

# 06: SPI, Analog I/O

## Microcontrollers

Stefan Huber  
[www.sthu.org](http://www.sthu.org)

Dept. for Information Technologies and Digitalisation  
FH Salzburg

Summer 2024

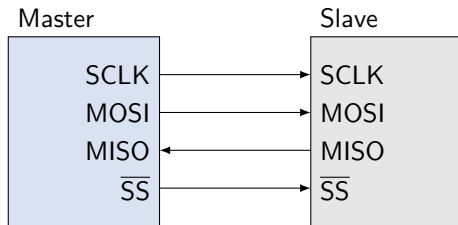
# Section 1

## SPI

The **Serial Peripheral Interface** (SPI) is a serial, synchronous, full duplex, master-slave communication interface.

It is a four-wire serial bus:

- ▶ SCLK: Serial clock
- ▶ MOSI: Master out, slave in
- ▶ MISO: Master in, slave out
- ▶  $\overline{SS}$ : Slave select, often active-low. Other names: chip enable  $\overline{CE}$  or chip select  $\overline{CS}$ .



Multiple slaves can be connect to the same SCLK, MOSI and MISO lines, which are single-ended<sup>1</sup>.

- ▶ The Raspberry Pi has two select lines,  $\overline{SS1}$  and  $\overline{SS2}$ , for two slaves.<sup>2</sup>
- ▶ A slave's MISO line is typically tri-stated and of high impedance when  $\overline{SS}$  is high.

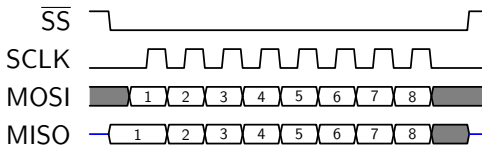
<sup>1</sup> In *single-ended signaling* the voltage on a single line carries the information. In *differential signaling*, where the potential difference between two lines carries the information.

<sup>2</sup> BCM2835, p. 148, fig. 10-1.

# SPI data transmission

Data is transmitted in the following steps:

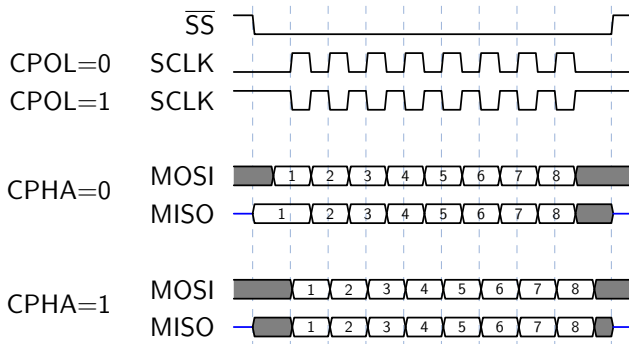
- 1 The master sets  $\overline{SS}$  low for the intended slave. Some slaves support to have  $\overline{SS}$  fixed to low (if there would be only one) and do not require a falling edge.
- 2 The master toggles the clock line. At each tick the master sends a bit to the slave over MOSI and at the same time the slave sends a bit to the master over MISO.
  - ▶ Transmission is always full-duplex even if only uni-directional communication is intended. [A slave cannot not communicate.](#)
  - ▶ Typically words of 8 bits are transmitted.



However, the polarity of the clock (CPOL) and phase shift of the data (CPHA) can be configured.

# SPI timing diagrams

Timing diagrams to transmit an 8-bit word for different CPOL (clock polarity) and CPHA (clock phase) settings<sup>3</sup>:



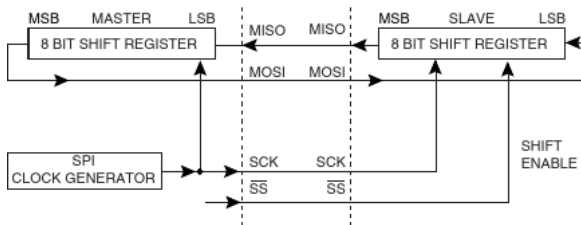
Depending on the data order the first bit is the MSB or the LSB.

<sup>3</sup> ATmega32, p. 139, fig. 67 and fig. 68.

# ATmega32 SPI communication

The ATmega32 can act as SPI master or slave<sup>4</sup>.

- ▶ A prescaler supports 7 different bit rates.
- ▶ We can set data order, CPOL and CPHA in the SPCR register.<sup>5</sup>
- ▶ The  $\overline{SS}$  line is **controlled in software**.



**Figure:** The SPDR register is a single shift register to transmit and, at the same time, receive data, see [ATmega32, fig. 66, p. 133].

