

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, RP2040 Datasheet

- Chapter 2 System Description
 - Chapter 2.15 *Clocks*
 - Subchapter 2.15.1
 - Subchapter 2.15.2
- Chapter 4 *Peripherals*
 - Chapter 4.6 *Timer*







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

Source	Usage
external crystal (XOSC)	a stable frequency is required, for instance when using USB
internal ring (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.

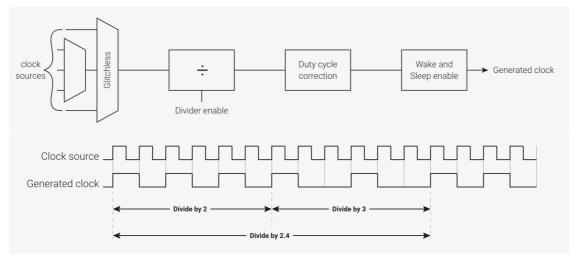
```
1 let p = embassy_rp::init(Default::default());
```

```
GPIO Muxing
                                                                                       clk_gpout0-3
External clocks
                      GPIO Muxing
                                                                                         clk_adc
                                                                                                               USB
                                                                                         clk_usb
                        USB PLL
                                                                                                                RTC
                                                                                          clk_rtc
                      System PLL
                                             Clock
                                                                                                             UART+SPI
                                                                                         clk_peri
                                                                                                       Processors, Bus fabric,
                     Crystal Oscillator
                                                                                                            Memories &
                                                                                         clk_sys
                                                                                                         Watchdog & Timers
                                                                                          clk_ref
                     Ring Oscillator
                         (ROSC)
                                                             Frequency counter -
                                                                  Resus
                                                                Clocks
```

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 desires the rate





Counter

increments a register at every clock cycle

Clock	
Counter Up / Down	value
direction	reset_value

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	SYST_CSR	SysTick Control and Status Register
0xe014	SYST_RVR	SysTick Reload Value Register
0xe018	SYST_CVR	SysTick Current Value Register
0xe01c	SYST_CALIB	SysTick Calibration Value Register

- decrements the value of SYST_CVR every μs
- when SYST_CVR becomes 0 :
 - triggers the SysTick the 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	Туре	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 to asserts the SysTick exception request.	RW	0x0
0	ENABLE	Enable SysTick counter: 0 = Counter disabled. 1 = Counter enabled.	RW	0x0

$$f = rac{1}{SVST \ \ BVR} * 1,000,000 [Hz]_{SI}$$

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	SYST_CSR	SysTick Control and Status Register
0xe014	SYST_RVR	SysTick Reload Value Register
0xe018	SYST_CVR	SysTick Current Value Register
0xe01c	SYST_CALIB	SysTick Calibration Value Register

```
const SYST_RVR: *mut u32 = 0xe000_e014 as *mut u32;
const SYST_CVR: *mut u32 = 0xe000_e018 as *mut u32;
const SYST_CSR: *mut u32 = 0xe000_e010 as *mut u32;

// fire systick every 5 seconds
let interval: u32 = 5_000_000;
unsafe {
    write_volatile(SYST_RVR, interval);
    write_volatile(SYST_CVR, 0);
// set fields `ENABLE` and `TICKINT`
urite_volatile(SYST_CSR, 0b11);
}
```

SYST_CSR register



Bits	Name	Description	Туре	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 to asserts the SysTick exception request.	RW	0x0
0	ENABLE	Enable SysTick counter: 0 = Counter disabled. 1 = Counter enabled.	RW	0x0

