



PWM and ADC

Lecture 4



PWM and ADC

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



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*



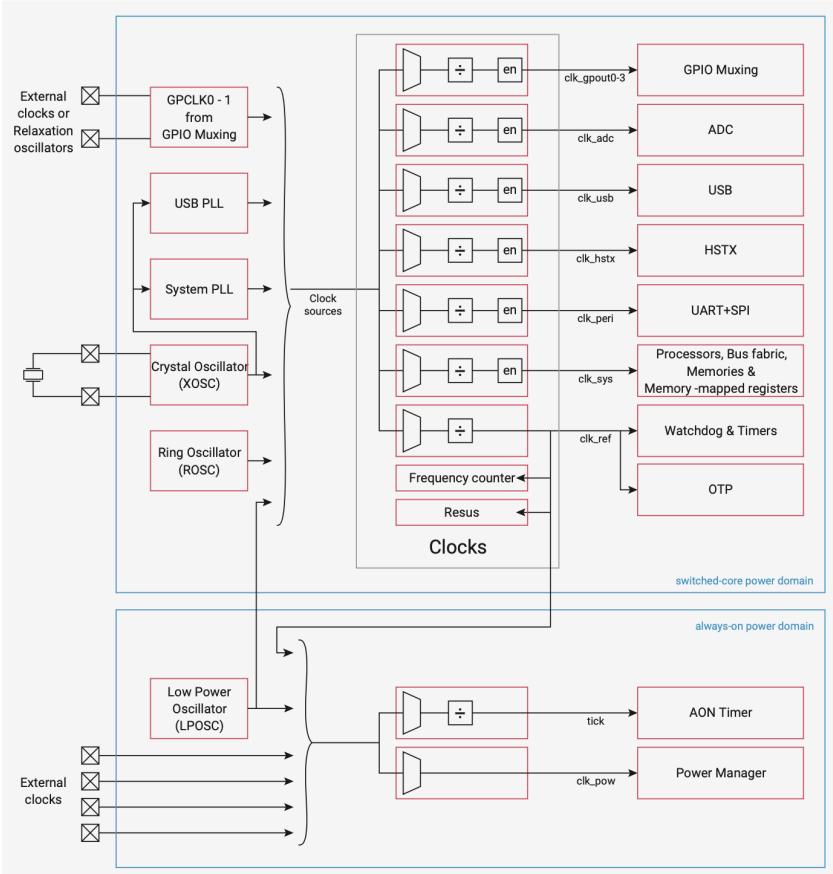
Clocks

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

| Source | Usage |
|-----------------------------------|---|
| <i>external crystal</i> (XOSC) | a stable frequency is required, for instance when using USB |
| <i>internal ring</i> (ROSC) | 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());
```

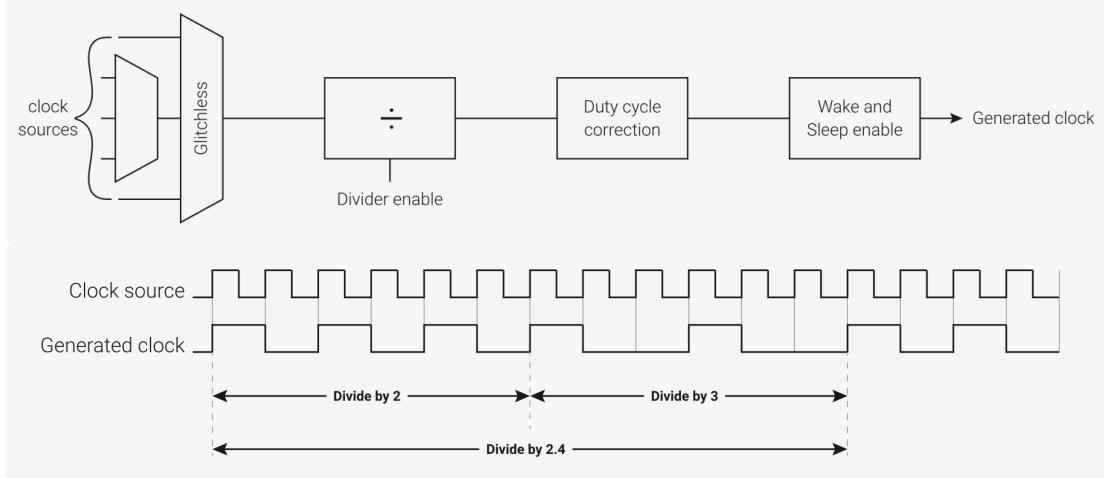




Frequency divider

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`

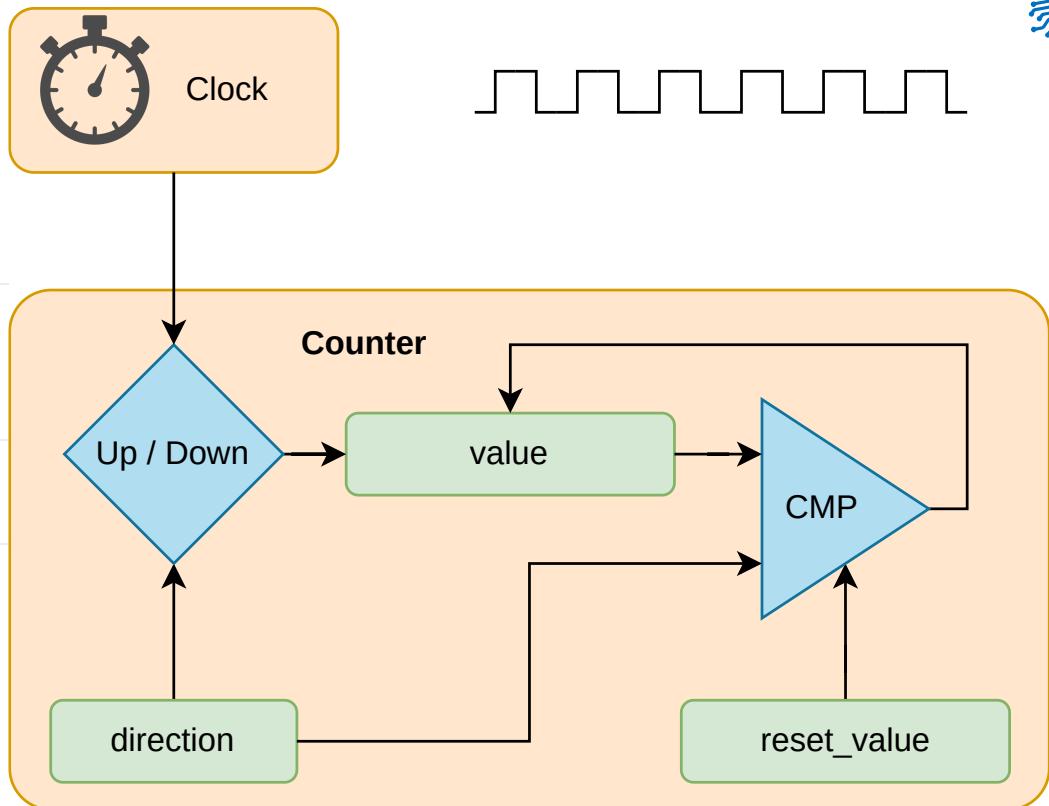
the current value of the counter

`direction`

set to count UP or DOWN

`reset`

UP: the value at which the counter resets to `0`
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 | <code>SYST_CSR</code> | SysTick Control and Status Register |
| 0xe014 | <code>SYST_RVR</code> | SysTick Reload Value Register |
| 0xe018 | <code>SYST_CVR</code> | SysTick Current Value Register |
| 0xe01c | <code>SYST_CALIB</code> | SysTick Calibration Value Register |

- decrements the value of `SYST_CVR` every μs
- 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. Selects the SysTick timer clock source: 0 = External reference clock. 1 = Processor clock. | RW | 0x0 |
| 1 | TICKINT | Enables SysTick exception request: 0 = Counting down to zero does not assert the SysTick exception request. 1 = Counting down to zero asserts the SysTick exception request. | RW | 0x0 |
| 0 | ENABLE | Enable SysTick counter: 0 = Counter disabled. 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 | <code>SYST_CSR</code> | SysTick Control and Status Register |
| 0xe014 | <code>SYST_RVR</code> | SysTick Reload Value Register |
| 0xe018 | <code>SYST_CVR</code> | SysTick Current Value Register |
| 0xe01c | <code>SYST_CALIB</code> | 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. Selects the SysTick timer clock source: 0 = External reference clock. 1 = Processor clock. | RW | 0x0 |
