberry Pi can act as the **I²S master**, generating BCK and LRCLK signals [28] [29]. The PCM5102A DAC is the slave that receives these clocks and data. We configure the Pi for e.g. 16-bit stereo audio. Each frame will be 32 bits (16 bits left, 16 bits right). The PCM5102A outputs high-quality analog audio corresponding to the digital samples.

PCM5102A DAC: This DAC supports up to 32-bit audio and common sample rates (44.1 kHz, 48 kHz, etc.). In our setup, we'll use 16-bit samples for simplicity. The DAC's analog output will be a reconstruction of the digital waveform provided. By feeding it a buffer of audio samples (e.g., a sine wave), we can produce a tone on its output.

Using DMA for I²S TX: Audio streaming requires moving large amounts of data to the DAC at a steady rate. For CD-quality audio, ~176,400 bytes per second must be written to the DAC (44,100 samples/sec * 4 bytes per sample frame). Doing this via CPU-driven writes or interrupts for each sample can cause high CPU usage and risk buffer underruns (audible glitches) [30]. Instead, we use DMA to feed the I²S FIFO. The Raspberry Pi's PCM module has FIFO and can request DMA service when the FIFO needs data (using a DREQ signal) [31] [32]. We can set up a DMA channel in **memory-to-peripheral mode**, so it copies from a memory buffer (source) to the PCM FIFO register (destination) whenever the DAC is ready for more data. The DMA ensures the audio samples are delivered on time, and the CPU only needs to intervene occasionally (e.g., to refill the buffer or handle end-of-stream).

Driver Implementation Snippets: PCM5102A I²S DAC with DMA

We will outline a kernel driver that configures the PCM (I²S) interface and uses DMA to output a continuous tone. We assume the Raspberry Pi's PCM controller base address is known (for BCM2712 it's similar to BCM2835 at 0x7E203000 for PCM registers [33]). In practice, you would obtain these via `platform_get_resource` or device tree. Here's what the driver does:

- **Configure PCM/I²S:** Set the PCM control registers for 16-bit stereo, master mode. For example, in the BCM PCM register map, you'd set TX channel A to 16-bit width and enable it, set frame length (e.g., 32 clocks per frame for 16×2) and enable the PCM interface [34] [35]. Also, enable the PCM clocks via clock manager. (On Pi, `dtparam=i2s=on` does this setup – if writing your own driver without using ALSA, you might need to enable the PCM clock via MMIO or use the clock API.)
- **Prepare Audio Buffer:** Create a buffer with audio sample data. For a test tone, you can generate a sine wave table or even a simple square/sawtooth. For example, allocate a buffer for one period of a 440 Hz sine wave at 48 kHz sample rate (~109 samples for one cycle). Use 16-bit PCM values. This buffer can be repeated in a loop to output a continuous tone.
- **DMA Setup for TX:** Allocate a **coherent DMA buffer** and copy the audio samples into it. Then request a DMA channel for the PCM TX (the Pi has specific DMA channels wired to PCM via DREQ signals). Using the DMA Engine API, we would configure a **slave DMA transfer**: source = our buffer, destination = PCM FIFO register, length = buffer size in bytes. We mark it for cyclic (looped) operation if we want the tone to play indefinitely. For example:

```
// Assume pcm_fifo_phys is the physical MMIO address of PCM FIFO (e.g.,
0x7E203004)
```

```
// and we have a dma_chan for PCM TX obtained via dma_request_chan().

struct dma_slave_config cfg = {
    .direction = DMA_MEM_TO_DEV,
    .dst_addr = pcm_fifo_phys,
    .dst_addr_width = DMA_SLAVE_BUSWIDTH_4_BYTES,  // FIFO access width
    .dst_maxburst = 4,    // burst size (words) - depends on FIFO depth
    .slave_id = PCM_DREQ_TX,  // The DMA request line for PCM TX (from
datasheet)
};
dmaengine_slave_config(dma_chan, &cfg);

// Prepare cyclic DMA transfer
struct dma_async_tx_descriptor *desc;
desc = dmaengine_prep_dma_cyclic(
    dma_chan,               // DMA channel
    buf_dma_addr,           // DMA address of source buffer
    buf_len,                // length of buffer in bytes
    buf_len,                // period length (same as buffer for full loop)
    DMA_MEM_TO_DEV,         // transfer direction
    DMA_PREP_INTERRUPT      // interrupt after each cycle
);
if (!desc) { /* handle error */ }

// Set a callback for when buffer has looped (optional, e.g., to update data)
desc->callback = pcm_dma_callback;
desc->callback_param = dev;
dma_cookie_t cookie = dmaengine_submit(desc);
dma_async_issue_pending(dma_chan);
```

