PROGRAMMERING AV INBYGGDA SYSTEM

Interrupts and timers

Dario Salvi, 2025

Materials

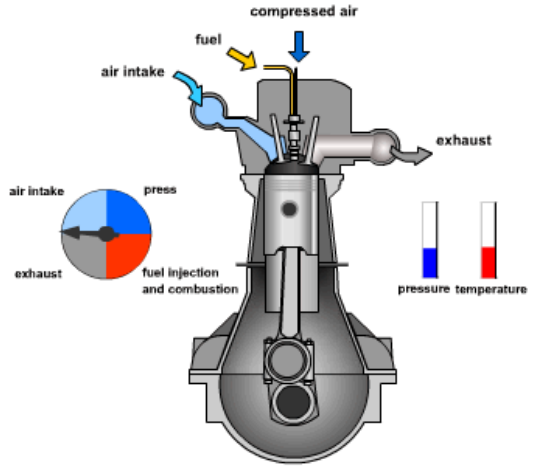
- ESP32 Technical Reference Manual: Interrupt Matrix and PID Controller, 64-bit Timers
- Making embedded system: chapter 4 “Outputs, Inputs, and Timers” and chapter 5 “Task management”
- Online resources:
  - https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/intr_alloc.html
  - https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/gpio.html
  - https://www.tutorialspoint.com/embedded_systems/es_interrupts.htm
  - https://www.tutorialspoint.com/embedded_systems/es_timer_counter.htm
  - https://www.cprogramming.com/tutorial/function-pointers.html
  - https://www.geeksforgeeks.org/function-pointer-in-c/
  - https://barrgroup.com/embedded-systems/how-to/c-volatile-keyword
  - https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/system_time.html

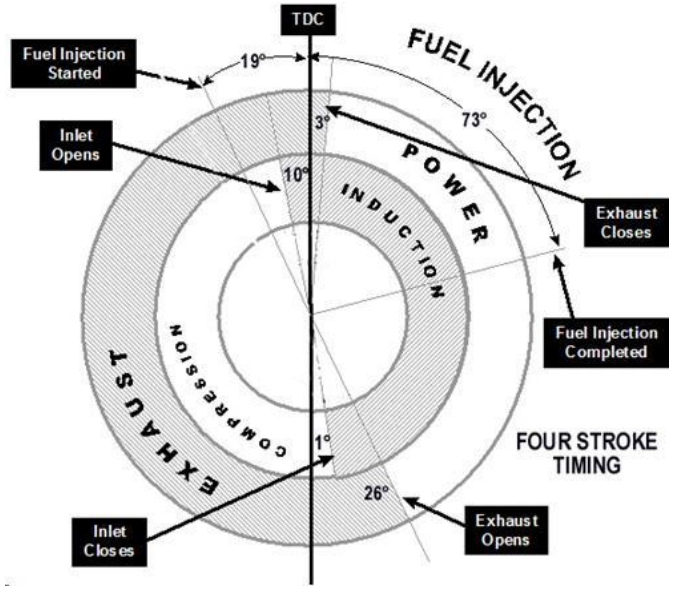
Real time requirements

- Suppose a Diesel engine with
  - Max RPM: 5000 rpm
  - N of injections per revolution: 5

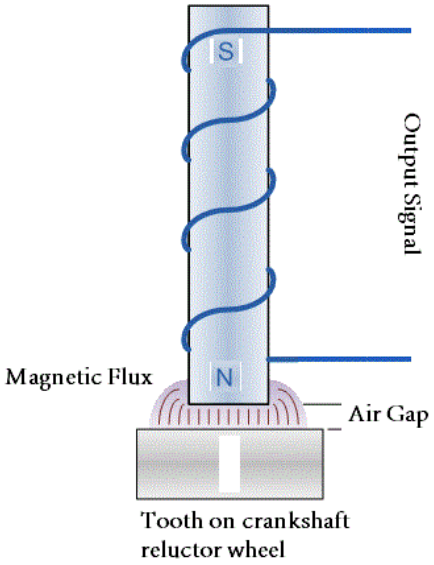
- Accuracy:
  - Cylinder position is determined by crank shaft angle
  - 5 injections during 70° of the full revolution
  - 14° between injections -> accuracy in degrees 5°
  - 1 full rotation at 5000 rpm takes 12ms
  - 5° take 0.14ms