| 1 | TICKINT | Enables SysTick exception request: 0 = Counting down to zero does not assert the SysTick exception request. 1 = Counting down to zero asserts the SysTick exception request. | RW | 0x0 |
| 0 | ENABLE | Enable SysTick counter: 0 = Counter disabled. 1 = Counter enabled. | RW | 0x0 |

Register SysTick handler

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

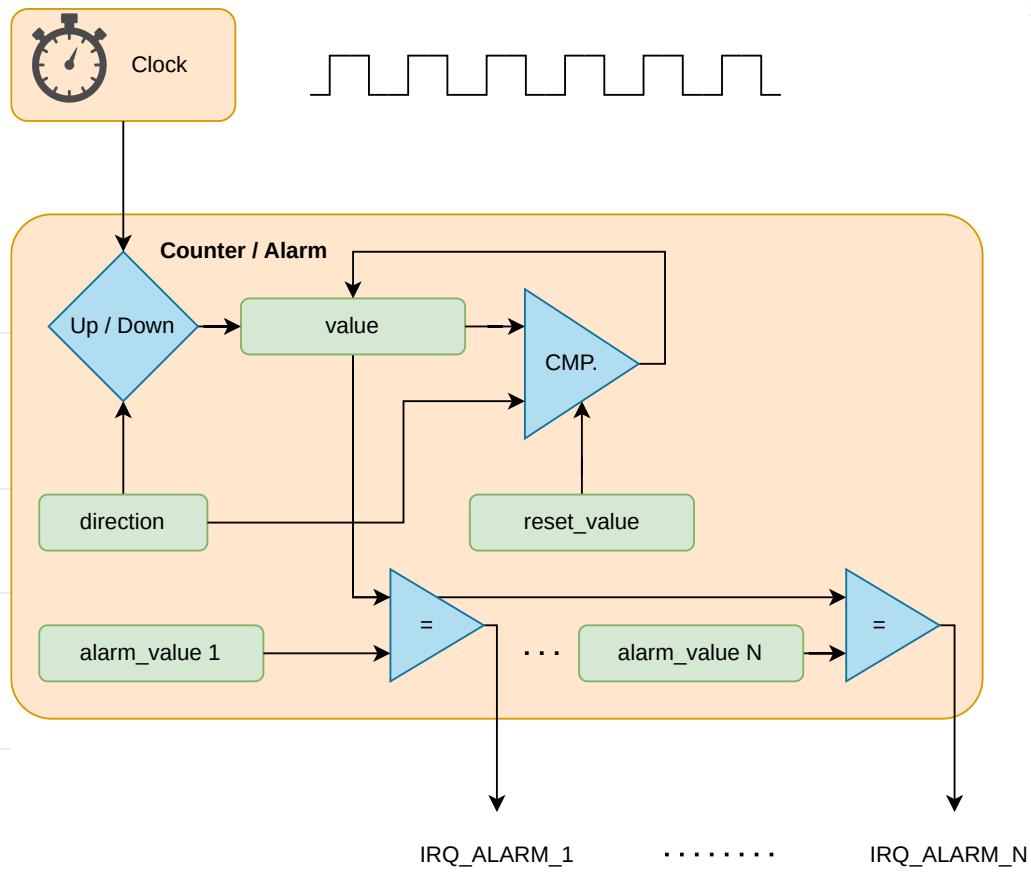


Alarm

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 DOWN: value after 0 |
| <code>alarm_x</code> | when <code>value == alarm_x</code> , triggers an interrupt, <code>x</code> in <code>1 ... n</code> |

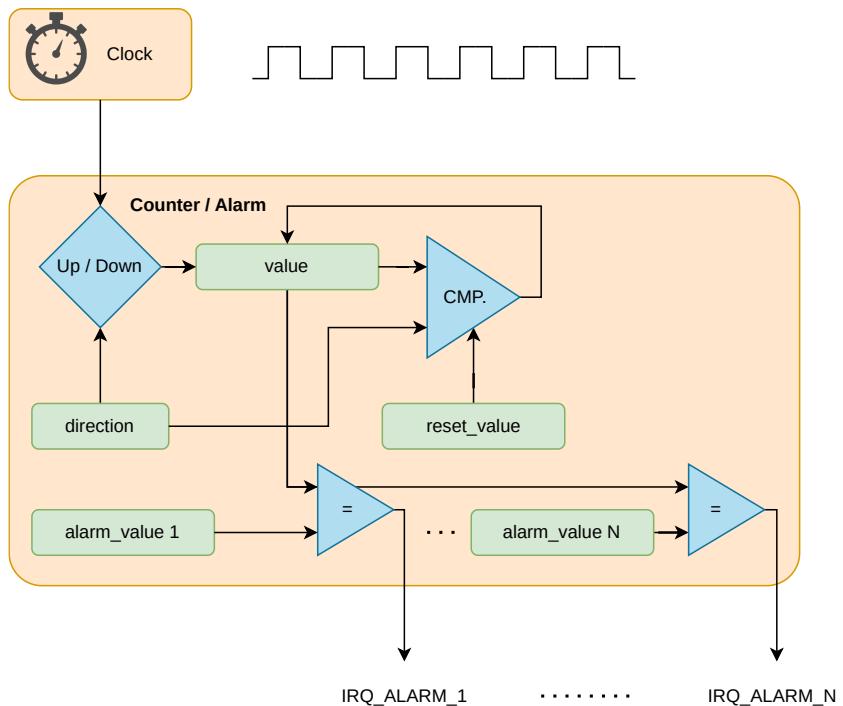




RP2350's Timers

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 μs
- 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 μs 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 `TIMELR` latches the value in `TIMEHR` (stops being updated) until `TIMEHR` 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 always write timelw before timehw |
| 0x04 | <code>TIMElw</code> | Write to bits 31:0 of time writes do not get copied to time until timehw is written |
| 0x08 | <code>TIMEHR</code> | Read from bits 63:32 of time always read timerl before timehr |
| 0x0c | <code>TIMElr</code> | Read from bits 31:0 of time |
| 0x10 | <code>ALARM0</code> | Arm alarm 0, and configure the time it will fire. Once armed, the alarm fires when <code>TIMER_ALARM0 == TIMELR</code> . 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. Once armed, the alarm fires when <code>TIMER_ALARM1 == TIMELR</code> . 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. Once armed, the alarm fires when <code>TIMER_ALARM2 == TIMELR</code> . 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. Once armed, the alarm fires when <code>TIMER_ALARM3 == TIMELR</code> . 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. A write to the corresponding ALARMx register arms the alarm. 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 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 | TIMEHW | Write to bits 63:32 of time always write timelw before timehw |