In this code, `dmaengine_prep_dma_cyclic` sets up the DMA to continuously loop over our audio buffer, re-sending it to the PCM FIFO. The DMA hardware will copy each sample word to the FIFO when the PCM's DREQ signal indicates space available (e.g., FIFO level below threshold) [36]. We specify an interrupt on each period, meaning every time the buffer finishes playing and restarts, an interrupt can fire (allowing us to modify the buffer if we wanted to implement, say, streaming audio or a longer waveform).

- **Interrupt Handling:** In a cyclic DMA, we might use the periodic callback to do things like update the buffer with new audio or stop the DMA after a certain duration. For a simple constant tone, we may not need to do much – the DMA will loop automatically. However, let's implement a callback to illustrate:

```
static void pcm_dma_callback(void *data) {
    struct my_dac_dev *dev = data;
    // This is called each time the DMA finishes sending the buffer.
    // We could generate or swap in new audio data here for the next cycle.
    // For a fixed tone, no change is needed. Just count cycles or signal if
needed.
```

```
    dev->cycles++;
    if (dev->cycles == TARGET_CYCLES) {
        // Example: stop after a certain number of cycles
        dmaengine_terminate_async(dev->dma_chan);
        complete(&dev->playback_done);
    }
}
```

This callback runs in softirq context (DMA completion tasklet). We ensure it does minimal work (e.g., flagging that playback is done or updating a buffer index). If we had used a normal IRQ line for PCM (the PCM module *can* issue interrupts for underrun, etc.), we could also demonstrate using `request_irq` for the PCM interrupt, but with DMA properly set up, PCM interrupts for TX may not be needed unless we want underrun notifications. In our DMA approach, as long as DMA keeps the FIFO fed, underruns won't occur.

Testing the PCM5102A DAC DMA Driver

Finally, test the audio output:

1. **Enable I²S/PCM Hardware:** Ensure the I²S interface is enabled on Pi 5. If using our custom driver that directly manages clocks and registers, this is handled in code. Otherwise, on Raspbian you might need `dtparam=i2s=on` in **/boot/config.txt** (and disable default audio with `dtparam=audio=off` to free the PCM pins) [37] .
2. **Load the DAC Driver Module:** Insert the compiled module (e.g., `sudo insmod pcm5102_dma.ko`). The driver will initialize the PCM interface and start the DMA transfer of the audio buffer to the DAC.
3. **Listen for Output:** Connect headphones or powered speakers to the PCM5102A output. You should hear a tone (for instance, a 440 Hz sine wave if that's what we generated). The sound should be clear and continuous. If you hear a distorted or choppy output, revisit the DMA configuration or clock setup. A common mistake is forgetting to tie SCK to ground – ensure that is done so the DAC isn't missing the master clock [22] . The PCM clocks (BCK, LRCLK) should be visible on an oscilloscope if available, showing the bit clock in the MHz range and LRCLK at the audio sample rate.
4. **Monitor System Load:** Even though audio is playing, the CPU usage should be very low. The DMA is handling the heavy lifting of transferring audio samples. This is in contrast to a possible interrupt-driven approach which might cause high CPU usage and risk glitches if the CPU cannot service each audio frame in time [30] . Using DMA, the CPU is free to do other work while audio plays, only interrupted at buffer boundaries or not at all (if no callback).

For experimentation, you can modify the driver to play different waveforms or even to read from the MCP3008 and feed that data to the DAC for a crude analog loopback. For example, one could sample a microphone or sensor via the MCP3008 and immediately output some transformed signal via the PCM5102A. This would involve coordinating the ADC sampling rate with the DAC output rate – a non-trivial but enlightening exercise.