- Crank shaft angle must be known within 0.14ms
- Sampling rate of 7 KHz



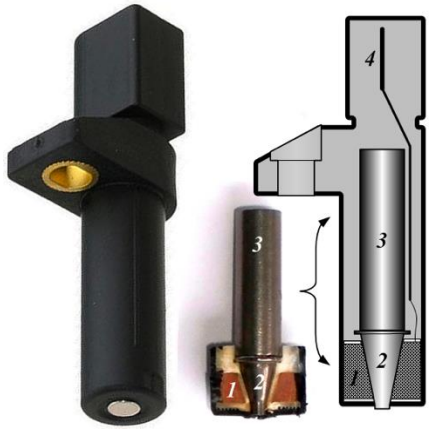


Inductive sensor



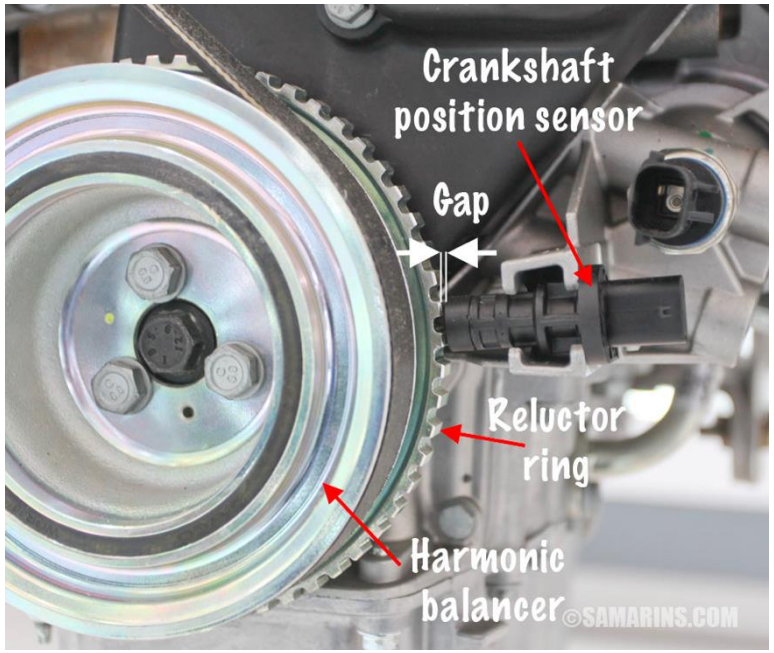
- Inductance describes the tendency of an electrical conductor, such as coil, to oppose a change in the electric current through it.
- Voltage depends on the rate of change of the current and the inductance.