| 0x04 | TIMELW | Write to bits 31:0 of time writes do not get copied to time until timehw is written |
| 0x08 | TIMEHR | Read from bits 63:32 of time always read timelr before timehr |
| 0x0c | TIMELR | Read from bits 31:0 of time |
| 0x10 | ALARM0 | Arm alarm 0, and configure the time it will fire. Once armed, the alarm fires when <code>TIMER_ALARM0 == TIMELR</code> . The alarm will disarm itself once it fires, and can be disarmed early using the ARMED status register. |
| 0x14 | ALARM1 | Arm alarm 1, and configure the time it will fire. Once armed, the alarm fires when <code>TIMER_ALARM1 == TIMELR</code> . The alarm will disarm itself once it fires, and can be disarmed early using the ARMED status register. |
| 0x18 | ALARM2 | Arm alarm 2, and configure the time it will fire. Once armed, the alarm fires when <code>TIMER_ALARM2 == TIMELR</code> . The alarm will disarm itself once it fires, and can be disarmed early using the ARMED status register. |
| 0x1c | ALARM3 | Arm alarm 3, and configure the time it will fire. Once armed, the alarm fires when <code>TIMER_ALARM3 == TIMELR</code> . The alarm will disarm itself once it fires, and can be disarmed early using the ARMED status register. |
| Offset | Name | Info |
| 0x38 | SOURCE | 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 <code>clk_sys</code> cycles instead. |
| 0x3c | INTR | Raw Interrupts |
| 0x40 | INTE | Interrupt Enable |
| 0x44 | INTF | Interrupt Force |
| 0x48 | INTS | Interrupt status after masking & forcing |



Signals

Digital Signals - Recap



Signals

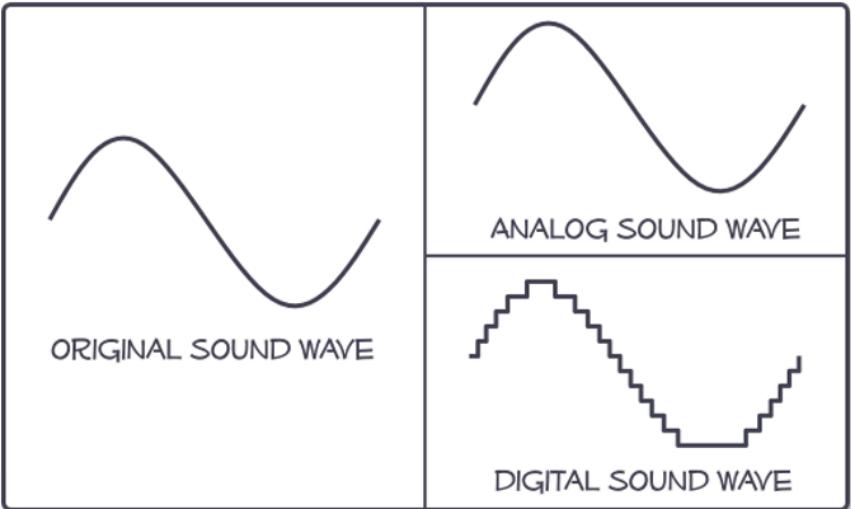
Analog vs Digital

- *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

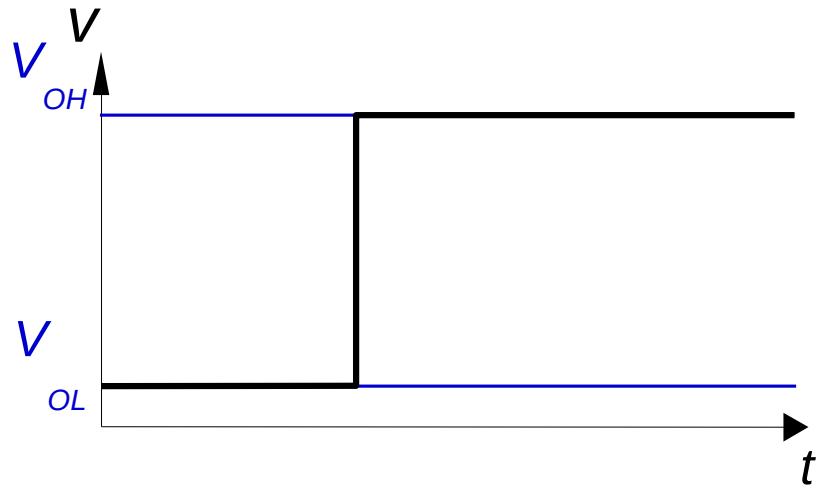
for PCB level / between integrated circuits on the same board / inside the same chip - things are a "a little simpler" - as detailed in the following



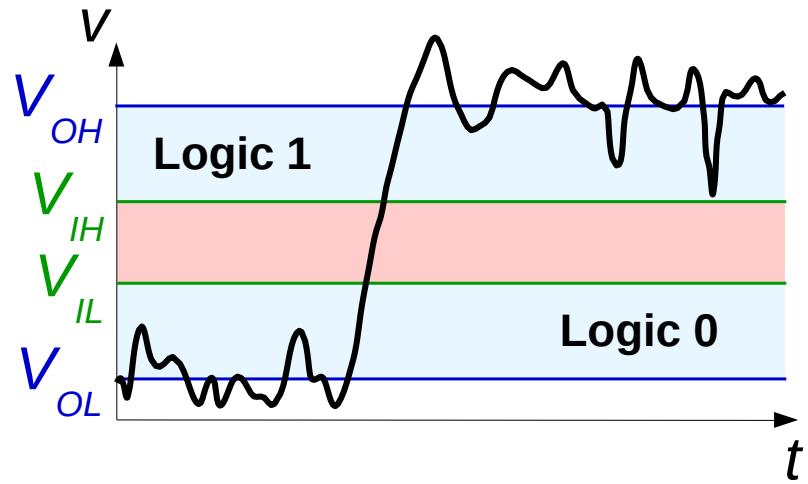


Why use digital in computing?