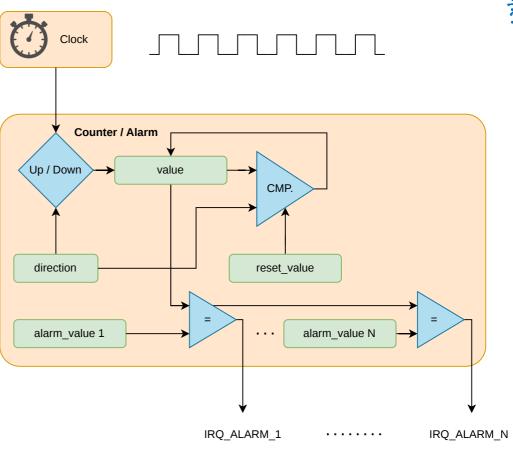
Register SysTick handler

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



counter that triggers interrupts after a time interval

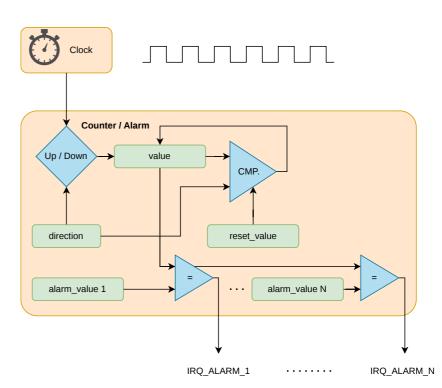
Registers	Description
value	the current value of the counter
direction	set to count UP or DOWN
reset	UP: max value before 0 DOWN: value after 0
alarm_x	<pre>when value == alarm_x , triggers an interrupt, x in 1 n</pre>



RP2040's Timer



- stores a 64 bit number (reset is 2^{64-1})
- starts with 0 at (the peripheral's) reset
- increments the number every μ s
- in practice fully monotonic (cannot over flow)
- allows 4 alarms that trigger interrupts
 - TIMER_IRQ_0
 - TIMER_IRQ_1
 - TIMER_IRQ_2
 - TIMER_IRQ_3
- alarm_0 ... alarm_3 registers are only 32 bits



RP2040's Timer

read the number of elapsed us since reset

The Timer registers start at a base address of 0x40054000 (defined as TIMER_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

Reading the time elapsed since restart

```
const TIMERLR: *const u32 = 0x4005_400c;
const TIMERHR: *const u32 = 0x4005_4008;

let time: u64 = unsafe {
    let low = read_volatile(TIMERLR);
    let high = read_volatile(TIMERHR);
    high as u64 << 32 | low
}</pre>
```

The **reading order maters** as reading TIMELR latches the value in TIMEHR (stops being updated) until TIMEHR is read. Works only in **single core**.



Offset	Name	Info
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 TIMER_ALARM0 == TIMELR. 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 TIMER_ALARM1 == TIMELR. 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 TIMER_ALARM2 == TIMELR. 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 TIMER_ALARM3 == TIMELR. The alarm will disarm itself once it fires, and can be disarmed early using the ARMED status register.
0x20	ARMED	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	TIMERAWH	Raw read from bits 63:32 of time (no side effects)
0x28	TIMERAWL	Raw read from bits 31:0 of time (no side effects)
0x2c	DBGPAUSE	Set bits high to enable pause when the corresponding debug ports are active
0x30	PAUSE	Set high to pause the timer
0x34	INTR	Raw Interrupts
0x38	INTE	Interrupt Enable
0x3c	INTF	Interrupt Force
0x40	INTS	Interrupt status after masking & forcing

Alarm

triggering an interrupt at an interval

```
#[interrupt]
     unsafe fn TIMER IRQ 0() { /* alarm fired */ }
     const TIMERLR: *const u32 = 0x4005_400c;
     const ALARM0: *mut u32 = 0x4005 4010;
     // + 0x2000 is bitwise set
     const INTE SET: *mut u32 = 0x4005 4038 + 0x2000;
     // set an alarm after 3 seconds
     let us = 3 0000 0000;
     unsafe {
         let time = read volatile(TIMERLR);
10
         // use `wrapping add` as overflowing may panic
11
12
         write volatile(ALARM0, time.wrapping add(us));
         write volatile(INTE SET, 1 << 0);</pre>
13
14 };
```

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



Offset	Name	Info
0x08	TIMEHR	Read from bits 63:32 of time always read timelr before timehr
0x0c	TIMELR	Read from bits 31:0 of time
0x10	ALARMO	Arm alarm 0, and configure the time it will fire. Once armed, the alarm fires when TIMER_ALARM0 == TIMELR. 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 TIMER_ALARM1 == TIMELR. 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 TIMER_ALARM2 == TIMELR. 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 TIMER_ALARM3 == TIMELR. The alarm will disarm itself once it fires, and can be disarmed early using the ARMED status register.
0x20	ARMED	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	TIMERAWH	Raw read from bits 63:32 of time (no side effects)
0x28	TIMERAWL	Raw read from bits 31:0 of time (no side effects)
0x2c	DBGPAUSE	Set bits high to enable pause when the corresponding debug ports are active
0x30	PAUSE	Set high to pause the timer
0x34	INTR	Raw Interrupts
0x38	INTE	Interrupt Enable
0x3c	INTF	Interrupt Force
0x40	INTS	Interrupt status after masking & forcing