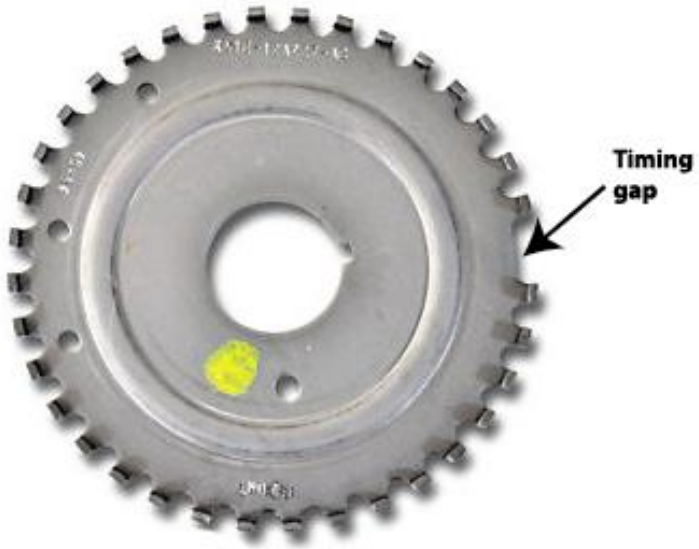
$$v = -L\frac{di}{dt}$$

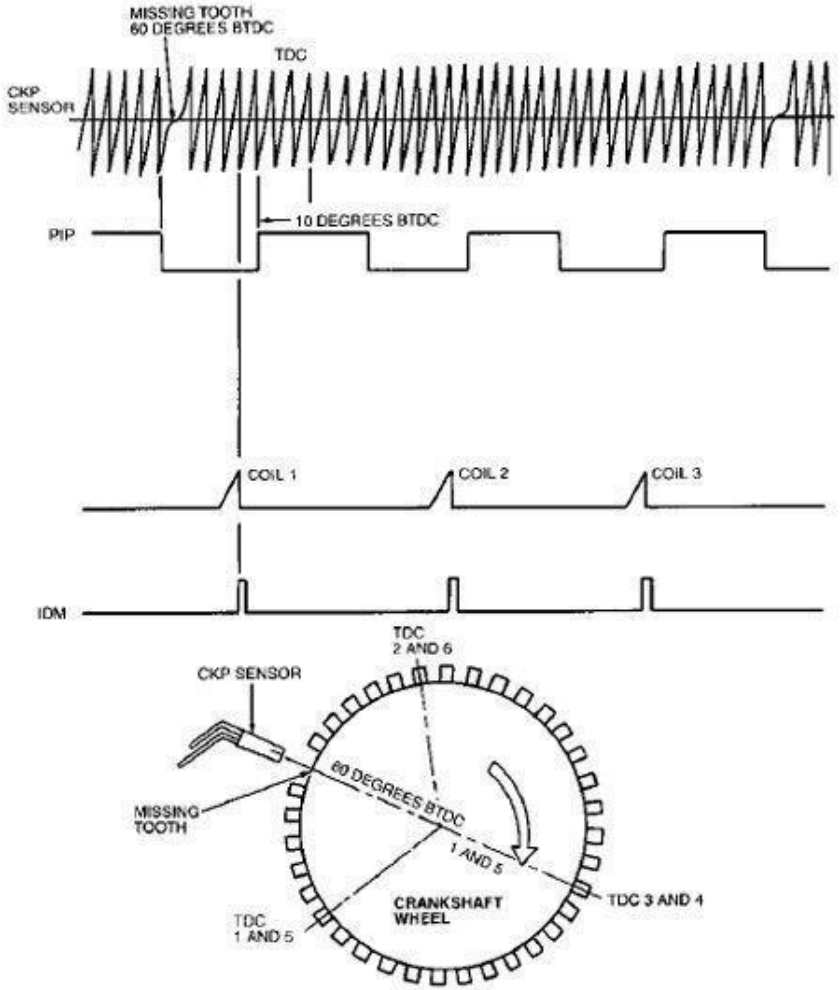


- The sensor measures changes in L, which gives a change in V as output.
- Can only measure movement, never static positions!
- Easy to measure by an MCU.
- Physical contact is not necessary → maintenance free, long life.

Crankshaft position sensor







To think about

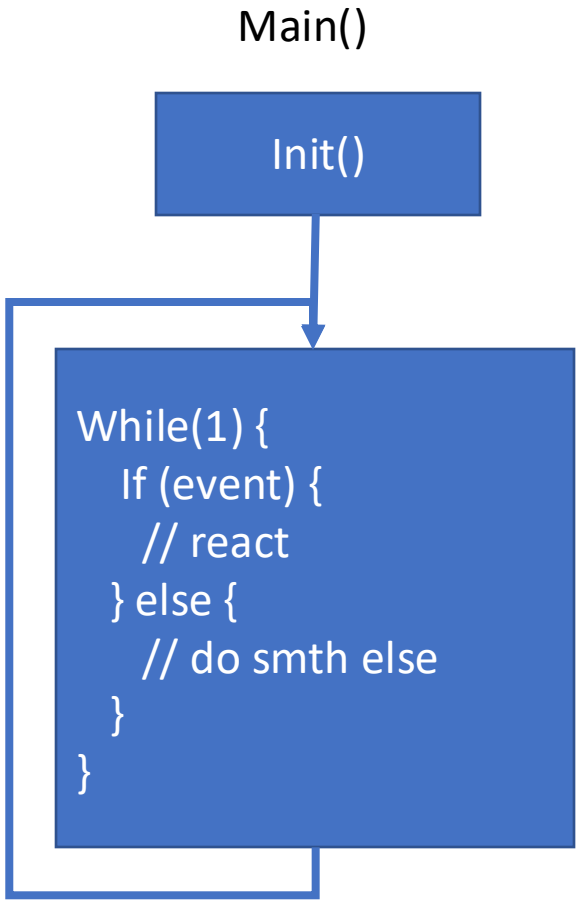
- How would you program your ESP32 so that it can measure rotations reliably?
  - Would you set a pin as input and read its value?
  - Would you use an infinite loop?

Reacting to events

- In an embedded system, there are always external events that the system should react to:
  - Buttons pressed
  - Inputs change value
  - Time passes

- Two strategies for dealing with this
  - **Polling**: You regularly check (read) each input or clock variable.
  - **Interrupts**: The external device alerts when something of interest happens.

