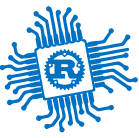


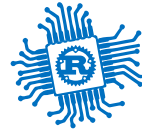
# Interrupts, PWM and ADC

Lecture 3



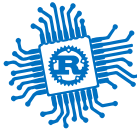
# Interrupts, PWM and ADC

- Interrupts
- Counters
- Timers and Alarms
- About Analog and Digital Signals
- Pulse Width Modulation (PWM)
- Analog to Digital Converters (ADC)



# Exceptions

for the ARM Cortex-M33 processor

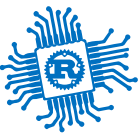


# Bibliography

for this section

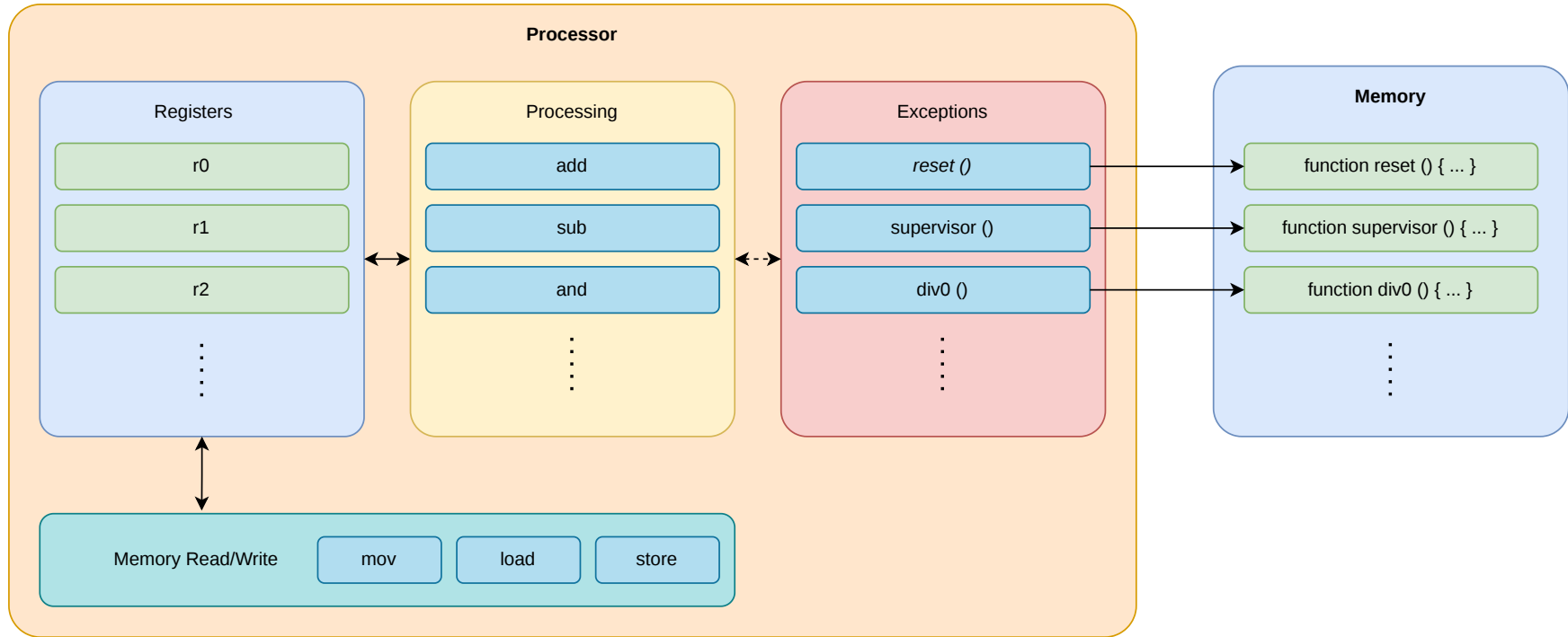
**Joseph Yiu**, *The Definitive Guide to ARM® Cortex®-M23 and Cortex-M33 Processors*

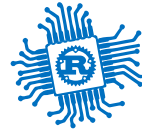
- Chapter 4 - *Architecture*
  - Section 4.5 - *Exceptions and Interrupts*
    - Subsection 4.4.1 - *What are exceptions*
- Chapter 8 - *Exceptions and Interrupts*
  - Section 8.1 - *What are Exceptions and Interrupts*
  - Section 8.2 - *Exception types+*



# Processor Exceptions

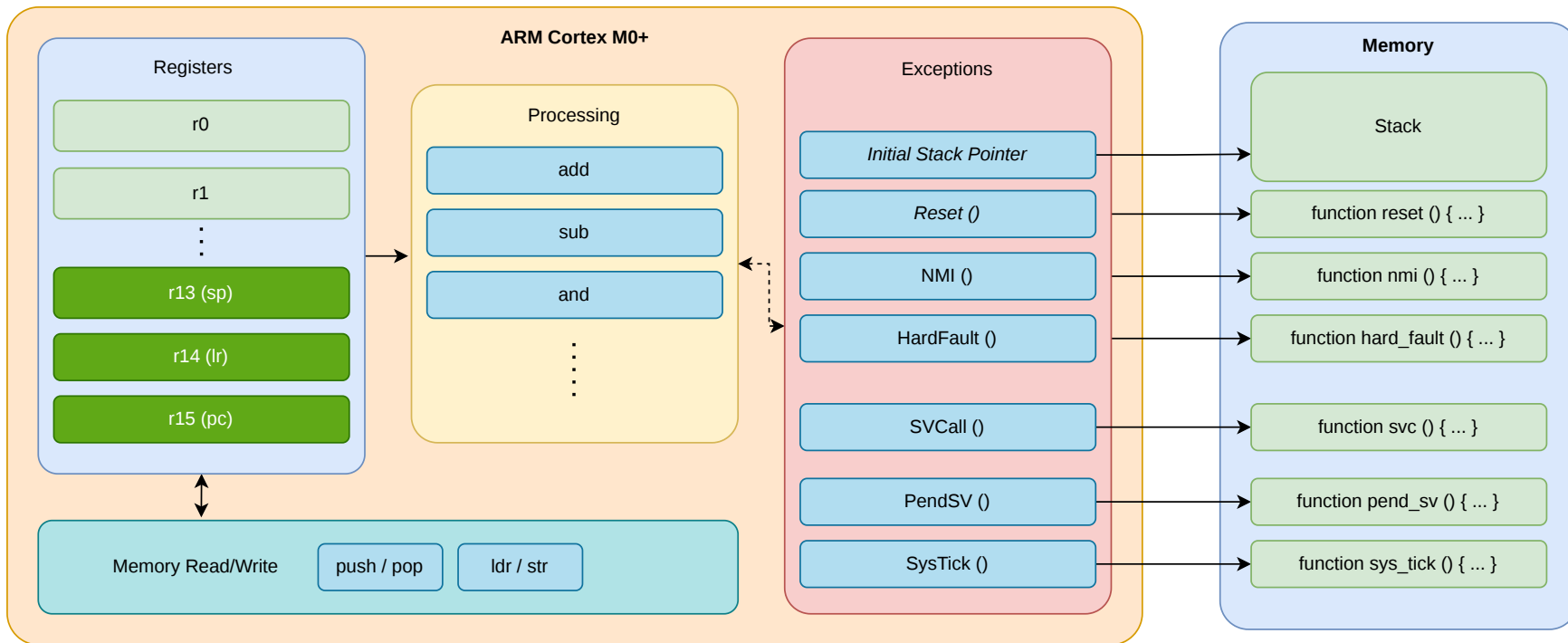
what happens if something does not work as required





# Standard ARM Cortex-M Exceptions

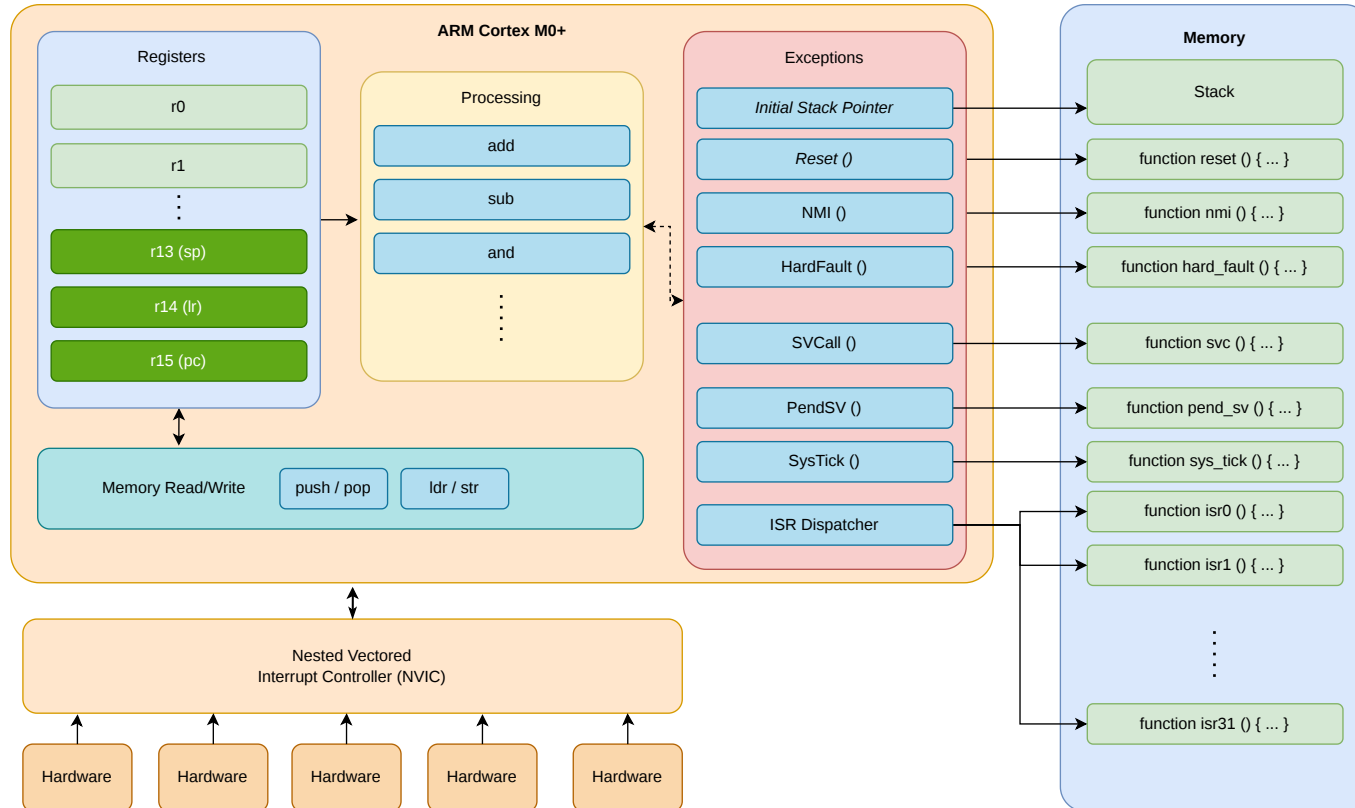
what happens if something does not work as required