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



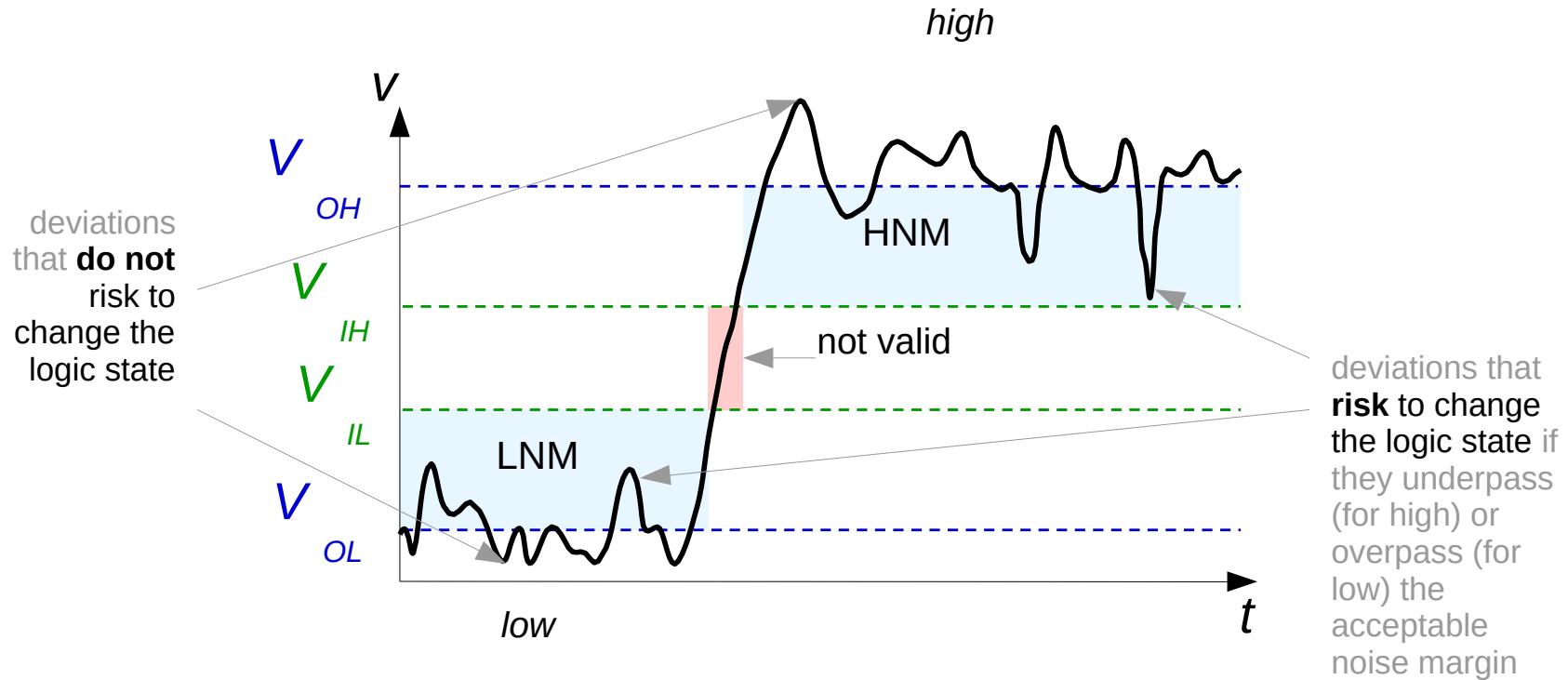
Signal that what we actually generate



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

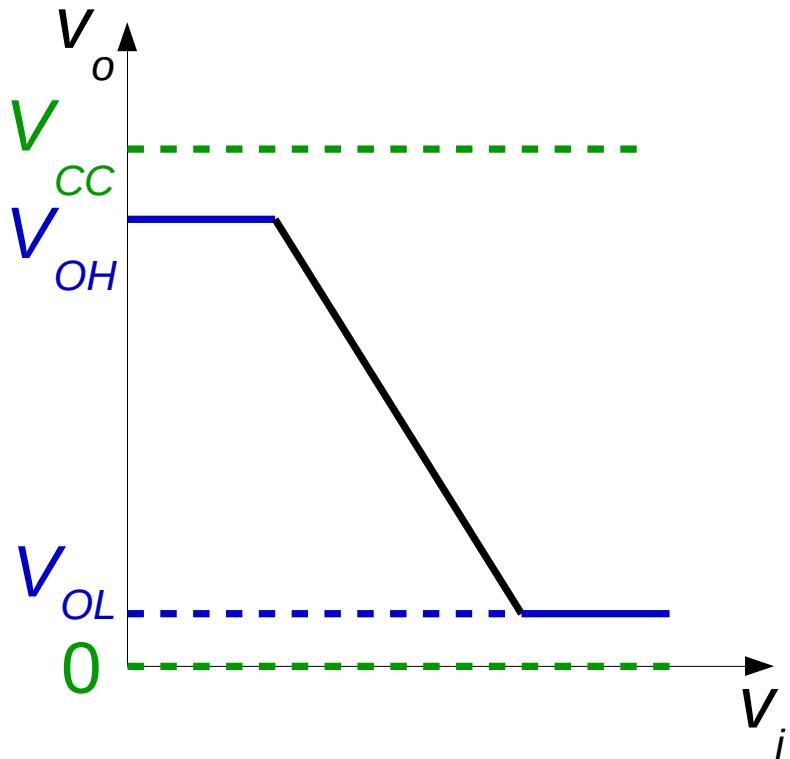




Why is the output not ideal?

The two corresponding voltage output levels are affected by:

- power supply voltage
- output current
- temperature
- variations in the manufacturing process





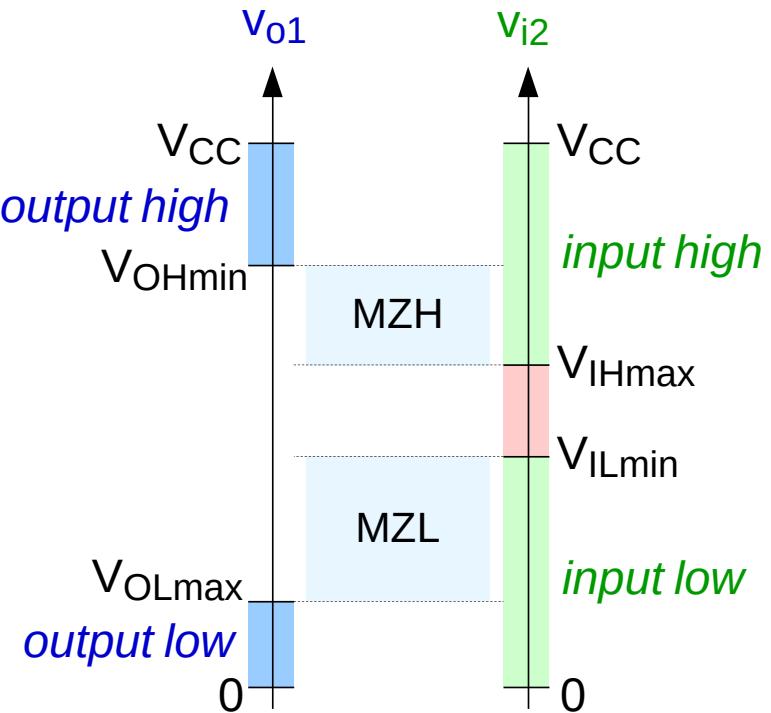
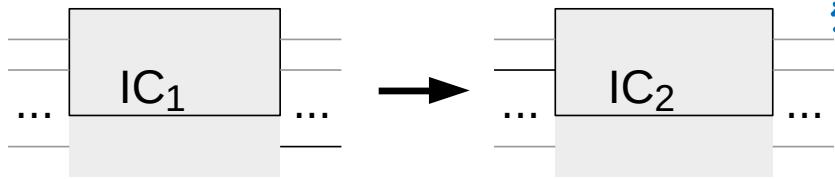
ICs same voltage

Usually will work as is

- usually, they will be compatible
- conditions:

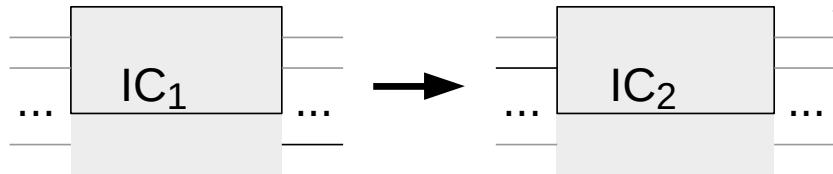
$$V_{OH_transmitter} > V_{IH_receiver}$$

$$V_{OL_transmitter} < V_{IL_receiver}$$





VCC1 > VCC2



Might work, might produce magic smoke

$$V_{OH_transmitter} > VCC_{receiver}$$

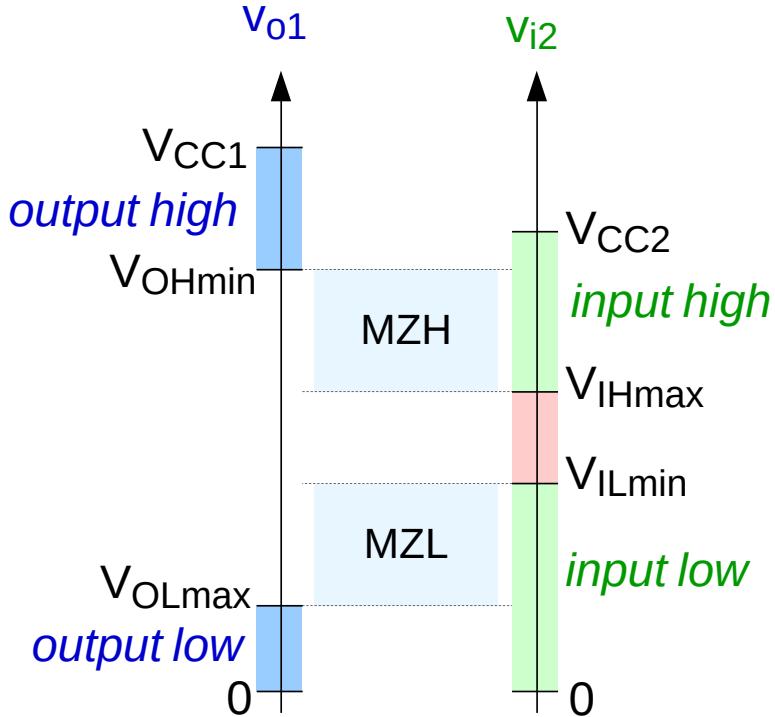
PROBLEM