<sup>4</sup> ATmega32, p. 132.

<sup>5</sup> ATmega32, p. 136.

# ATmega32 SPI example

```
1 #define DD_SS      DDB4
2 #define DD_MOSI    DDB5
3 #define DD_MISO    DDB6
4 #define DD_SCK     DDB7
5
6 uint8_t readwrite(uint8_t in) {
7     /* Transmit data */
8     SPDR = in;
9     /* Busy wait until interrupt flag is raised. */
10    while (!(SPSR & (1 << SPIF)));
11    /* Return the received data. Clears SPIF, see p.138. */
12    return SPDR;
13 }
14
15 void init() {
16     /* Set MOSI, SCK and SS port as output, and all others as input. */
17     DDRB = (1 << DD_MOSI) | (1 << DD_SCK) | (1 << DD_SS);
18     /* Set DD_SS active (low). */
19     PORTB &= ~(1 << DD_SS);
20     /* Enable SPI, master mode; MSB first order, CPOL=0, CPHA=0, rate is 2 MHz. */
21     SPCR = (1 << SPE) | (1 << MSTR);
22 }
```

## Section 2

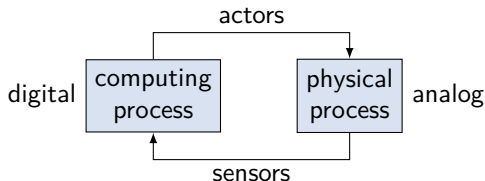
# Analog I/O



# Cyber-physical systems

Many embedded systems are forming cyber-physical systems:

- ▶ Some computing processes act on physical processes through **actors**.
- ▶ Some computing processes obtain information from physical processes through **sensors**.



Physical processes of the physical world deal with **physical quantities**, which are analog:

- ▶ Force, temperature, humidity, current, voltage, position, ...

So at some point we need **analog-digital conversion**.

- ▶ All physical quantities are at the end converted into voltages, which we convert into digital quantities, and vice versa.

Elaborate sensors and actors may provide already a **digital interface**.

But many do not. For instance:

- ▶ A simple photo transistor as light sensor.
- ▶ A humidity or temperature sensor, which is essentially just a resistor.
- ▶ Three-phase motion control requires to output three analog sinus signals.

Hence, we often use analog I/O of the microcontroller:

**ADC** Analog-digital conversion to read analog input.

**DAC** Digital-analog conversion to write analog output.

# DAC via PWM

Analog output can simply be done by low-pass filtering PWM signals.

- ▶ See last lecture.

Analog output can also be done through a [resistor network](#).

See different lecture (or appendix).

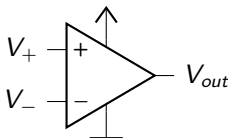
# 1-bit ADC

A 1-bit analog digital conversion is nothing more than an **analog comparator**.

- ▶ Given two voltages  $V_+$  and  $V_-$  a comparator gives

$$\begin{cases} 1 & \text{if } V_+ > V_- \\ 0 & \text{otherwise} \end{cases}.$$

- ▶ In hardware this is simply achieved by an operation amplifier.



It outputs

$$V_{out} = \begin{cases} V_{cc} & \text{if } V_+ > V_- \\ 0\text{ V} & \text{if } V_+ < V_- \end{cases}$$

assuming that  $V_+ \not\approx V_-$ . In case of  $V_+ \approx V_-$  we suffer from *meta stability*.

# ATmega32 analog comparator

The ATmega32 features an analog comparator<sup>6</sup>:

- ▶ It compares the voltage at positive pin AIN0 against the voltage of the negative pin AIN1.
- ▶ The comparator logic can raise a [analog comparator interrupt](#). It knows three different modes when to raise an interrupt:
  - ▶ Output toggle
  - ▶ Falling edge
  - ▶ Rising edge
- ▶ Also the timer/counter 1 [input capture](#) can be triggered.
- ▶ The ATmega32 can be configured to use any of the analog input pins ADC0 to ADC7 instead of AIN1 by setting the Analog Comperator Multiplexer Enable bit ACME and the ADC multiplexer select ADMUX bits.

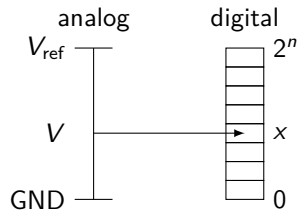
---

<sup>6</sup> ATmega32, p. 198.

# Analog-digital conversion

We convert the analog voltage  $V$  into an  $n$ -bit digital value  $x$ .