Polling



Interrupts

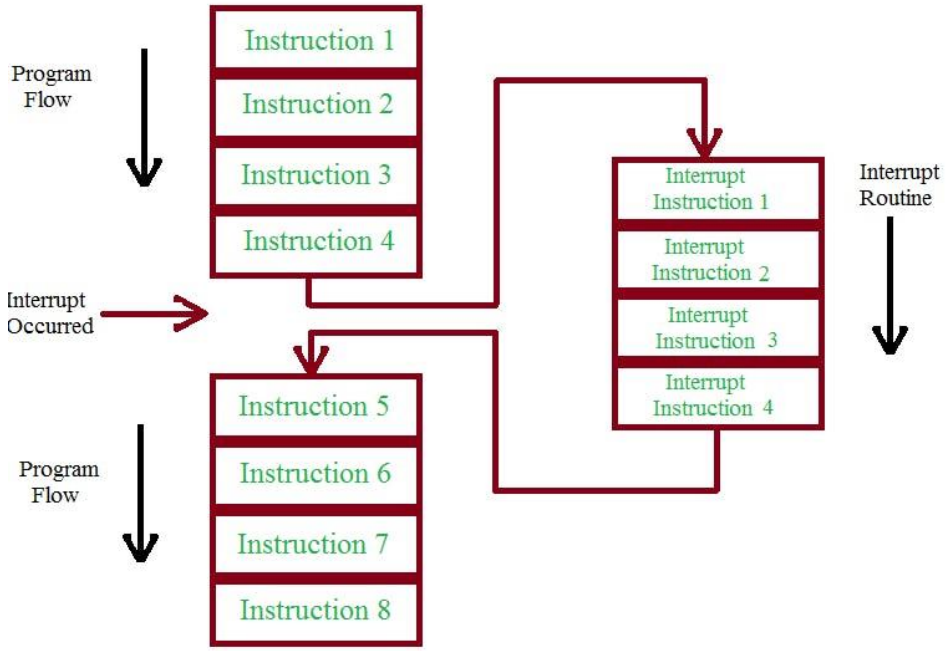
- **An event that interrupts the regular program**
  - Then the system deals with the cause
- The event needs to be dealt with **quickly**.
- Interrupts may have different sources
  - Hardware, such as external inputs (pins), timers or serial ports
  - Software, i.e. embedded in the program code

- As a programmer, one can usually control whether an interrupt should be allowed or not (maskable)