Solutions:

- level shifter
- resistor divider / voltage limiter

Examples:

- Bi-Directional Level Shifter with 4 Channels
- Level Shifter Multi-Channel
- 8 Channels Level Shifter





VCC1 < VCC2

Might work

$$V_{CC_transmitter} \lesssim V_{IH_receiver}$$

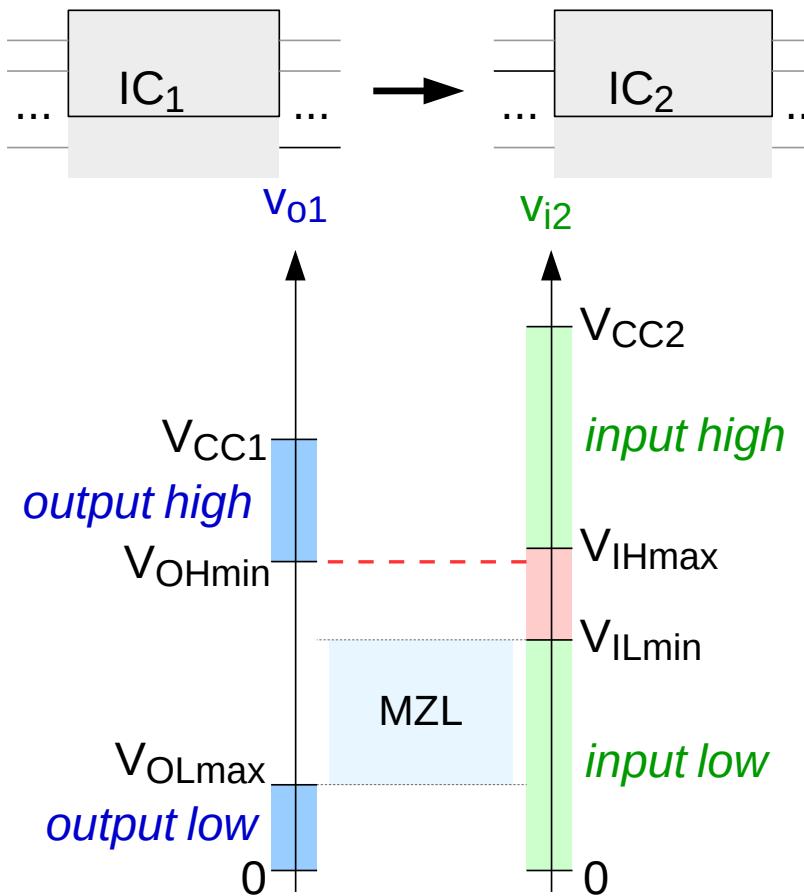
Might work in an intermittent mode - hard to debug!

Solutions:

- level shifter
- resistor divider / voltage limiter

Examples:

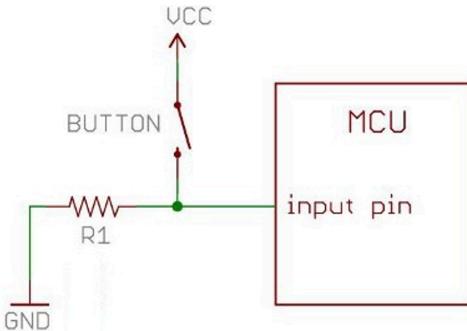
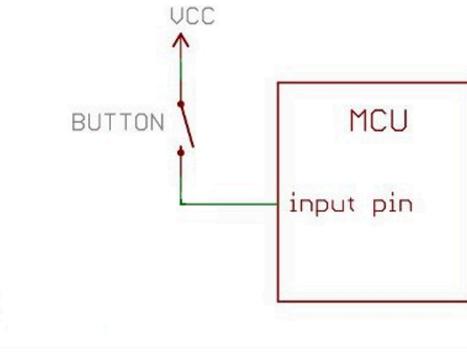
- Bi-Directional Level Shifter with 4 Channels
- Level Shifter Multi-Channel
- 8 Channels Level Shifter





Why Pull-Down R

- Without pull-down – when the button is not pressed, it leaves the input pin floating.
- The second design ensures that the voltage level has a well-defined state, regardless of the button's state.
- R1 is called a "pull-down" resistor.