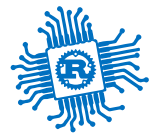




# ARM Cortex-M Interrupts

some hardware device notifies the MCU





# Exceptions and Interrupts in Embassy

- Embassy registers handlers for Exceptions
- Each of the Embassy drivers that you use provides interrupt handlers for the peripheral they control
- Developers have to **bind** interrupts to the driver.

| IRQ | Interrupt Source | IRQ | Interrupt Source | IRQ | Interrupt Source | IRQ | Interrupt Source | IRQ | Interrupt Source |
|-----|------------------|-----|------------------|-----|------------------|-----|------------------|-----|------------------|
| 0   | TIMER0_IRQ_0     | 11  | DMA_IRQ_1        | 22  | IO_IRQ_BANK0_NS  | 33  | UART0_IRQ        | 44  | POWMAN_IRQ_POW   |
| 1   | TIMER0_IRQ_1     | 12  | DMA_IRQ_2        | 23  | IO_IRQ_QSPI      | 34  | UART1_IRQ        | 45  | POWMAN_IRQ_TIMER |
| 2   | TIMER0_IRQ_2     | 13  | DMA_IRQ_3        | 24  | IO_IRQ_QSPI_NS   | 35  | ADC_IRQ_FIFO     | 46  | SPAREIRQ_IRQ_0   |

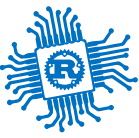
List of some of the RP2350's interrupts

Register the Interrupt

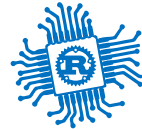
```
bind_interrupts!(struct Irqs {  
    ADC_IRQ_FIFO => InterruptHandler;  
});
```

Bind it to the driver

```
let mut adc = Adc::new(p.ADC, Irqs, Config::default());
```



# Timers

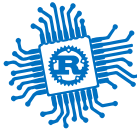


# Bibliography

for this section

## **Raspberry Pi Ltd, *RP2350 Datasheet***

- Chapter 8 - *Clocks*
  - Chapter 8.1 - *Overview*
    - Subchapter 8.1.1
    - Subchapter 8.1.2
- Chapter 12 - *Peripherals*
  - Chapter 12.8 - *System Timers*



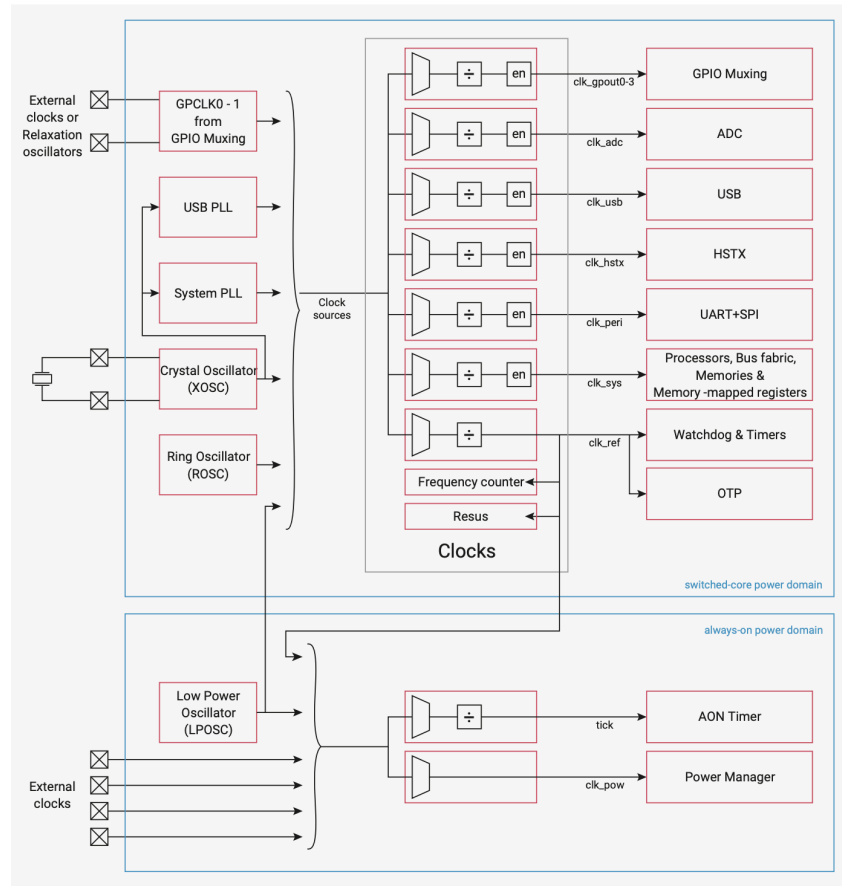
# Clocks

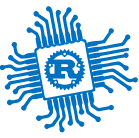
all peripherals and the MCU use a clock to execute at certain intervals

| Source                         | Usage                                                       |
|--------------------------------|-------------------------------------------------------------|
| <i>external crystal (XOSC)</i> | a stable frequency is required, for instance when using USB |
| <i>internal ring (ROSC)</i>    | low frequency, in between 1.8 - 12 MHz (varies)             |

Embassy initializes the Raspberry Pi Pico with the clock source from the 12 MHz crystal.

```
1 let p = embassy_rp::init(Default::default());
```

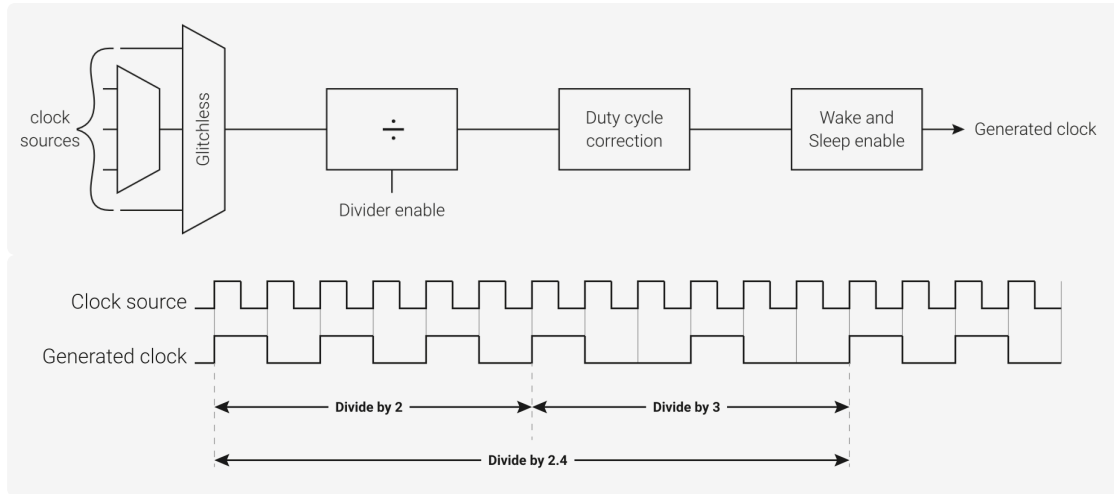




# Frequency divider

stabilizing the signal and adjusting it

1. divides down the clock signals used for the timer, giving reduced overflow rates
2. allows the timer to be clocked at a user desired rate