Signals

Analog and Digital

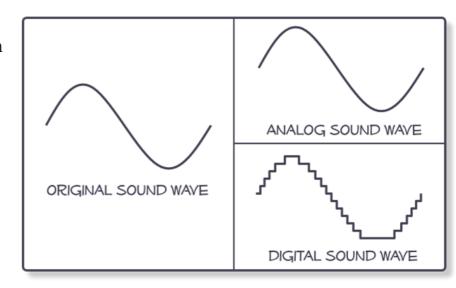


Analog vs Digital

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

Exceptions

- >= 100Mbit Ethernet
- WiFi
- SSD storage



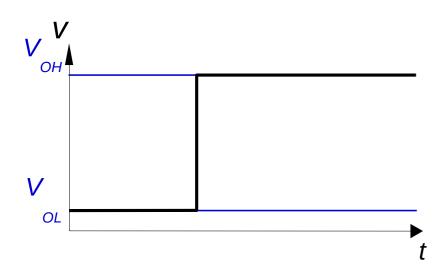


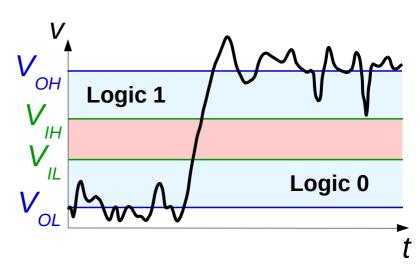


in computing

Signal that we want to generate with an output pin

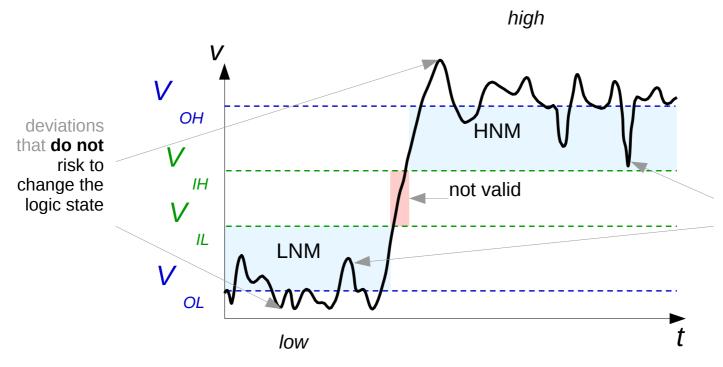
Signal that what we actually generate





Noise Margin





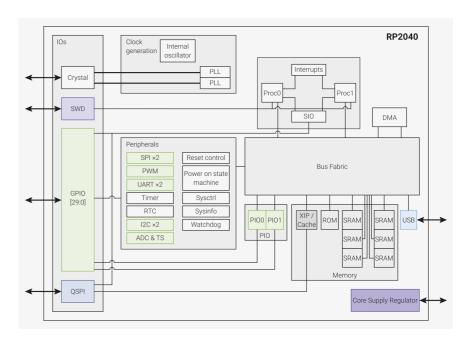
deviations that risk to change the logic state if they underpass (for high) or overpass (for low) the acceptable noise margin



10

using digital signals

- use higher voltage
 - high noise margin
 - higher power consumption ...
- lower noise by using better electronic circuits
- every device samples and regenerates the signal





PWM

Pulse Width Modulation

Bibliography

for this section

- 1. Raspberry Pi Ltd, RP2040 Datasheet
 - Chapter 4 Peripherals
 - Chapter 4.5 *PWM*
- 2. Paul Denisowski, *Understanding PWM*

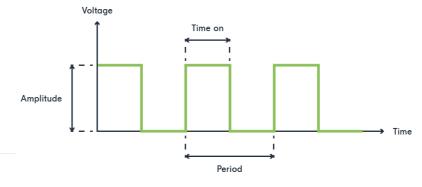


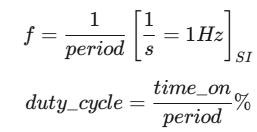
PWM

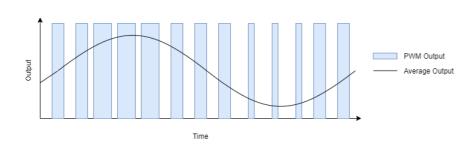


simulates an *analog* signal (using integration)