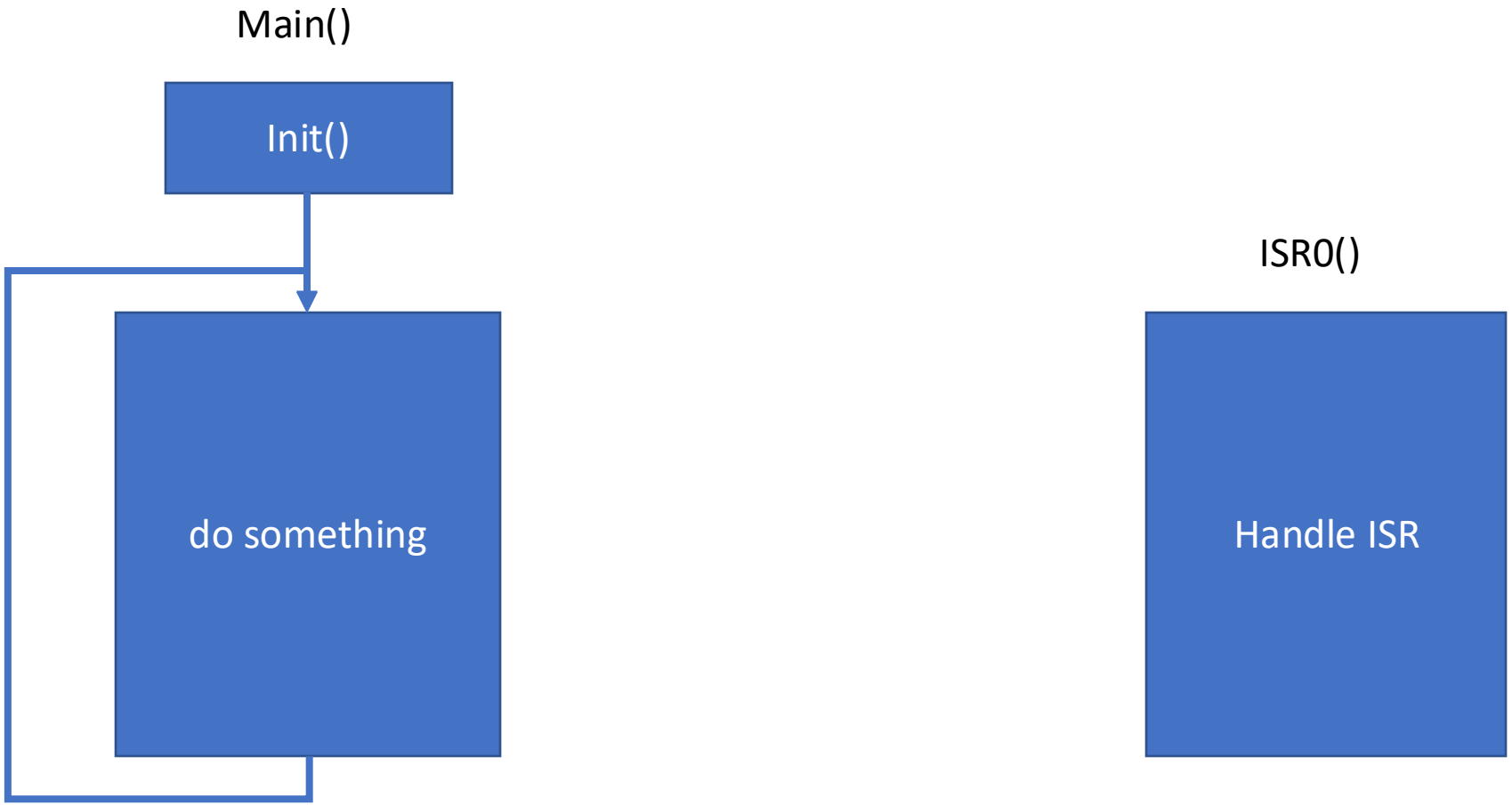
Handling interrupts

- Interrupt handler or interrupt service routine (**ISR**):
  - Code that handles interrupt.
  - Usually does not count as part of the main program.
  - Usually only allows little memory / time.
  - If complex execution is needed, it is delegated to the main program.

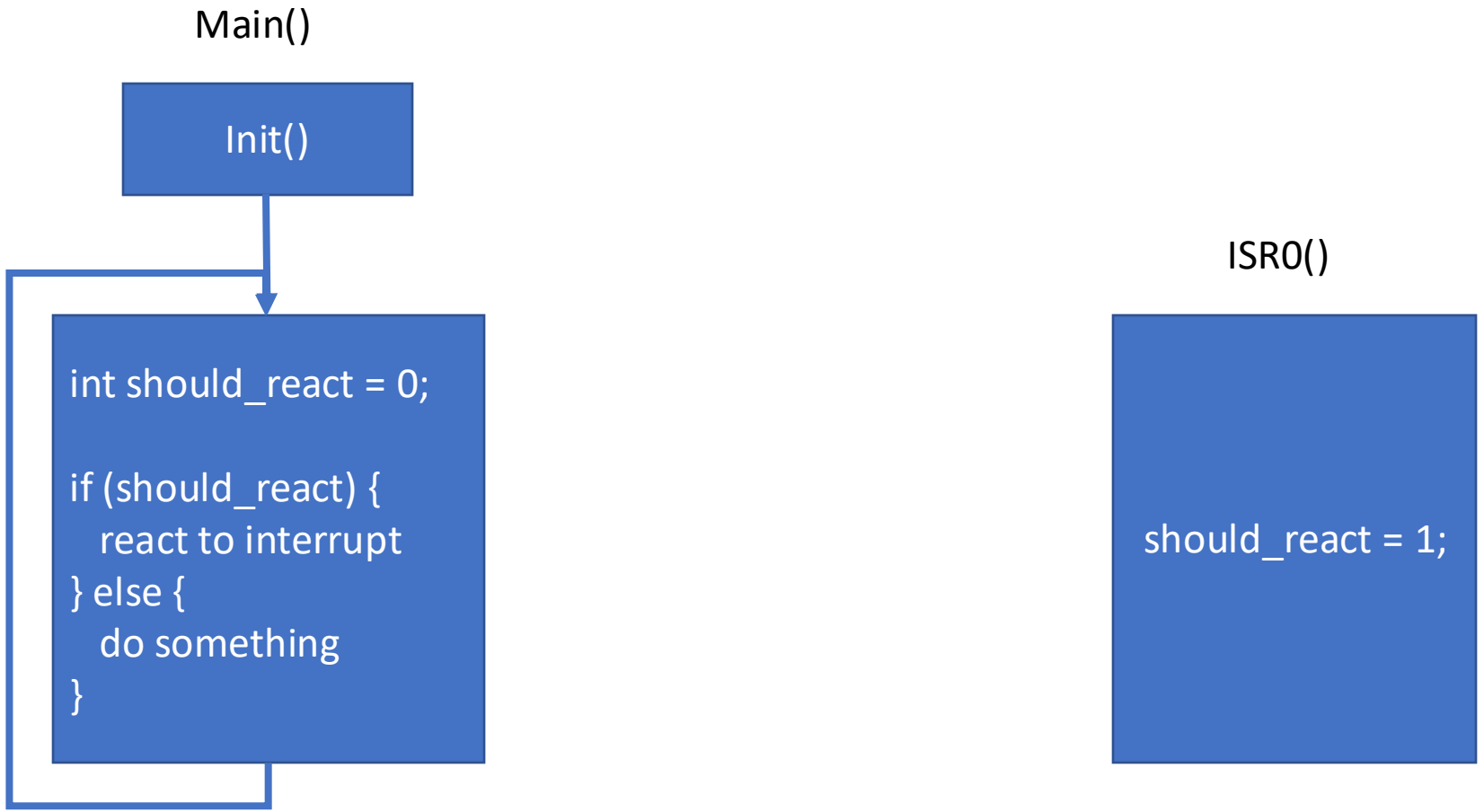
- Each processor family handles interrupts differently!