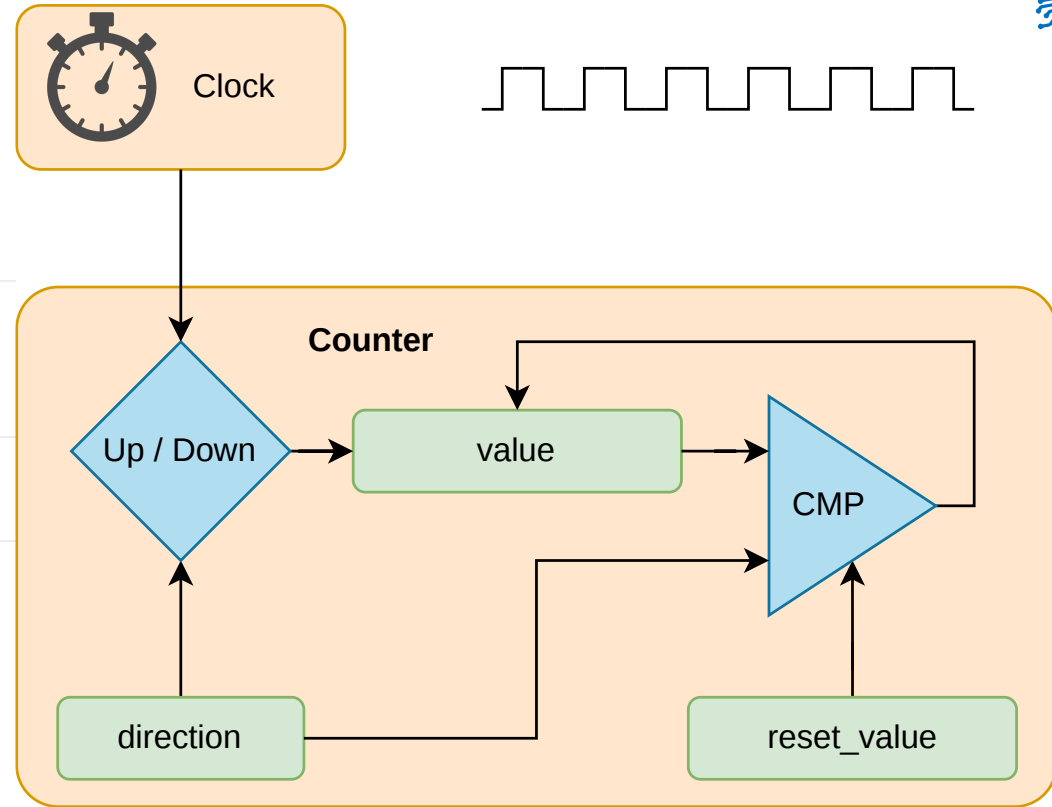


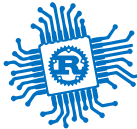


# Counter

increments a register at every clock cycle

| Registers | Description                                                                                                      |
|-----------|------------------------------------------------------------------------------------------------------------------|
| value     | the current value of the counter                                                                                 |
| direction | set to count UP or DOWN                                                                                          |
| reset     | UP: the value at which the counter resets to 0<br>DOWN: the value to which the counter resets after getting to 0 |





# SysTick

## ARM Cortex-M time counter

The ARM Cortex-M0+ registers start at a base address of `0xe0000000` (defined as `PPB_BASE` in SDK).

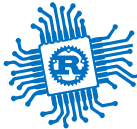
| Offset | Name                       | Info                                |
|--------|----------------------------|-------------------------------------|
| 0xe010 | <a href="#">SYST_CSR</a>   | SysTick Control and Status Register |
| 0xe014 | <a href="#">SYST_RVR</a>   | SysTick Reload Value Register       |
| 0xe018 | <a href="#">SYST_CVR</a>   | SysTick Current Value Register      |
| 0xe01c | <a href="#">SYST_CALIB</a> | SysTick Calibration Value Register  |

- decrements the value of `SYST_CVR` every  $\mu s$
- when `SYST_CVR` becomes 0 :
  - triggers the SysTick exception
  - next clock cycle sets the value of `SYST_CVR` to `SYST_RVR`
- `SYST_CALIB` is the value of `SYST_RVR` for a 10ms interval (might not be available)

## SYST\_CSR register

| Bits  | Name      | Description                                                                                                                                                                           | Type | Reset |
|-------|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|-------|
| 31:17 | Reserved. | -                                                                                                                                                                                     | -    | -     |
| 16    | COUNTFLAG | Returns 1 if timer counted to 0 since last time this was read. Clears on read by application or debugger.                                                                             | RO   | 0x0   |
| 15:3  | Reserved. | -                                                                                                                                                                                     | -    | -     |
| 2     | CLKSOURCE | SysTick clock source. Always reads as one if SYST_CALIB reports NOREF.<br>Selects the SysTick timer clock source:<br>0 = External reference clock.<br>1 = Processor clock.            | RW   | 0x0   |
| 1     | TICKINT   | Enables SysTick exception request:<br>0 = Counting down to zero does not assert the SysTick exception request.<br>1 = Counting down to zero to asserts the SysTick exception request. | RW   | 0x0   |
| 0     | ENABLE    | Enable SysTick counter:<br>0 = Counter disabled.<br>1 = Counter enabled.                                                                                                              | RW   | 0x0   |

$$f = \frac{1}{SYST\_RVR} * 1,000,000 [Hz]_{SI}$$



# SysTick

## ARM Cortex-M peripheral

The ARM Cortex-M0+ registers start at a base address of `0xe0000000` (defined as `PPB_BASE` in SDK).

| Offset | Name                       | Info                                |
|--------|----------------------------|-------------------------------------|
| 0xe010 | <a href="#">SYST_CSR</a>   | SysTick Control and Status Register |
| 0xe014 | <a href="#">SYST_RVR</a>   | SysTick Reload Value Register       |
| 0xe018 | <a href="#">SYST_CVR</a>   | SysTick Current Value Register      |
| 0xe01c | <a href="#">SYST_CALIB</a> | SysTick Calibration Value Register  |

```
1  const SYST_RVR: *mut u32 = 0xe000_e014 as *mut u32;
2  const SYST_CVR: *mut u32 = 0xe000_e018 as *mut u32;
3  const SYST_CSR: *mut u32 = 0xe000_e010 as *mut u32;
4
5  // fire systick every 5 seconds
6  let interval: u32 = 5_000_000;
7  unsafe {
8      write_volatile(SYST_RVR, interval);
9      write_volatile(SYST_CVR, 0);
10     // set fields `ENABLE` and `TICKINT`
11     write_volatile(SYST_CSR, 0b11);
12 }
```

## SYST\_CSR register

| Bits  | Name      | Description                                                                                                                                                                           | Type | Reset |
|-------|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|-------|
| 31:17 | Reserved. | -                                                                                                                                                                                     | -    | -     |
| 16    | COUNTFLAG | Returns 1 if timer counted to 0 since last time this was read. Clears on read by application or debugger.                                                                             | RO   | 0x0   |
| 15:3  | Reserved. | -                                                                                                                                                                                     | -    | -     |
| 2     | CLKSOURCE | SysTick clock source. Always reads as one if SYST_CALIB reports NOREF.<br>Selects the SysTick timer clock source:<br>0 = External reference clock.<br>1 = Processor clock.            | RW   | 0x0   |
| 1     | TICKINT   | Enables SysTick exception request:<br>0 = Counting down to zero does not assert the SysTick exception request.<br>1 = Counting down to zero to asserts the SysTick exception request. | RW   | 0x0   |
| 0     | ENABLE    | Enable SysTick counter:<br>0 = Counter disabled.<br>1 = Counter enabled.                                                                                                              | RW   | 0x0   |

## Register SysTick handler

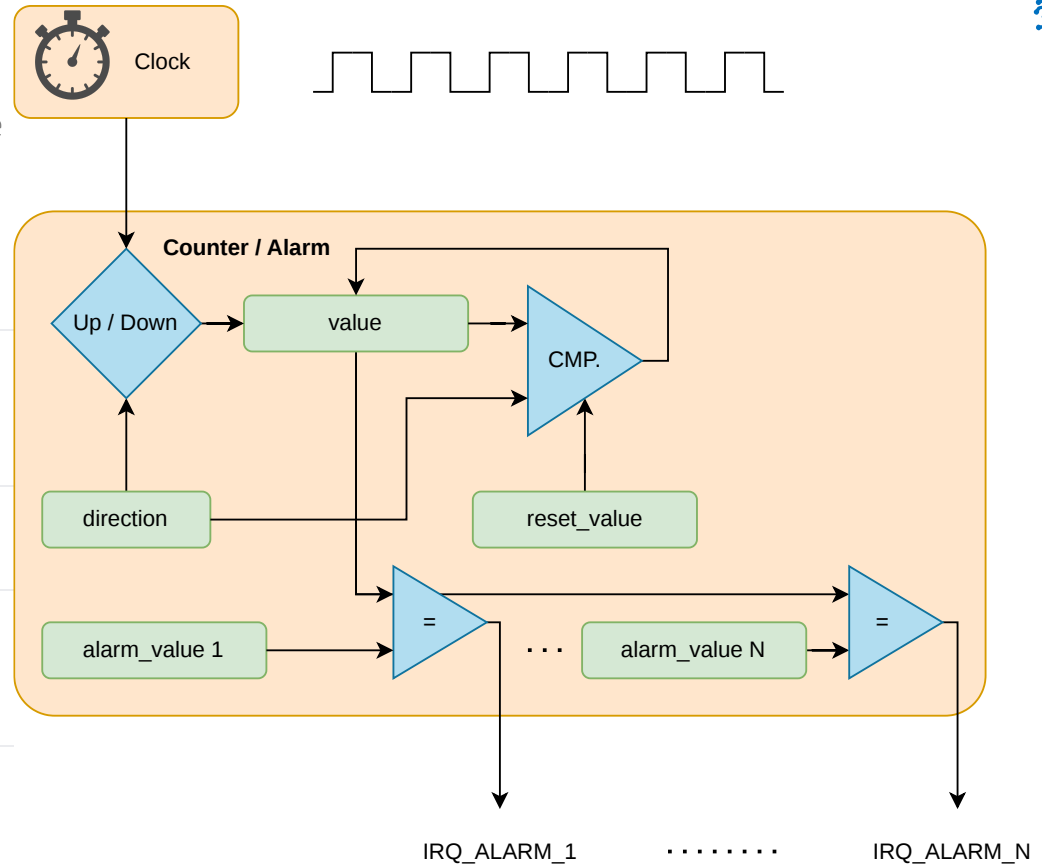
```
1  #[exception]
2  unsafe fn SysTick() {
3      /* systick fired */
4  }
```