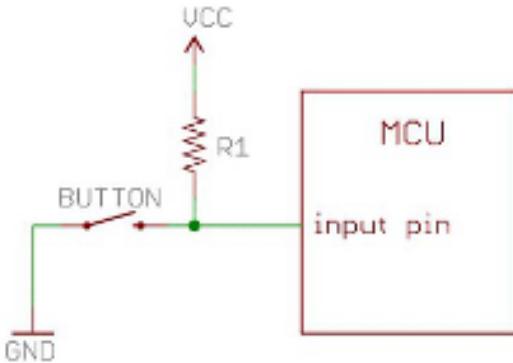


Why Pull-Up R

- Same reasoning
- R1 is called a "pull-up" resistor.

##Obs:

- most microcontrollers have at least a pull-up resistor incorporated on GPIOs - that can be activated in software
- some have both pull-up and pull-down
- typically, these are sized for a 50 - 10 nA current consumption





Notes on output pins

- most microcontrollers have a limit of around 10mA per output PIN
- ! do not connect an LED without a resistor in series (to limit the current)
- ! do not connect a motor / any type of inductive load

Solutions:

- use a transistor
- use an IC with incorporated Darlintonts (eg: ULN2003)



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*



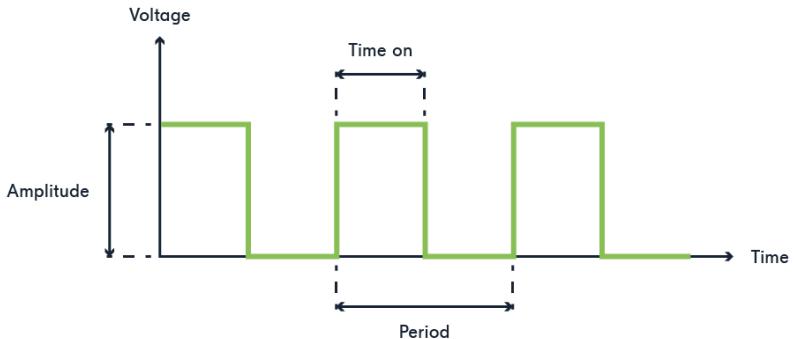
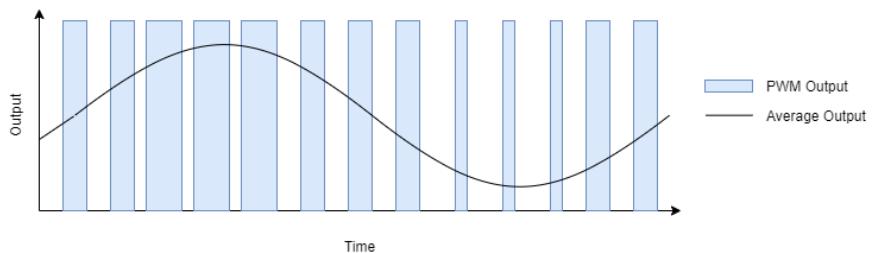
PWM

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



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

$$\text{duty_cycle} = \frac{\text{time_on}}{\text{period}} \%$$

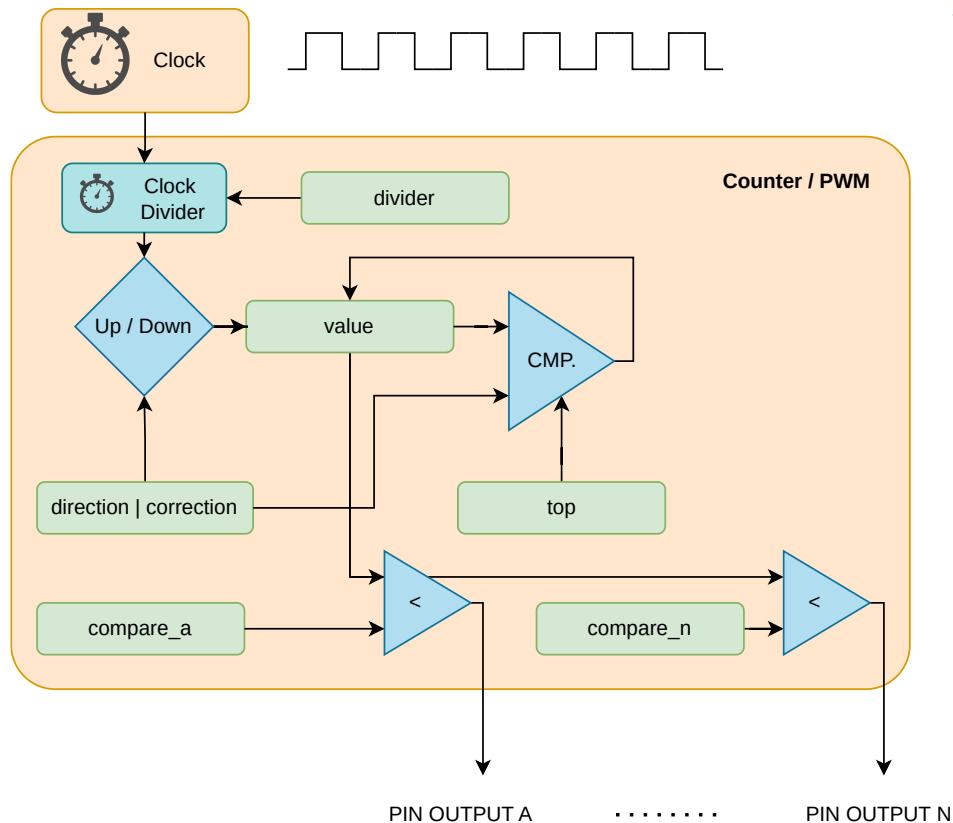


PWM

generic device

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

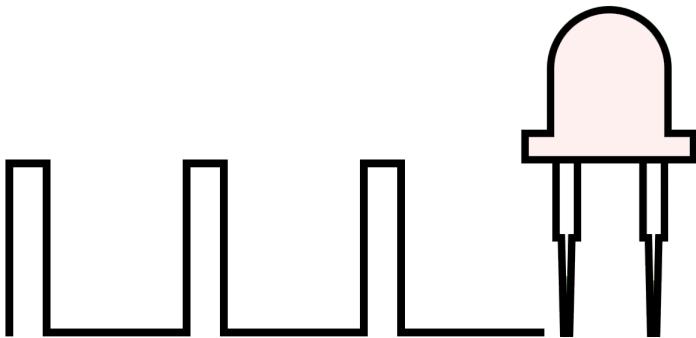
$$pin_{a,b} = \begin{cases} 0 & compare_{a,b} \geq value \\ 1 & compare_{a,b} < value \end{cases}$$



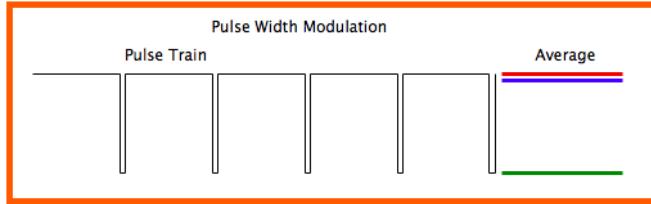


Usage examples

- dimming an LED



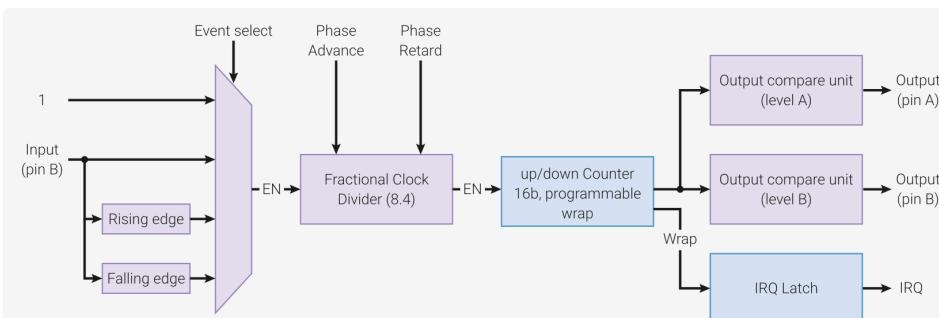