Handle exception in ISR



Handle exception in main



Interrupt Vector Table

- Each ISR is located somewhere in memory.
- The processor determines which ISR to call by looking in the **interrupt vector table** (IVT).
- This table is located at a specific area of memory.
- The interrupt vector table is **a list of callback functions**, one for each type of interrupt. Really, the vector table is just a list of pointers to functions.
- Each interrupt has a number, the lookup in the table is done by adding this number multiplied by the size of addresses to a base address:

  ISR_n = BASE_TABLE_ADDR + (n * SIZE_OF_ADDR)

Calling an ISR

- When an interrupt occurs, the CPU:
  - **Saves the the current state** of the the processor ("where you are in the program").
  - **Reads the address** in the vector for that particular interrupt.
  - **Jumps to the address** and executes the ISR code.
- When the ISR code is completed, the MCU returns to where it was in the program execution before.

- Very similar to adding a function to the stack and executing it, but ISRs need to additionally *restore the context* when they end.
  - In fact, at the end of their execution, functions and ISRs use different machine code operations.

Saving the context

- On an interrupt the "**context**" is saved on the stack:
  - ❖The current program counter
  - ❖Stack pointer
  - ❖A number of registers (a sort of local cache of the RAM)
- The context must then be restored after exiting the ISR.
  - This to enable the main program to continue executing where it was before the interrupt occurred.
- Context switching takes time (called "system latency").
  - E.g. in a 30MHz processor, handling audio in an interrupt at 44100Hz with a 10-cycle latency uses 1.47% of the CPU time simply calling the interrupt handler.