- ▶  $n$  is the **resolution**
- ▶ Typical resolutions are 8-bit to 14-bit, but sometimes also higher.

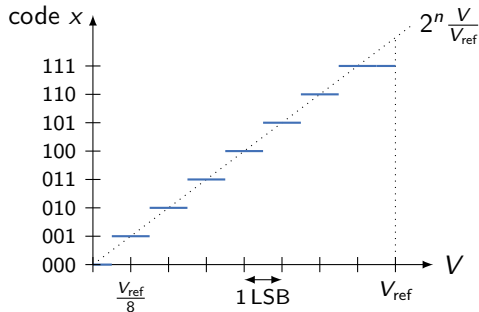


Converting  $x$  back to an analog voltage shall yield approximately  $V$  again:

$$V - x \cdot \frac{V_{\text{ref}}}{2^n} \approx 0 \quad \text{resp.} \quad x \approx 2^n \cdot \frac{V}{V_{\text{ref}}}$$

# Transfer function

The **coding function** or **transfer function** tells how the analog values are mapped to digital values.



The **least-significant bit** (1 LSB) of an ADC is the voltage of a “step size”:

$$1 \text{ LSB} = \frac{V_{ref}}{2^n}$$

Note that the step size at the beginning and end is not equal to 1 LSB.

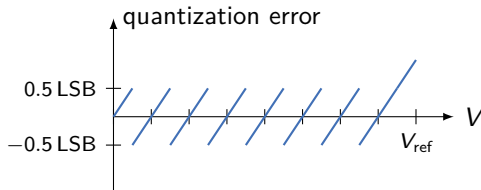
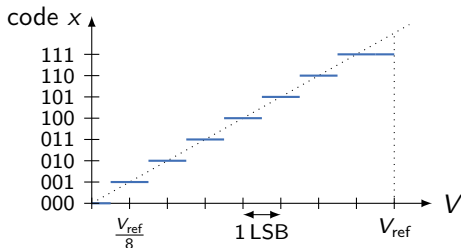
**Figure:** Example transfer function of a 3-bit ADC.

# Quantization error

- ▶ The **quantization error** is the rounding error introduced by the quantization; we define it as

$$V - x \cdot \text{LSB}.$$

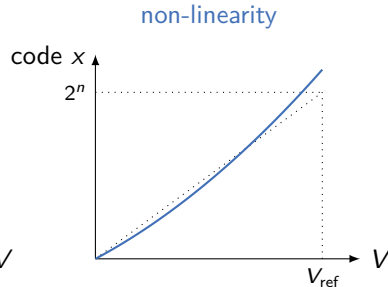
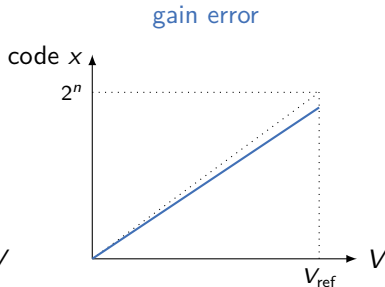
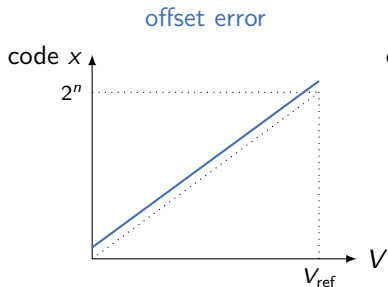
- ▶ The quantization error is between  $-0.5 \text{ LSB}$  and  $0.5 \text{ LSB}$  (except for  $V > V_{\text{ref}} - 0.5 \text{ LSB}$ ). It can be improved using **dithering**, where little noise is added to the input.





# Accuracy

The quantization error is naturally part of the game of analog-digital conversion. However, additional errors have further impact on accuracy:



- ▶ Offset and gain calibration eliminates (or reduces) errors.
- ▶ Temperature drifts add further complications.

# ADC techniques

**Parallel comparator** Uses  $2^n - 1$  comparators to directly determine the output code. Also known as **flash ADC**. Can achieve up to gigahertz sampling rates, but lower resolution.

**Successive approximation** Similar to **binary search** on the transfer function using one comparator and a DAC. We start with the interval  $[0, 2^n - 1]$  and in each step divide the range in two using a single comparator. It gives one bit of the output code at a time.

**Ramp compare** We produce a voltage ramp (saw-tooth signal) and **measure the time** until the input voltage  $V$  is hit. The ramp could be generated by charging a capacitor in an integrator circuit.