- controlling motors
 - controlling the angle of a stepper motor
 - controlling the RPM of a motor





RP2350's PWM

- generates square signals
 - counts the pulse width of input signals
 - 8 or 12^[1] 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`)

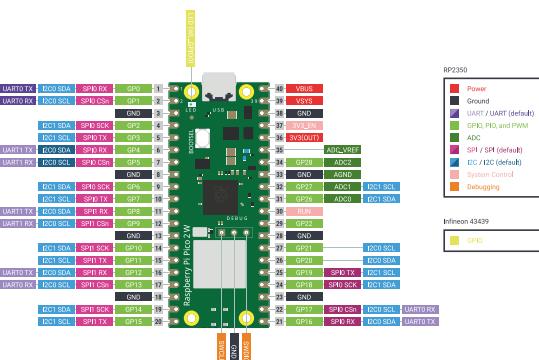


| GPIO | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|-------------|----|----|----|----|-----|-----|-----|-----|----|----|----|----|-----|-----|-----|-----|
| PWM Channel | 0A | 0B | 1A | 1B | 2A | 2B | 3A | 3B | 4A | 4B | 5A | 5B | 6A | 6B | 7A | 7B |
| GPIO | 16 | 17 | 18 | 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 | CH0_CSR | Control and status register |
| 0x004 | CH0_DIV | 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 | CH0_CTR | Direct access to the PWM counter |
| 0x00c | CH0_CC | Counter compare values |
| 0x010 | CH0_TOP | Counter wrap value |

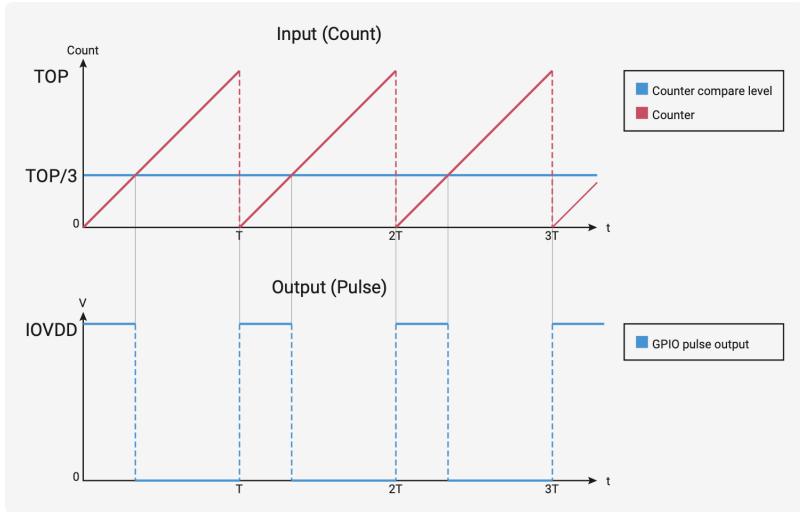


1. Depends on the RP2350 package ↵

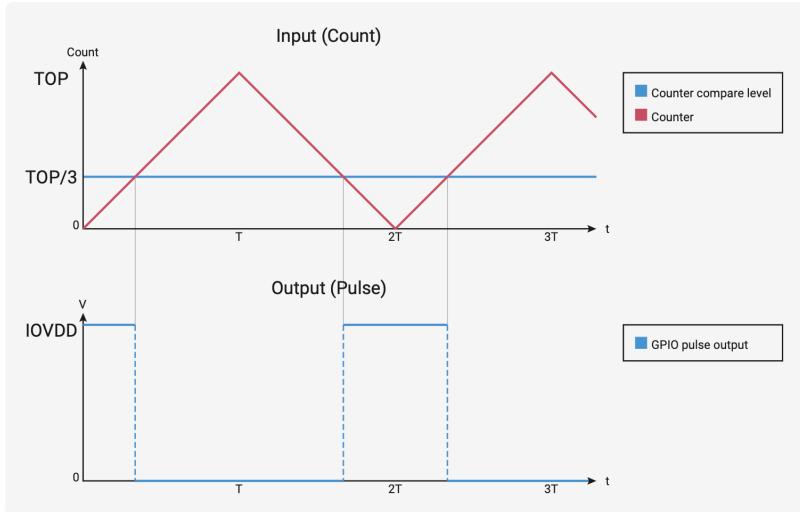


RP2350's PWM Modes

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

sampling rate

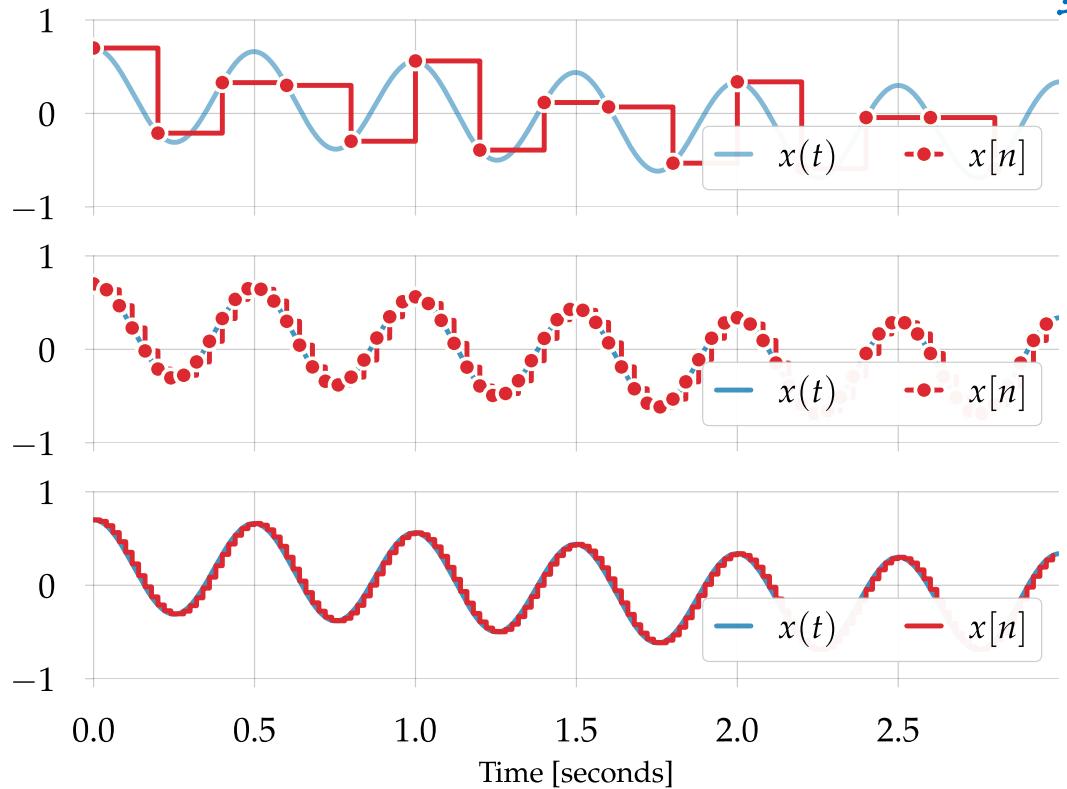
Hz

the frequency at which a new sample is read

resolution

bits

the number of bits used to store a sampled value



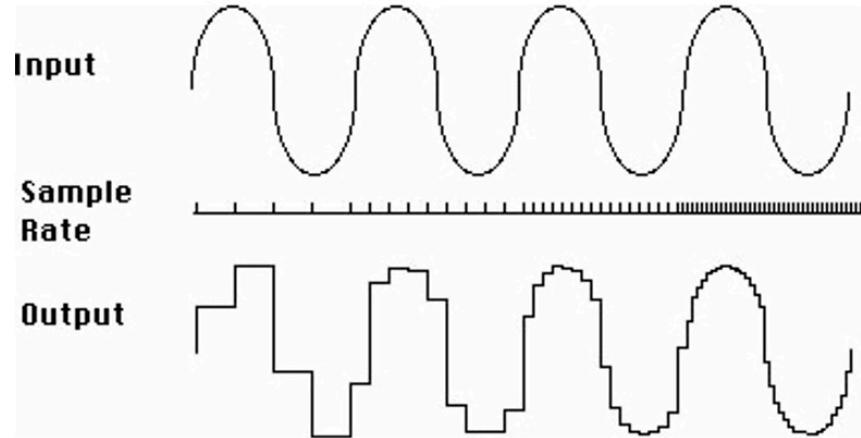
Lower sample rates yield the *aliasing effect*.