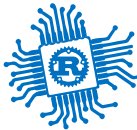


# Alarm

counter that triggers interrupts after a time interval

| Registers              | Description                                                                           |
|------------------------|---------------------------------------------------------------------------------------|
| <code>value</code>     | the current value of the counter                                                      |
| <code>direction</code> | set to count UP or DOWN                                                               |
| <code>reset</code>     | UP: max value before 0<br>DOWN: value after 0                                         |
| <code>alarm_x</code>   | when <code>value == alarm_x</code> , triggers an interrupt, <code>x</code> in 1 ... n |

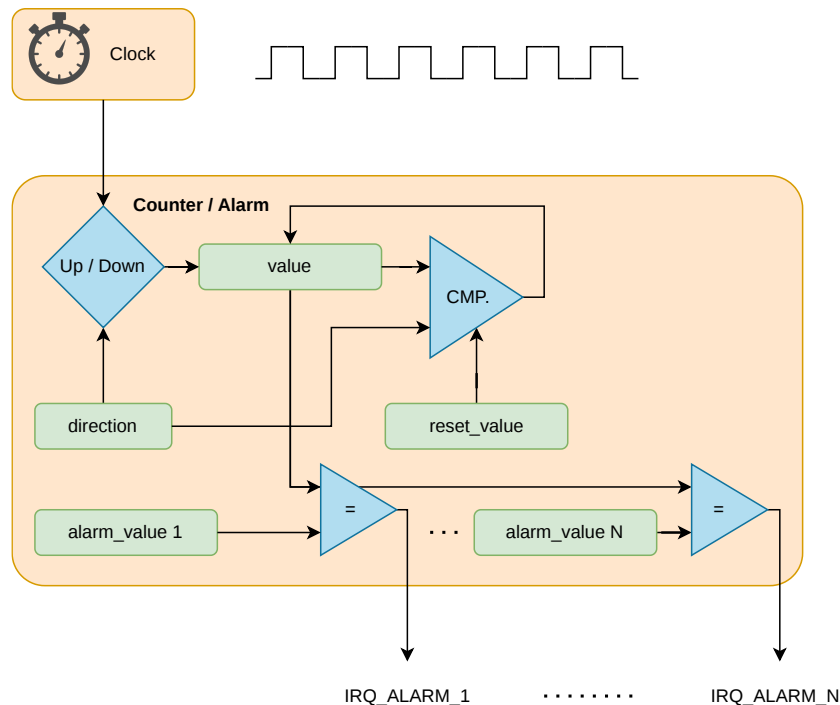


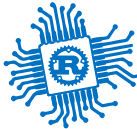


# RP2350's Timers

two timers, `TIMER0` and `TIMER1`

- store a 64 bit number ( `reset` is  $2^{64}-1$  )
- start with `0` at (the peripheral's) reset
- increment the number every  $\mu\text{s}$
- in practice fully monotonic (cannot over flow)
- allow 4 alarms that trigger interrupts
  - `TIMER0_IRQ_0` and `TIMER1_IRQ_0`
  - `TIMER0_IRQ_1` and `TIMER1_IRQ_1`
  - `TIMER0_IRQ_2` and `TIMER1_IRQ_2`
  - `TIMER0_IRQ_3` and `TIMER1_IRQ_3`
- `alarm_0 ... alarm_3` registers are only 32 bits wide





# RP2350's Timer instance

read the number of elapsed  $\mu$ s since reset

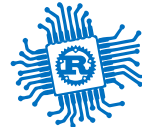
## Reading the time elapsed since restart

```
1  const TIMERLR: *const u32 = 0x400b_000c;
2  const TIMERHR: *const u32 = 0x400b_0008;
3
4  let time: u64 = unsafe {
5      let low = read_volatile(TIMERLR);
6      let high = read_volatile(TIMERHR);
7      high as u64 << 32 | low
8  }
```

The **reading order matters** as reading `TIMERLR` latches the value in `TIMERHR` (stops being updated) until `TIMERHR` is read. Works only in **single core**.

The `TIMER0` and `TIMER1` registers start at base addresses of `0x400b0000` and `0x400b8000` respectively (defined as `TIMER0_BASE` and `TIMER1_BASE` in SDK).

| Offset | Name                  | Info                                                                                                                                                                                                                                               |
|--------|-----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0x00   | <code>TIMEHW</code>   | Write to bits 63:32 of time<br>always write timelw before timehw                                                                                                                                                                                   |
| 0x04   | <code>TIMELW</code>   | Write to bits 31:0 of time<br>writes do not get copied to time until timehw is written                                                                                                                                                             |
| 0x08   | <code>TIMEHR</code>   | Read from bits 63:32 of time<br>always read timelr before timehr                                                                                                                                                                                   |
| 0x0c   | <code>TIMERLR</code>  | Read from bits 31:0 of time                                                                                                                                                                                                                        |
| 0x10   | <code>ALARM0</code>   | Arm alarm 0, and configure the time it will fire.<br>Once armed, the alarm fires when <code>TIMER_ALARM0 == TIMERLR</code> .<br>The alarm will disarm itself once it fires, and can be disarmed early using the ARMED status register.             |
| 0x14   | <code>ALARM1</code>   | Arm alarm 1, and configure the time it will fire.<br>Once armed, the alarm fires when <code>TIMER_ALARM1 == TIMERLR</code> .<br>The alarm will disarm itself once it fires, and can be disarmed early using the ARMED status register.             |
| 0x18   | <code>ALARM2</code>   | Arm alarm 2, and configure the time it will fire.<br>Once armed, the alarm fires when <code>TIMER_ALARM2 == TIMERLR</code> .<br>The alarm will disarm itself once it fires, and can be disarmed early using the ARMED status register.             |
| 0x1c   | <code>ALARM3</code>   | Arm alarm 3, and configure the time it will fire.<br>Once armed, the alarm fires when <code>TIMER_ALARM3 == TIMERLR</code> .<br>The alarm will disarm itself once it fires, and can be disarmed early using the ARMED status register.             |
| 0x20   | <code>ARMED</code>    | Indicates the armed/disarmed status of each alarm.<br>A write to the corresponding <code>ALARMx</code> register arms the alarm.<br>Alarms automatically disarm upon firing, but writing ones here will disarm immediately without waiting to fire. |
| 0x24   | <code>TIMERAWH</code> | Raw read from bits 63:32 of time (no side effects)                                                                                                                                                                                                 |
| 0x28   | <code>TIMERAWL</code> | Raw read from bits 31:0 of time (no side effects)                                                                                                                                                                                                  |
| 0x2c   | <code>DBGPAUSE</code> | Set bits high to enable pause when the corresponding debug ports are active                                                                                                                                                                        |
| 0x30   | <code>PAUSE</code>    | Set high to pause the timer                                                                                                                                                                                                                        |
| 0x34   | <code>LOCKED</code>   | Set locked bit to disable write access to timer<br>Once set, cannot be cleared (without a reset)                                                                                                                                                   |



# Alarm

triggering an interrupt at an interval

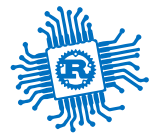
```
1  #[interrupt]
2  unsafe fn TIMER0_IRQ_0() { /* alarm fired */ }

1  const TIMERLR: *const u32 = 0x400b_000c;
2  const ALARM0: *mut u32 = 0x400b_0010;
3  // + 0x2000 is bitwise set
4  const INTE_SET: *mut u32 = 0x400b_0040;
5
6  // set an alarm after 3 seconds
7  let us = 3_0000_0000;
8
9  unsafe {
10     let time = read_volatile(TIMERLR);
11     // use `wrapping_add` as overflowing may panic
12     write_volatile(ALARM0, time.wrapping_add(us));
13     write_volatile(INTE_SET, 1 << 0);
14 };
```

- the alarm can be set only for the lower 32 bits
- maximum 72 minutes (use *RTC* for longer alarms)

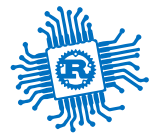
The *TIMER0* and *TIMER1* registers start at base addresses of *0x400b0000* and *0x400b8000* respectively (defined as *TIMER0\_BASE* and *TIMER1\_BASE* in SDK).