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





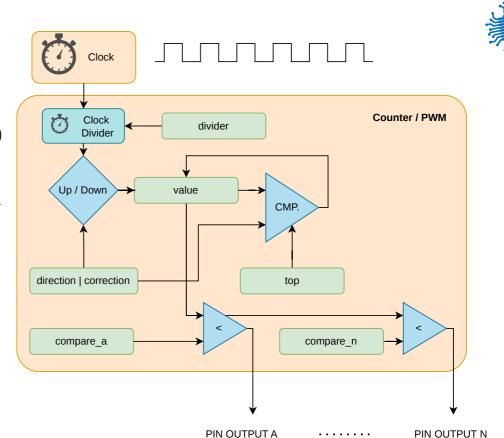


PWM

generic device

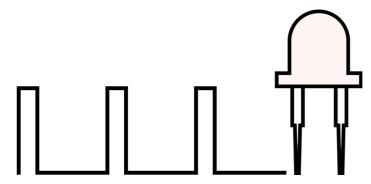
$$f = egin{cases} rac{f_{clock}}{divider imes (top+1)} & correction = 0 \ & & \ rac{f_{clock}}{divider imes 2 imes (top+1)} & correction = 1 \end{cases}$$

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

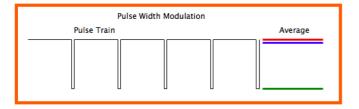


Usage examples

dimming an LED



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





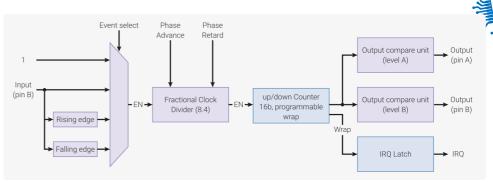
RP2040's PWM

- generates square signals
- counts the pulse with of input signals
- 8 PWM units, each with 2 channels (A and B)
- each PWM channel is connected to a certain pin
- some channels are connected to two pins

Registers

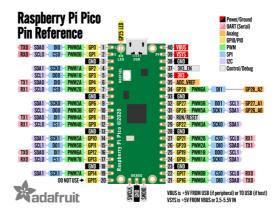
The PWM registers start at a base address of 0x40050000 (defined as PWM_BASE in SDK).

Offset	Name	Info
0x00	CH0_CSR	Control and status register
0x04	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.
0x08	CH0_CTR	Direct access to the PWM counter
0x0c	CH0_CC	Counter compare values
0x10	CH0_TOP	Counter wrap value



All 30 GPIO pins on RP2040 can be used for PWM:

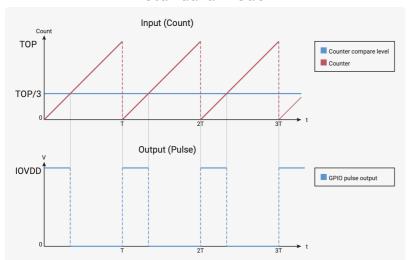
GPIO	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
PWM Channel	0A	0B	1A	1B	2A	2B	3A	3B	4A	4B	5A	5B	6A	6B	7A	7B
GPIO	16	17	18	19	20	21	22	23	24	25	26	27	28	29		
PWM Channel	0A	0B	1A	1B	2A	2B	3A	3B	4A	4B	5A	5B	6A	6B		



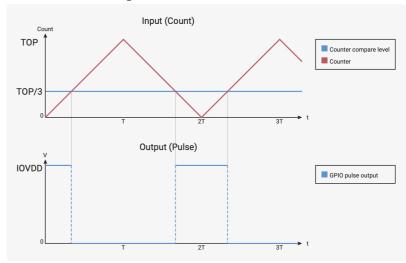
RP2040's PWM Modes



standard mode



phase-correct mode



$$period = (TOP + 1) imes (PH_CORRECT + 1) imes \left(DIV_INT + rac{DIV_FRAC}{16}
ight)[s]_{SI}$$