Conclusion

Through these two sections, we built a simple analog input -> output pipeline on Raspberry Pi 5, leveraging kernel-level device drivers with DMA and interrupts for efficiency:

- In **Section A**, an analog voltage from a potentiometer was digitized by the MCP3008 ADC and fetched over SPI using DMA (with the driver handling the SPI bus and an interrupt indicating when a batch of samples is ready). This demonstrated how to perform high-speed ADC reads without burdening the CPU for each sample [13].

- In **Section B**, we configured the Raspberry Pi's I²S interface to stream digital audio to a DAC. We used DMA to continuously feed audio samples to the PCM5102A DAC, producing a stable audio tone with minimal CPU intervention. We also ensured proper synchronization by using the hardware's FIFO and DREQ mechanisms, and handled interrupts for cycle completion. The PCM interface on the Pi provides a straightforward way to get high-quality sound out via an external DAC [38].

Data Flow Recap: *Analog In (potentiometer) → MCP3008 (SPI ADC) → Raspberry Pi (memory buffer via DMA) → PCM5102A (I²S DAC) → Analog Out (audio).* Each step was backed by a Linux driver component: SPI driver for ADC, PCM driver for DAC. DMA was the hero that moved data efficiently: SPI DMA RX for capturing ADC samples into memory, and I²S DMA TX for playing memory samples out to DAC. Interrupts were used to synchronize these transfers (signaling completion or needing attention) while keeping ISR handlers minimal and deferring work appropriately (using waits or callbacks) [8].

By following this tutorial, you learned how to write **easy-to-follow Linux kernel drivers** for real hardware on Raspberry Pi, complete with **clear wiring diagrams**, protocol explanations, and **code snippets** for DMA and interrupts. This foundation can be expanded – for instance, integrating the ADC as a Linux Industrial I/O (IIO) device or the DAC as an ALSA sound card – but the concepts remain the same. You now have a practical understanding of SPI and I²S communication, DMA-driven data transfer, and kernel interrupt handling in the context of device drivers. Happy hacking with your new analog/digital skills!

Sources: The wiring and technical details were confirmed from Raspberry Pi documentation and tutorials [1] [15]. The theory and code behaviors reference the Linux device drivers development guides and kernel documentation [38] [8], as well as community examples of using DMA for SPI and I²S on Raspberry Pi [30] [22].

---

[1] [2] [3] Connecting the Cobbler to a MCP3008 | Analog Inputs for Raspberry Pi Using the MCP3008 | Adafruit Learning System
https://learn.adafruit.com/reading-a-analog-in-and-controlling-audio-volume-with-the-raspberry-pi/connecting-the-cobbler-to-a-mcp3008

[4] [5] [7] [9] [10] [31] [32] [36] drivers/spi/spi-bcm2835.c - kernel/common - Git at Google
https://android.googlesource.com/kernel/common/+/bce1305c0ece3/drivers/spi/spi-bcm2835.c?autodive=0%2F%2F%2F%2F

[6] python - RPi DMA transfer over SPI - Raspberry Pi Stack Exchange
https://raspberrypi.stackexchange.com/questions/118628/rpi-dma-transfer-over-spi

8 ch10.pdf
file://file-9uF9fWkp28GkpKjbXbFt35

11 12 Raspberry Pi DMA programming in C – Lean2
https://iosoft.blog/2020/05/25/raspberry-pi-dma-programming/

13 14 An experiment in bit-banging SPI - Raspberry Pi Forums
https://forums.raspberrypi.com/viewtopic.php?t=71089

15 16 17 20 21 22 23 24 25 37 How to connect a PCM5102 I2S DAC to your Raspberry Pi – Himbeer's Blog
https://blog.himbeer.me/2018/12/27/how-to-connect-a-pcm5102-i2s-dac-to-your-raspberry-pi/

18 19 28 29 33 34 35 Microsoft Word - BCM2835 ARM Peripherals.docx
https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf

26 27 Raspberry Pi I2S Audio Interface – Simple DIY Electronic Music Projects
https://diyelectromusic.com/2021/11/21/raspberry-pi-i2s-audio-interface/

30 GitHub - philpoole/snd_pi_i2s: An I2S PCM ALSA driver for Raspberry Pi
https://github.com/philpoole/snd_pi_i2s

38 PCM at Raspberry Pi GPIO Pinout
https://pinout.xyz/pinout/pcm