| Offset | Name                   | Info                                                                                                                                                                                                                                  |
|--------|------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0x00   | <a href="#">TIMEHW</a> | Write to bits 63:32 of time<br>always write timelw before timehw                                                                                                                                                                      |
| 0x04   | <a href="#">TIMELW</a> | Write to bits 31:0 of time<br>writes do not get copied to time until timehw is written                                                                                                                                                |
| 0x08   | <a href="#">TIMEHR</a> | Read from bits 63:32 of time<br>always read timelr before timehr                                                                                                                                                                      |
| 0x0c   | <a href="#">TIMELR</a> | Read from bits 31:0 of time                                                                                                                                                                                                           |
| 0x10   | <a href="#">ALARM0</a> | Arm alarm 0, and configure the time it will fire.<br>Once armed, the alarm fires when <code>TIMER_ALARM0 == TIMELR</code> .<br>The alarm will disarm itself once it fires, and can be disarmed early using the ARMED status register. |
| 0x14   | <a href="#">ALARM1</a> | Arm alarm 1, and configure the time it will fire.<br>Once armed, the alarm fires when <code>TIMER_ALARM1 == TIMELR</code> .<br>The alarm will disarm itself once it fires, and can be disarmed early using the ARMED status register. |
| 0x18   | <a href="#">ALARM2</a> | Arm alarm 2, and configure the time it will fire.<br>Once armed, the alarm fires when <code>TIMER_ALARM2 == TIMELR</code> .<br>The alarm will disarm itself once it fires, and can be disarmed early using the ARMED status register. |
| 0x1c   | <a href="#">ALARM3</a> | Arm alarm 3, and configure the time it will fire.<br>Once armed, the alarm fires when <code>TIMER_ALARM3 == TIMELR</code> .<br>The alarm will disarm itself once it fires, and can be disarmed early using the ARMED status register. |
| Offset | Name                   | Info                                                                                                                                                                                                                                  |
| 0x38   | <a href="#">SOURCE</a> | Selects the source for the timer. Defaults to the normal tick configured in the ticks block (typically configured to 1 microsecond). Writing to 1 will ignore the tick and count clk_sys cycles instead.                              |
| 0x3c   | <a href="#">INTR</a>   | Raw Interrupts                                                                                                                                                                                                                        |
| 0x40   | <a href="#">INTE</a>   | Interrupt Enable                                                                                                                                                                                                                      |
| 0x44   | <a href="#">INTF</a>   | Interrupt Force                                                                                                                                                                                                                       |
| 0x48   | <a href="#">INTS</a>   | Interrupt status after masking & forcing                                                                                                                                                                                              |



# Signals

Digital Signals - Recap



# Signals

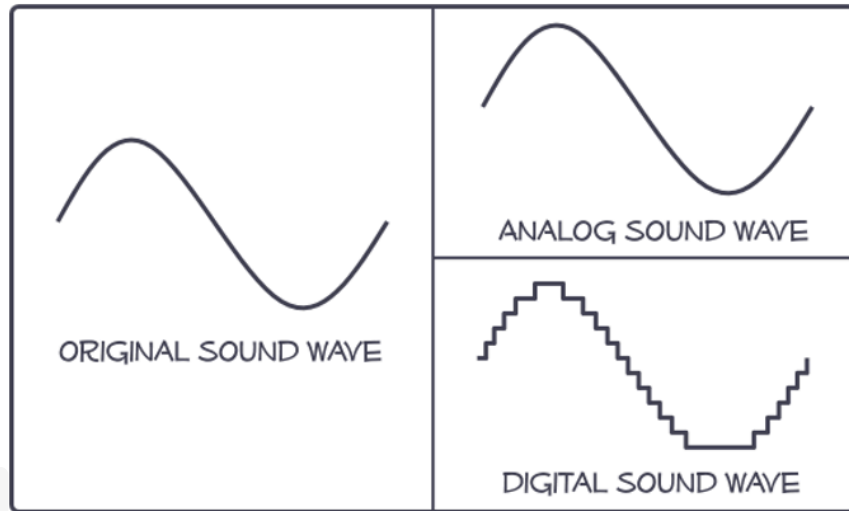
## Analog vs Digital

- *analog signals* are *real signals*
- *digital signals* are a *numerical representation* of an analog signal (software level)
- hardware usually works with two-level digital signals (hardware level)

### Exceptions

- in wireless and in high-speed cable communication things get more complicated

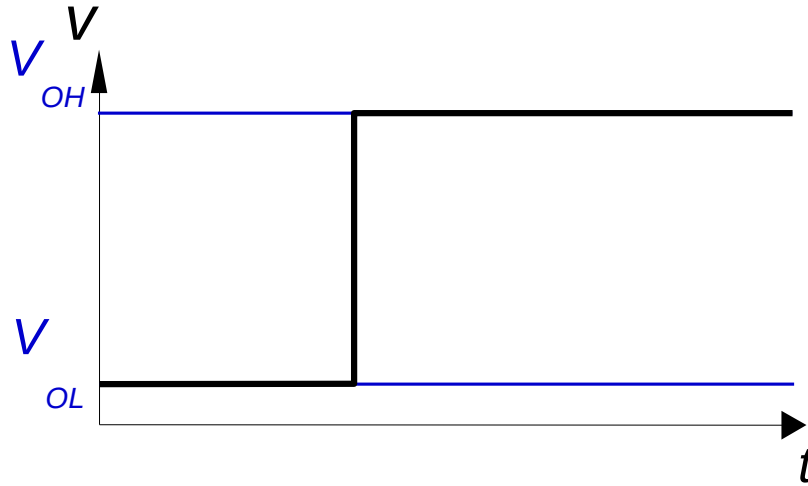
for PCB level / between integrated circuits on the same board / inside the same chip - things are a "a little simpler" - as detailed in the following



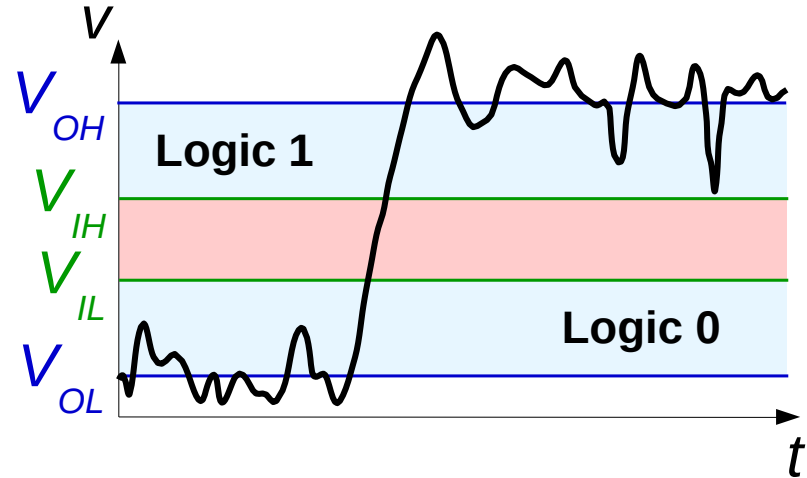


# Why use digital in computing?

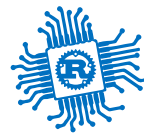
Signal that we *want* to generate with an output pin



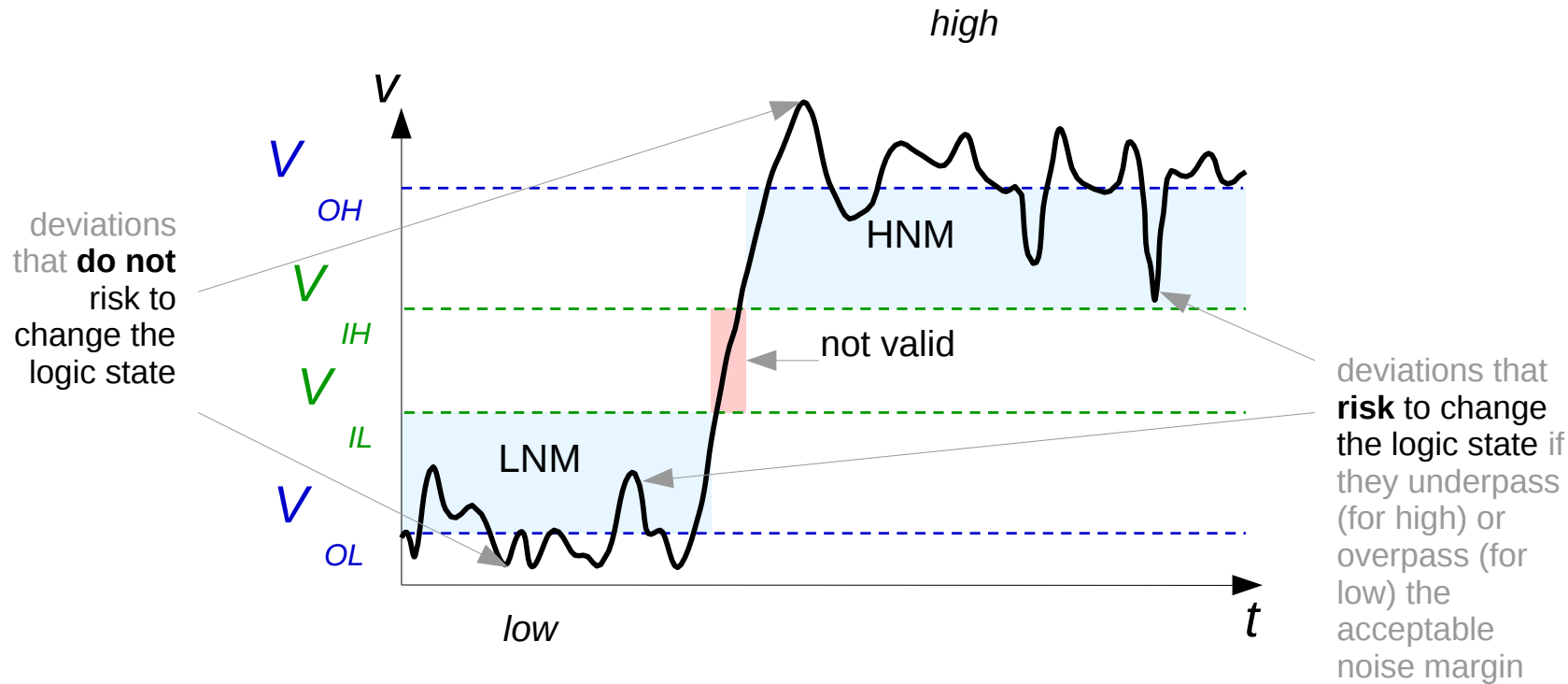
Signal that what we actually generate



Why we still use it? Because after passing through an IC or a gate inside an IC - the signal is "rebuilt" and if the "digital discipline" described in the following is respected - we can preserve the information after numerous "passes". Thus, each element can behave with a large margin for error, yet the final result is correct.