A **dual slope** ADC first charges the capacitor with the input voltage  $V$  for a constant time  $\tau$  (run-up period) and then discharges the capacitor with the reference voltage  $V_{\text{ref}}$  until ground potential is reached again (run-down period). The time for the run-down period tells us what  $V$  was. If we make  $\tau$  longer we can improve resolution at the expense of lower sample rates.

And there are many more.

# ADC techniques

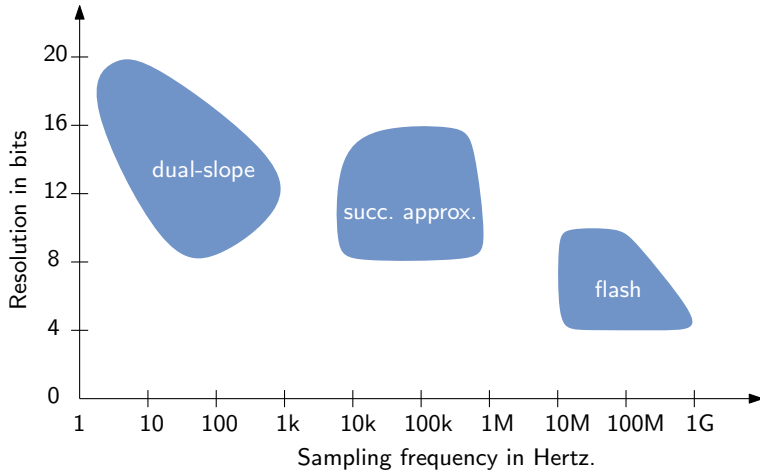


Figure: Resolution and sampling frequency of ADC techniques. Adopted from [TS99, fig. 18.21].

The ATmega32 feature a 10-bit successive approximation ADC.<sup>7</sup>

- ▶ 8 analog input pins can be **multiplexed** to the one ADC unit.
- ▶ Differential input channels with optional gains of 10 or 200 can be configured.
- ▶ Sample rates up to 15 kSPS (samples per seconds) at maximum resolution.
- ▶ Reference voltage is  $V_{cc}$ , an internal 2.56 V voltage, or an external voltage via pin AREF.

# ATmega32: Successive approximation

A sample-and-hold circuit keeps signal constant during conversion.

The ADC receives a clock signal, which also tells how many samples per seconds are retrieved.

- ▶ Higher sample rates increase noise and reduce (effective) resolution of the ADC.
- ▶ Noise can be reduced by putting the ATmega32 into sleep mode during conversion.
- ▶ A clock of 50 kHz to 200 kHz has to be used for maximum resolution.
- ▶ For less than 10 bit the clock can be higher than 200 kHz.
- ▶ The ADC clock prescaler is configured via register ADCSRA.

A conversion takes 13 ADC clock cycles. However, the first time after switch on it takes 25 cycles.

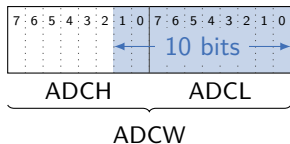
# ATmega32: Simple conversion

## Simple ADC conversion

- ▶ ADC is enabled by the `ADEN` bit in register `ADCSRA`.
- ▶ Setting the bit `ADSC` (start conversion) in register `ADCSRA` kicks off a single conversion.

When the `ADSC` bit is cleared again, the conversion is done:

- ▶ Of course, there is an ADC completed interrupt flag `ADIF` and an `ADIE` interrupt enable bit.
- ▶ The 10-bit result is found in registers `ADCH` and `ADCL`. In C we can simply access all bits via `ADCW`.



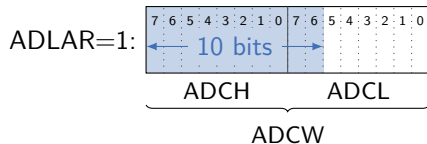
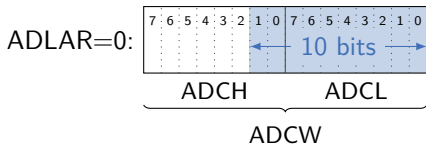
# ATmega32: Simple conversion

## Simple ADC conversion

- ▶ ADC is enabled by the `ADEN` bit in register `ADCSRA`.
- ▶ Setting the bit `ADSC` (start conversion) in register `ADCSRA` kicks off a single conversion.

When the `ADSC` bit is cleared again, the conversion is done:

- ▶ Of course, there is an ADC completed interrupt flag `ADIF` and an `ADIE` interrupt enable bit.
- ▶ The 10-bit result is found in registers `ADCH` and `ADCL`. In C we can simply access all bits via `ADCW`.
- ▶ If we are only interested in an 8 bit result, we can activate left adjustment by setting the `ADLAR` bit in register `ADMUX` and just read `ADCH`, which then contains the 8 most-significant bits.