Nyquist–Shannon Sampling Theorem

$$\text{sampling}_f > 2 \times \text{max}_f$$

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



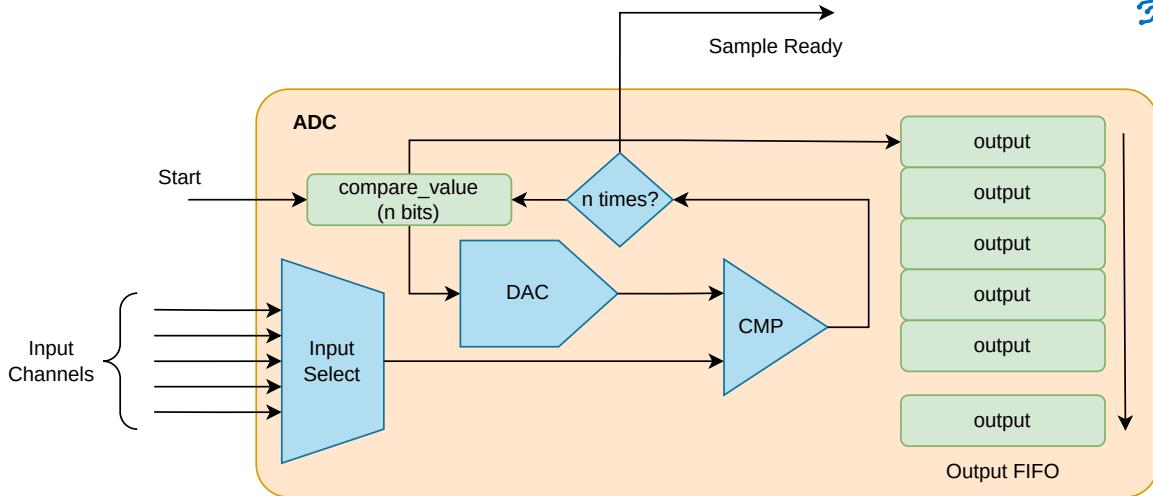
-
1. Aliasing is the overlapping of frequency components. This overlap results in distortion or artifacts when the signal is reconstructed from samples which causes the **reconstructed signal to differ from the original** continuous signal. ↵



Sampling

how the ADC works

- assumes bit_{n-1} of compare_value is 1
- compares the input signal with a generated analog signal from compare_value
 - if input is lower, bit_{n-1} is 0
 - if input if higher, bit_{n-1} is 1
- repeats for bit_{n-2}, bit_{n-3} ... bit₀



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

channels 4 or 8^[1]

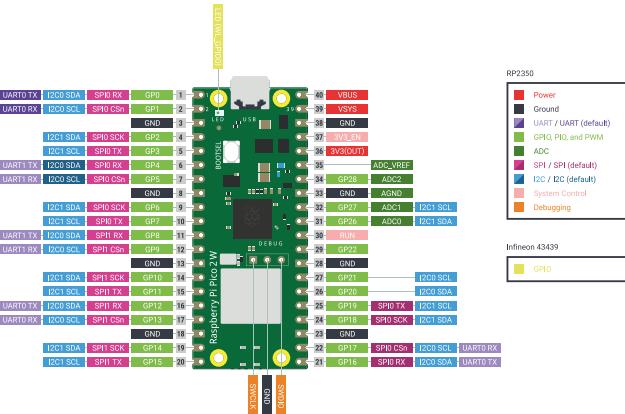
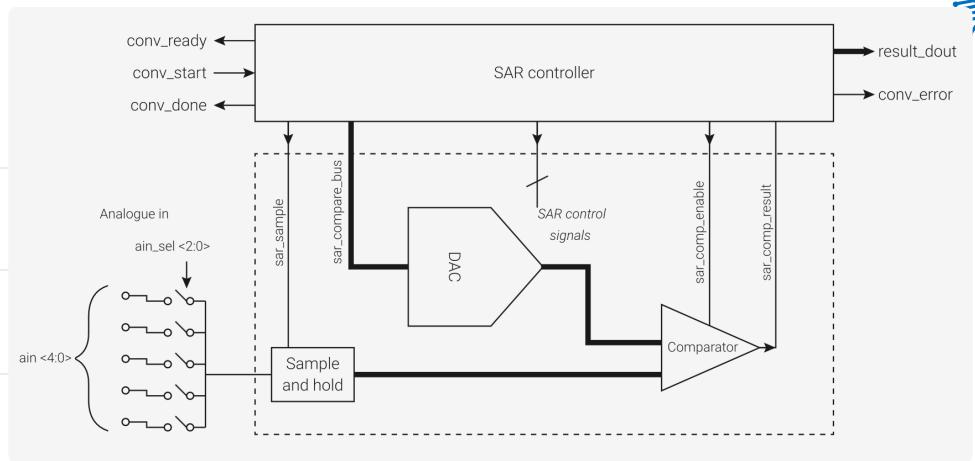
sampling rate 500 kHz

resolution 12 bits

V_{max} 3.3 V

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

$$t = 27 - \frac{(V_{input_4} - 0.706)}{0.001721} [\text{ }^{\circ}\text{C}]_{SI}$$





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

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