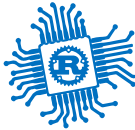
# Noise Margin





# PWM

Pulse Width Modulation



# Bibliography

for this section

## 1. **Raspberry Pi Ltd**, *RP2350 Datasheet*

- Chapter 12 - *Peripherals*
  - Section 12.5 - *PWM*

## 2. **Paul Denisowski**, *Understanding PWM*



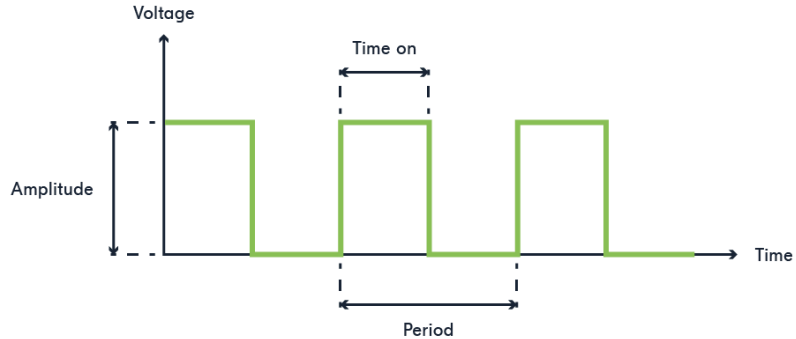
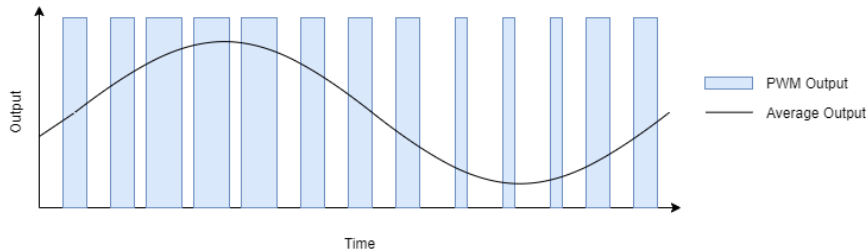
# PWM

simulates an *analog* signal (using integration)

- generates a square signal
- if integrated (averaged), it looks like an analog signal

*frequency*    Hz    The number of repeats per s

*duty\_cycle*    %    The percentage of the time when the signal is High



$$f = \frac{1}{\text{period}} \left[ \frac{1}{s} = 1Hz \right]_{SI}$$

$$\text{duty\_cycle} = \frac{\text{time\_on}}{\text{period}} \%$$

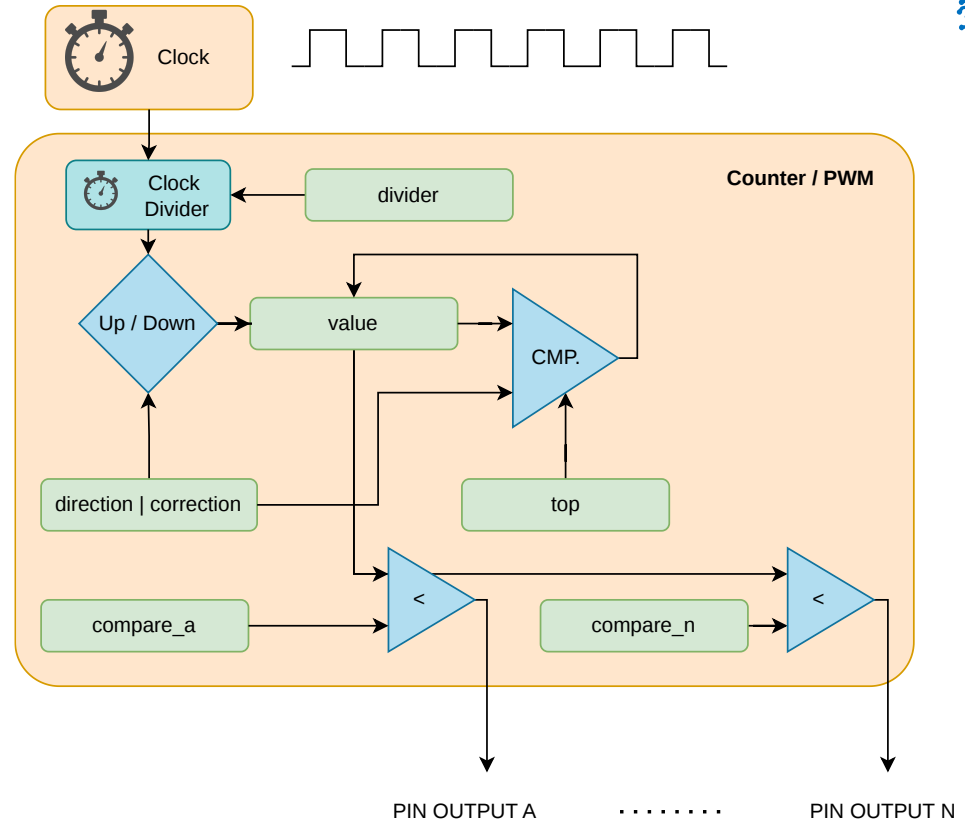


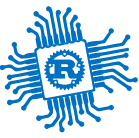
# PWM

generic device

$$f = \begin{cases} \frac{f_{clock}}{divider \times (top+1)} & correction = 0 \\ \frac{f_{clock}}{divider \times 2 \times (top+1)} & correction = 1 \end{cases}$$

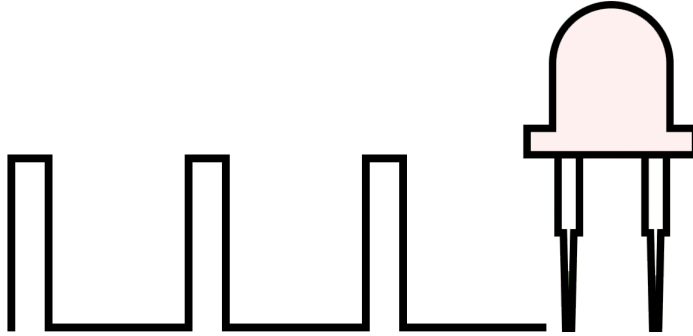
$$pin_{a,b} = \begin{cases} 0 & compare_{a,b} \geq value \\ 1 & compare_{a,b} < value \end{cases}$$



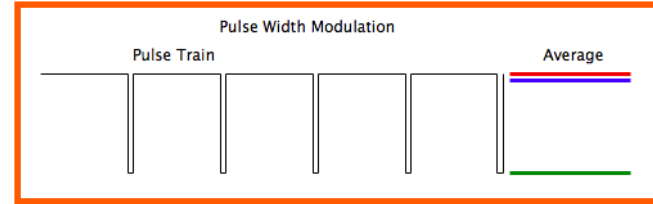


# Usage examples

- dimming an LED



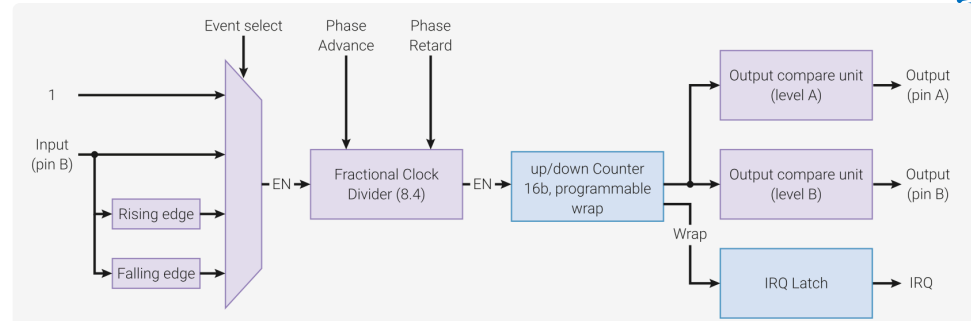
- controlling motors
  - controlling the angle of a stepper motor
  - controlling the RPM of a motor





# RP2350's PWM

- generates square signals
- counts the pulse width of input signals
- 8 or 12<sup>[1]</sup> PWM slices, each A and B channels
- each PWM channel is linked to a fixed pin
- some channels are connected to two pins
- may be used as timers ( `IRQ1_INTE` )