Calling functions in an ISR

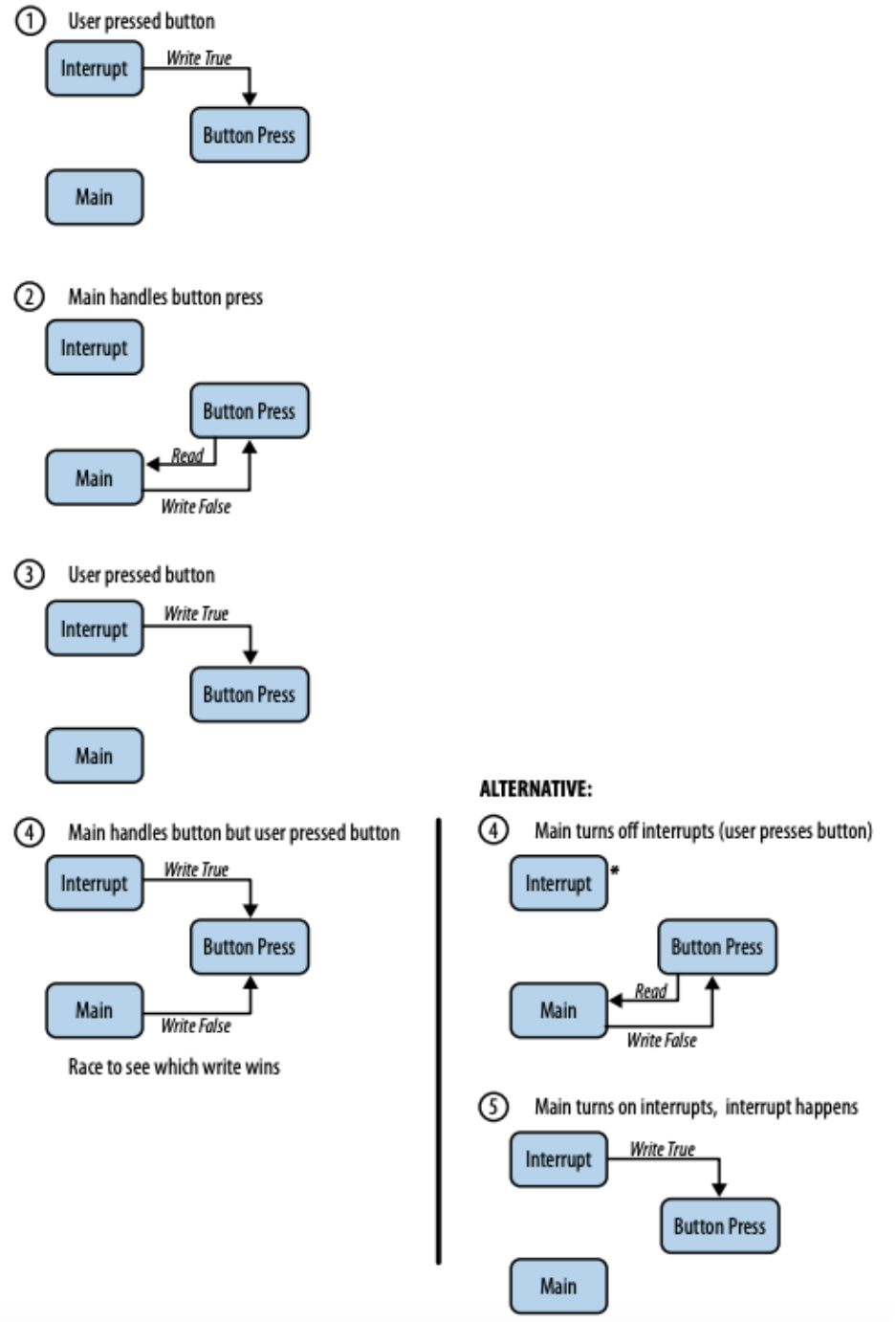
- Calling functions takes some (little) additional time that, in an ISR, adds up.
- Functions may modify global variables, which may be subject to *race conditions*.
- A function that can be safely called multiple times while it is running is called **re-entrant**.

```
int t;
void swap(int* x, int* y) {
    t = *x;
   *x = *y;
    // hardware interrupt might invoke isr() here!
    // and swap may be called again
    *y = t;
}

void isr () {
    int x = 1, y = 2;
    swap(&x, &y);
}
```

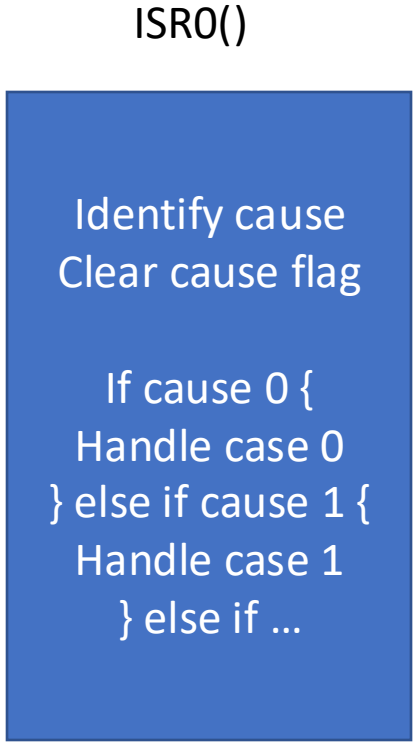
Race conditions

- To avoid race conditions, you can always **inhibit interrupts** from calling ISRs.
- This should be done <u>whenever a shared variable is changed</u> and at the very beginning of the ISR.
- The interrupt should be re-allowed at the end of the ISR.