# ATmega32: Starting conversion and auto-trigger

Two modes on starting ADC conversion:

- ▶ **Single conversion mode.** As explained before, by manually setting the start conversion bit ADSC in register ADCSRA for a single conversion.
- ▶ **Auto-trigger mode.** Certain events automatically trigger an ADC conversion.

We can select different event sources for the auto-trigger mode through ADTS bits in the SFIOR register<sup>8</sup>:

- ▶ Free running mode: Periodically running a conversion triggered by a prescaled clock<sup>9</sup>.
- ▶ Timer compare match or overflow or input capture event
- ▶ Analog comparator
- ▶ External interrupt

---

<sup>8</sup> See [ATmega32, p. 218].

<sup>9</sup> See [ATmega32, p. 204].



# ATmega32: ADC block diagram

[ATmega32, p. 202, fig. 98]

# ATmega32: ADC example

```
1 #include <avr/io.h>
2
3 uint16_t read() {
4     /* Start a conversion */
5     ADCSRA |= 1 << ADSC;
6     while (ADCSRA & (1 << ADSC));
7
8     /* ADCW contains ADCH and ADCL. */
9     return ADCW;
10 }
11
12 void init() {
13     /* Turn PA0 to an input pin. */
14     DDRA &= ~(1 << PA0);
15     /* Select Vcc as voltage reference, no left-adjust result, and select ADC0 in
16      * the multiplexer. */
17     ADMUX = ...;
18     /* Enable ADC, no auto-trigger, no interrupt, a clock prescaler of 128. */
19     ADCSRA = ...;
20 }
```

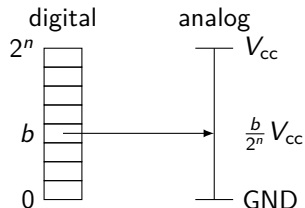
- [ATmega32] *ATmega32: 8-bit AVR Microcontroller with 32KBytes In-System Programmable Flash.* Atmel Corporation. Feb. 2011.
- [BCM2835] *BCM2835 ARM Peripherals.* Broadcom Corporation. 2012. URL: <https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf>.
- [TS99] Ulrich Tietze and Christoph Schenk. *Halbleiter-Schaltungstechnik.* Springer-Verlag GmbH, 1999. ISBN: 3-540-64192-0.

# DAC via resistor networks

We convert the digital value  $b$  into an analog voltage:

- ▶  $b$  is a  $n$ -bit value output via digital output pins.
- ▶ Hence, the **output voltage** shall be

$$\frac{b}{2^n} V_{cc}$$



We consider  $b$  as a number to basis 2 with digits  $b_k$ :

$$b = b_{n-1} 2^{n-1} + b_{n-2} 2^{n-2} + \dots + b_2 2^2 + b_1 2^1 + b_0 2^0 = \sum_{k=0}^{n-1} b_k 2^k.$$

Hence, the output voltage shall be

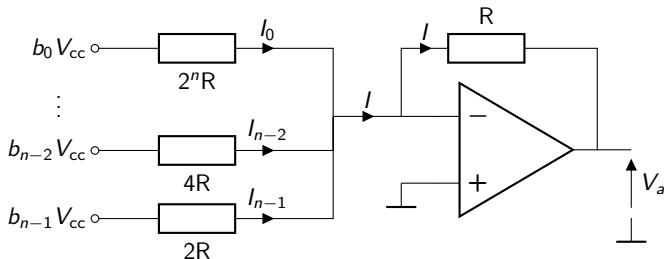
$$\sum_{k=0}^{n-1} \frac{2^k}{2^n} \cdot b_k V_{cc}.$$

# DAC by resistor networks

The output voltage is a **weighted sum** of digital voltages:

$$\sum_{k=0}^{n-1} \underbrace{\frac{2^k}{2^n}}_{\text{weight}} \cdot \underbrace{b_k V_{cc}}_{\text{digital I/O}} = \frac{1}{2^n} \cdot b_0 V_{cc} + \frac{1}{2^{n-1}} \cdot b_1 V_{cc} + \dots + \frac{1}{2} \cdot b_{n-1} V_{cc}$$

Weighted sums are analogously computed by an amplifier circuit:



$$\begin{aligned} V_a &= R \cdot I \\ &= R \cdot \sum_k I_k = R \cdot \sum_k \frac{b_k V_{cc}}{2^{n-k} R} \\ &= \sum_k \frac{2^k}{2^n} \cdot b_k V_{cc}. \end{aligned}$$