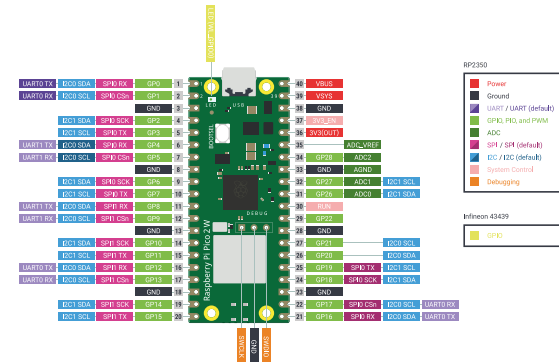


|             |    |    |    |    |     |     |     |     |    |    |    |    |     |     |     |     |
|-------------|----|----|----|----|-----|-----|-----|-----|----|----|----|----|-----|-----|-----|-----|
| GPIO        | 0  | 1  | 2  | 3  | 4   | 5   | 6   | 7   | 8  | 9  | 10 | 11 | 12  | 13  | 14  | 15  |
| PWM Channel | 0A | 0B | 1A | 1B | 2A  | 2B  | 3A  | 3B  | 4A | 4B | 5A | 5B | 6A  | 6B  | 7A  | 7B  |
| GPIO        | 16 | 17 | 18 | 19 | 20  | 21  | 22  | 23  | 24 | 25 | 26 | 27 | 28  | 29  | 30  | 31  |
| PWM Channel | 0A | 0B | 1A | 1B | 2A  | 2B  | 3A  | 3B  | 4A | 4B | 5A | 5B | 6A  | 6B  | 7A  | 7B  |
| GPIO        | 32 | 33 | 34 | 35 | 36  | 37  | 38  | 39  | 40 | 41 | 42 | 43 | 44  | 45  | 46  | 47  |
| PWM Channel | 8A | 8B | 9A | 9B | 10A | 10B | 11A | 11B | 8A | 8B | 9A | 9B | 10A | 10B | 11A | 11B |

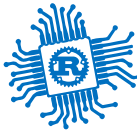
## Registers

The PWM registers start at a base address of `0x400a8000` (defined as `PWM_BASE` in the SDK).

| Offset | Name                 | Info                                                                                                                                                                      |
|--------|----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0x000  | <code>CH0_CSR</code> | Control and status register                                                                                                                                               |
| 0x004  | <code>CH0_DIV</code> | INT and FRAC form a fixed-point fractional number. Counting rate is system clock frequency divided by this number. Fractional division uses simple 1st-order sigma-delta. |
| 0x008  | <code>CH0_CTR</code> | Direct access to the PWM counter                                                                                                                                          |
| 0x00c  | <code>CH0_CC</code>  | Counter compare values                                                                                                                                                    |
| 0x010  | <code>CH0_TOP</code> | Counter wrap value                                                                                                                                                        |

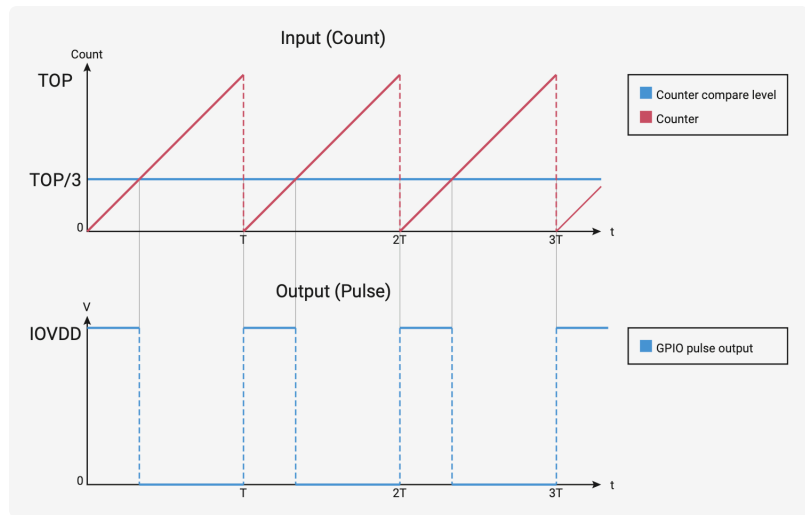


1. Depends on the RP2350 package ↩

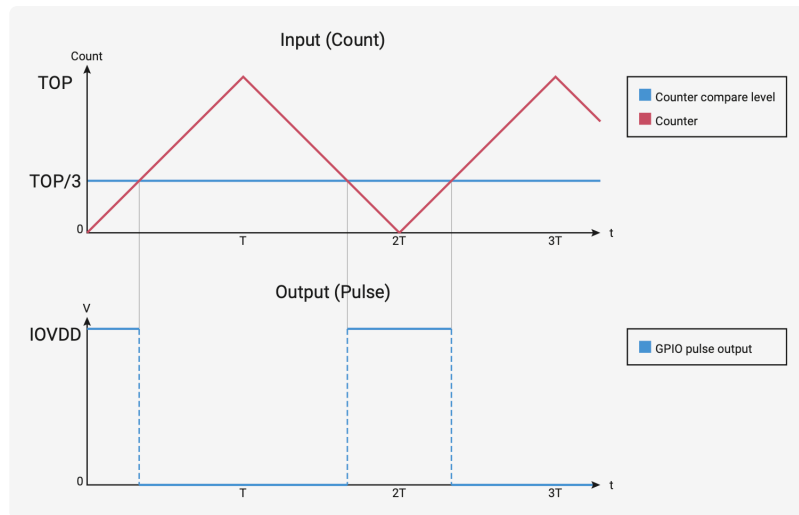


# RP2350's PWM Modes

standard mode



phase-correct mode



$$period = (TOP + 1) \times (PH\_CORRECT + 1) \times \left( DIV\_INT + \frac{DIV\_FRAC}{16} \right) [s]_{SI}$$

$$f = \frac{f_{sys}}{period} [Hz]_{SI}$$



# Example

using Embassy

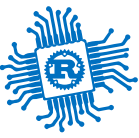
```
1 use embassy_rp::pwm::{Config, Pwm};
2
3 let p = embassy_rp::init(Default::default());
4
5 let mut c: Config = Default::default();
6 c.top = 0x8000;
7 c.compare_b = 8;
8
9 let mut pwm = Pwm::new_output_b(
10     p.PWM_CH4,
11     p.PIN_25,
12     c.clone()
13 );
14
15 loop {
16     info!("LED duty cycle: {}/32768", c.compare_b);
17     Timer::after_secs(1).await;
18     c.compare_b += 10;
19     pwm.set_config(&c);
20 }
```

```
pub struct Config {
    /// Inverts the PWM output signal on channel A.
    pub invert_a: bool,
    /// Inverts the PWM output signal on channel B.
    pub invert_b: bool,
    /// Enables phase-correct mode for PWM operation.
    pub phase_correct: bool,
    /// Enables the PWM slice, allowing it to generate an out
    pub enable: bool,
    /// A fractional clock divider, represented as a fixed-po
    /// 8 integer bits and 4 fractional bits. It allows preci
    /// the PWM output frequency by gating the PWM counter in
    /// A higher value will result in a slower output frequen
    pub divider: fixed::FixedU16<fixed::types::extra::U4>,
    /// The output on channel A goes high when `compare_a` is
    /// counter. A compare of 0 will produce an always low ou
    pub compare_a: u16,
    /// The output on channel B goes high when `compare_b` is
    /// counter.
    pub compare_b: u16,
    /// The point at which the counter wraps, representing th
    /// period. The counter will either wrap to 0 or reverse
    /// setting of `phase_correct`.
    pub top: u16,
}
```



# ADC

Analog to Digital Converter



# Bibliography

for this section

**Raspberry Pi Ltd, *RP2040 Datasheet***  
-----

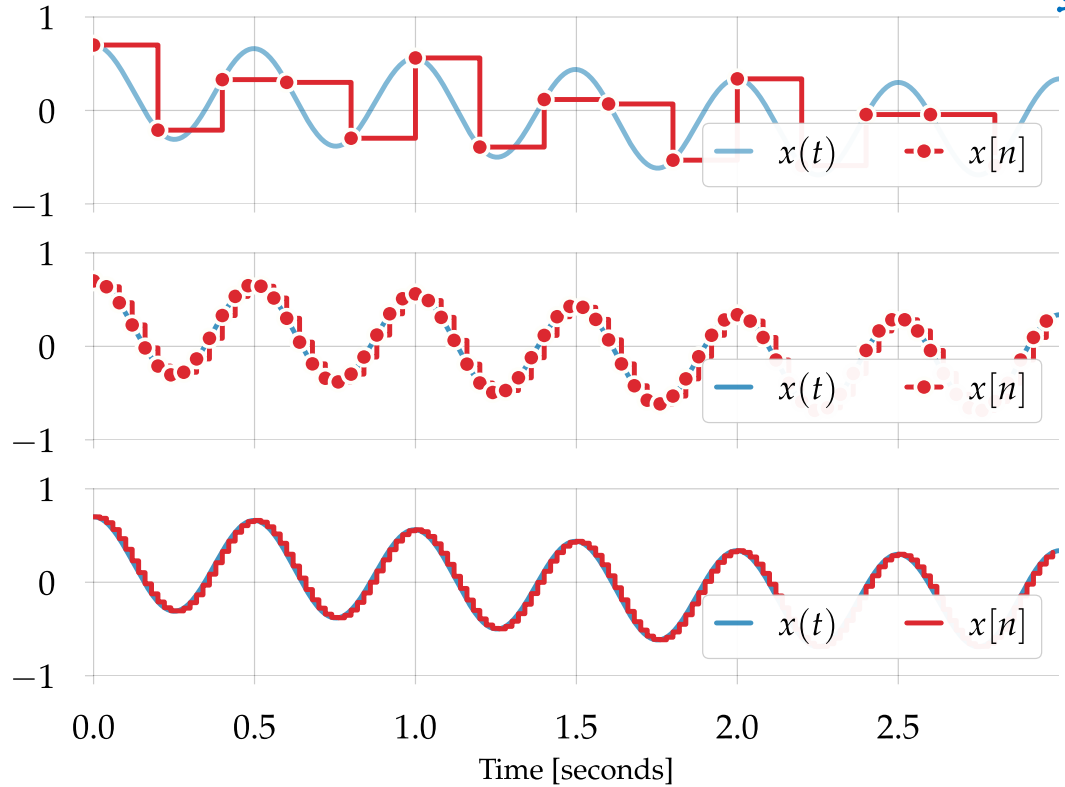
- Chapter 12 - *Peripherals*
  - Section 12.4 - *ADC and Temperature Sensor*
    - Subchapter 12.4.2
    - Subchapter 12.4.3
    - Subchapter 12.4.6