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

Example

using Embassy

```
use embassy rp::pwm::{Config, Pwm};
      let p = embassy rp::init(Default::default());
     let mut c: Config = Default::default();
     c.top = 0 \times 8000;
     c.compare b = 8;
 8
     let mut pwm = Pwm::new_output_b(
          p.PWM CH4,
10
11
          p.PIN 25,
12
          c.clone()
13
     );
14
15
     loop {
16
          info!("LED duty cycle: {}/32768", c.compare b);
          Timer::after secs(1).await;
17
          c.compare b += 10;
18
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

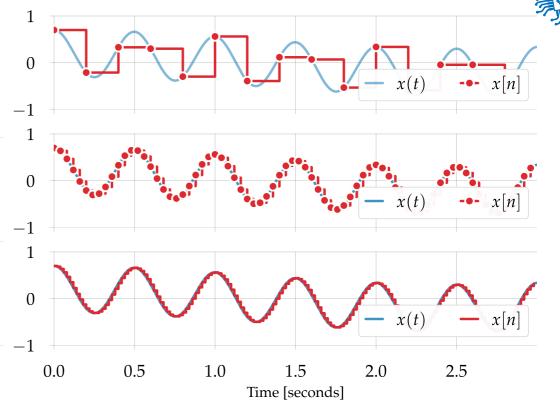
- Chapter 4 *Peripherals*
 - Chapter 4.9 *ADC and Temperature Sensor*
 - Subchapter 4.9.1
 - Subchapter 4.9.2
 - Subchapter 4.9.5



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			



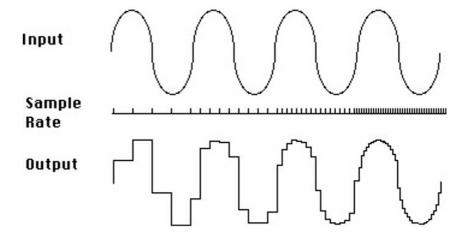
Lower sample rates yield the *aliasing effect*.





$$sampling_f>=2 imes max_f$$

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

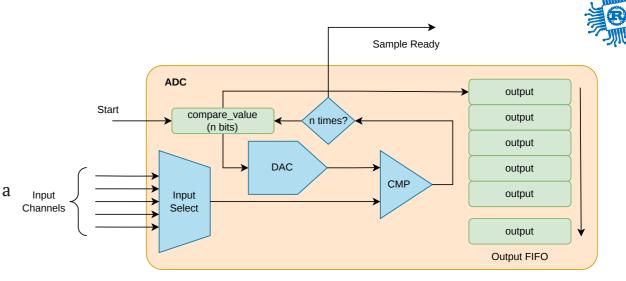


1. Aliasing is the overlapping of frequency components. This overlap results in distortion or artifacts when the signal is reconstructed from samples which causes the **reconstructed signal to differ from the original** continuous signal. ←

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} ... bit_0



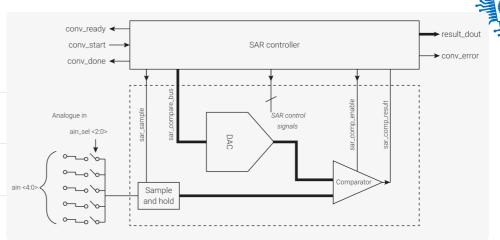
There are different types of ADCs depending on the architecture. The most common used is SAR (*Successive Approximation Register*) ADC, also integrated in RP2040.

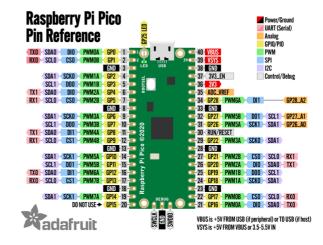
RP2040's ADC

channels	5
sampling rate	500 kHz
resolution	12 bits
V_{max}	3.3 V

- requires a 48 MHz clock signal
- channel 4 is connected to the internal temperature sensor

$$t=27-rac{(V_{input_4}-0.706)}{0.001721}[\degree C]_{SI}$$







in Embassy

```
use embassy rp::adc::{Adc, Channel, Config, InterruptHandler};
     bind interrupts!(struct Irgs {
         ADC IRQ FIFO => InterruptHandler;
     });
 6
     let p = embassy rp::init(Default::default());
     let mut adc = Adc::new(p.ADC, Irgs, 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);
         let voltage = 3300 * level / 4095;
15
16
         info!("Pin 26 voltage: {}.{}V", voltage / 1000, voltage % 1000);
         Timer::after secs(1).await;
17
18 }
```

Conclusion

we talked about

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