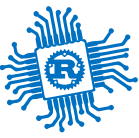
# ADC

sampling an analog signal to an array of values

|                      |      |                                                  |
|----------------------|------|--------------------------------------------------|
| <i>sampling rate</i> | Hz   | the frequency at which a new sample is read      |
| <i>resolution</i>    | bits | the number of bits used to store a sampled value |



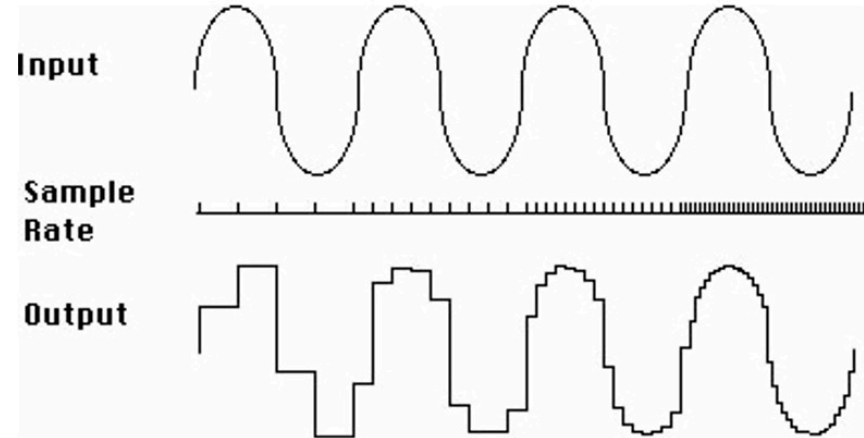
Lower sample rates yield the *aliasing effect*.



# Nyquist–Shannon Sampling Theorem

$$sampling_f > 2 \times max_f$$

The **sampling frequency** has to be at least **two times higher** than the **maximum frequency** of the signal to avoid frequency aliasing<sup>[1]</sup>.



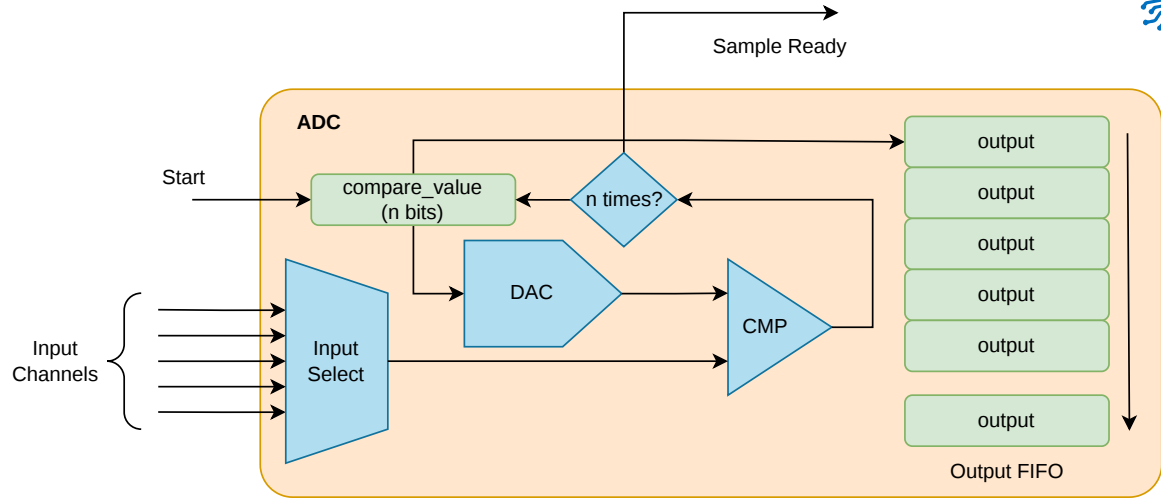
- 
1. Aliasing is the overlapping of frequency components. This overlap results in distortion or artifacts when the signal is reconstructed from samples which causes the **reconstructed signal to differ from the original** continuous signal. ↩



# Sampling

how the ADC works

- assumes  $\text{bit}_{n-1}$  of `compare_value` is 1
- compares the input signal with a generated analog signal from `compare_value`
  - if input is lower,  $\text{bit}_{n-1}$  is 0
  - if input is higher,  $\text{bit}_{n-1}$  is 1
- repeats for  $\text{bit}_{n-2}$ ,  $\text{bit}_{n-3} \dots \text{bit}_0$



There are different types of ADCs depending on the architecture. The most common used is SAR (Successive Approximation Register) ADC, also integrated in RP2350.

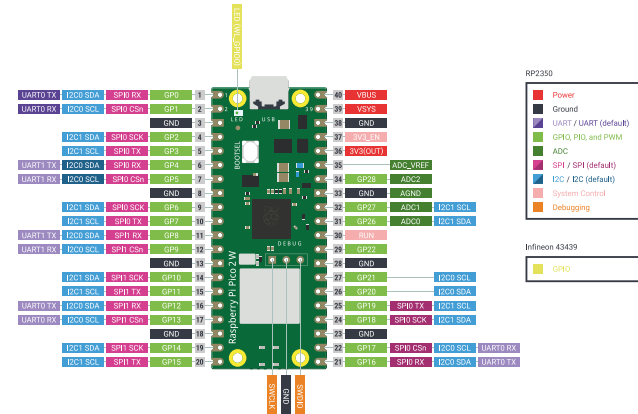
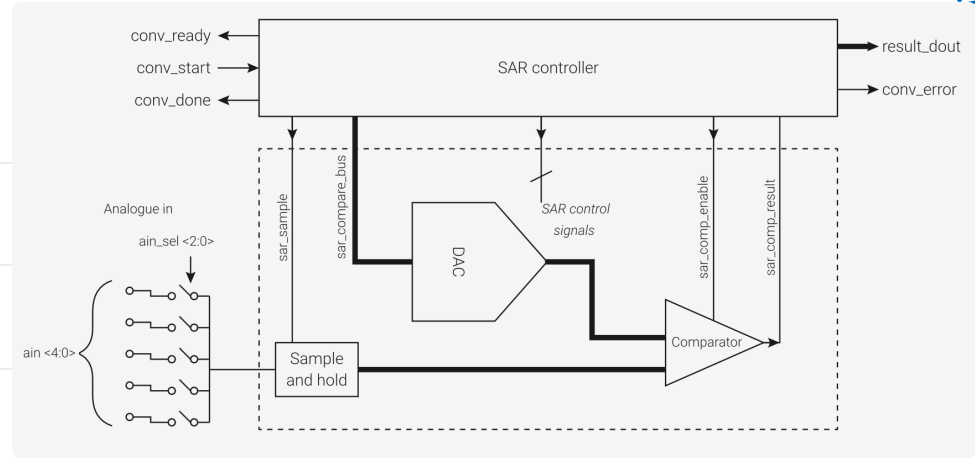


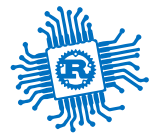
# RP2350's ADC

|                        |                       |
|------------------------|-----------------------|
| <i>channels</i>        | 4 or 8 <sup>[1]</sup> |
| <i>sampling rate</i>   | 500 kHz               |
| <i>resolution</i>      | 12 bits               |
| <i>V<sub>max</sub></i> | 3.3 V                 |

- requires a 48 MHz clock signal
- channel 4 or 8<sup>[1:1]</sup> is connected to the internal temperature sensor

$$t = 27 - \frac{(V_{input\_4} - 0.706)}{0.001721} [^{\circ}C]_{SI}$$

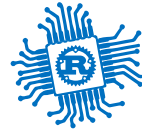




# ADC

in Embassy

```
1  use embassy_rp::adc::{Adc, Channel, Config, InterruptHandler};
2
3  bind_interrupts!(struct Irqs {
4      ADC_IRQ_FIFO => InterruptHandler;
5  });
6
7  let p = embassy_rp::init(Default::default());
8  let mut adc = Adc::new(p.ADC, Irqs, Config::default());
9
10 let mut p26 = Channel::new_pin(p.PIN_26, Pull::None);
11
12 loop {
13     let level = adc.read(&mut p26).await.unwrap();
14     info!("Pin 26 ADC: {}", level);
15     let voltage = 3300 * level / 4095;
16     info!("Pin 26 voltage: {:.{}V", voltage / 1000, voltage % 1000);
17     Timer::after_secs(1).await;
18 }
```



# Conclusion

we talked about

- Exceptions and Interrupts
- Counters
- SysTick
- Timers and Alarms
- PWM
- Analog and Digital
- ADC