Shared Interrupts

- To save space for the Interrupt Vector Table, interrupts are shared among different causes.
  - Example: 8 pins share the same interrupt.
- Once the ISR is called, the code must understand **which cause** actually triggered the interrupt. This is done by looking at a register.
  - Example: the ISR checks, one by one, which pins activated the interrupt.
- Some processors require ISRs to ”clear” the flag related to the cause or they’ll fire the interrupt again.



ISR prototype

1. Disable interrupts to avoid race conditions (all interrupts, or, if we are sure it’s OK, just the ones of the same type)
2. Identify the cause of the interrupt (and clear it if needed)
3. Deal with the cause, quickly, avoid calling non re-entrant functions
4. Enable interrupts
5. If anything is left that is race-safe, do it here

Nested interrupts

- Some processors allow **interrupts within interrupts**.
- Since the code in ISR is executed like any other code, another interrupt may interrupt it (nested interrupt).
- A **priority** can be used to determine whether an interrupt can supersede the one currently running.
- **Only a higher priority interrupt can interrupt the current ISR**.
- In the ESP32 there are **seven** priority levels: Level 1, Level 2, Level 3, Level 4, Level 5, Level 6 (Debug), and NMI (non maskable).
- Each level of interrupt is assigned an interrupt vector entry address.
- The priority level is usually determined during the initialization but can be changed dynamically.

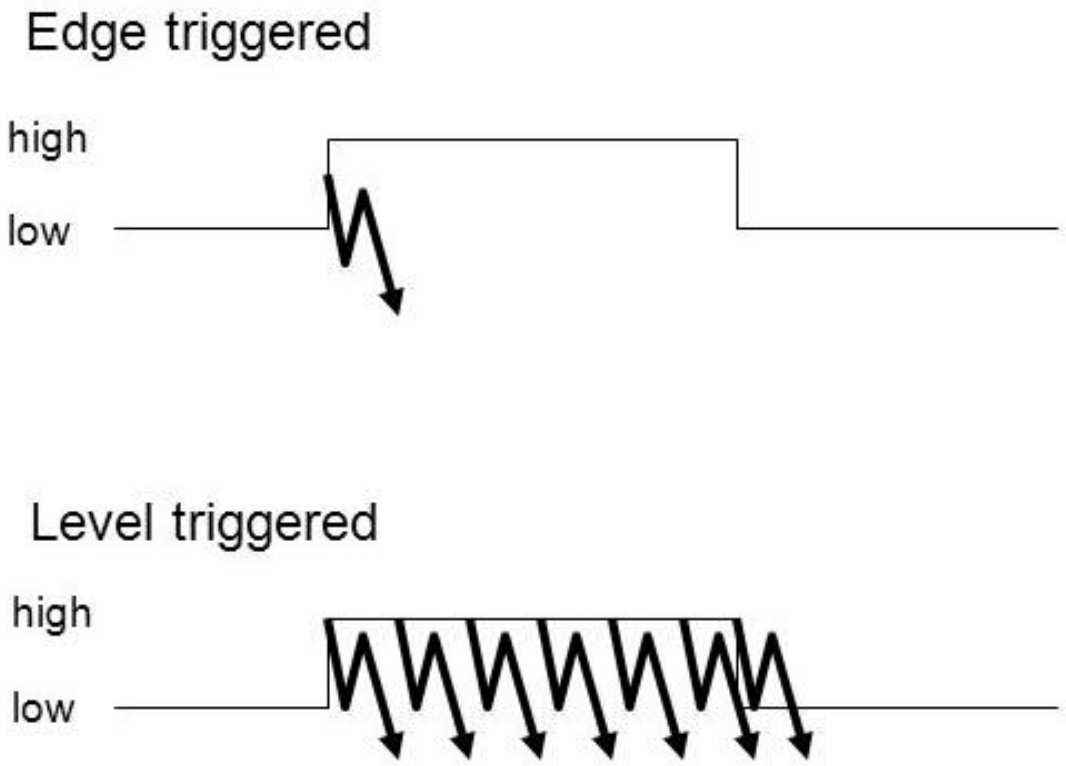
Software interrupts

- Some interrupts can be caused by the code itself. These are sometimes called *exceptions*.
- Examples:
  - Exiting the execution of the program (fatal error).
  - Accessing functionalities offered by the operating system.

IO interrupt

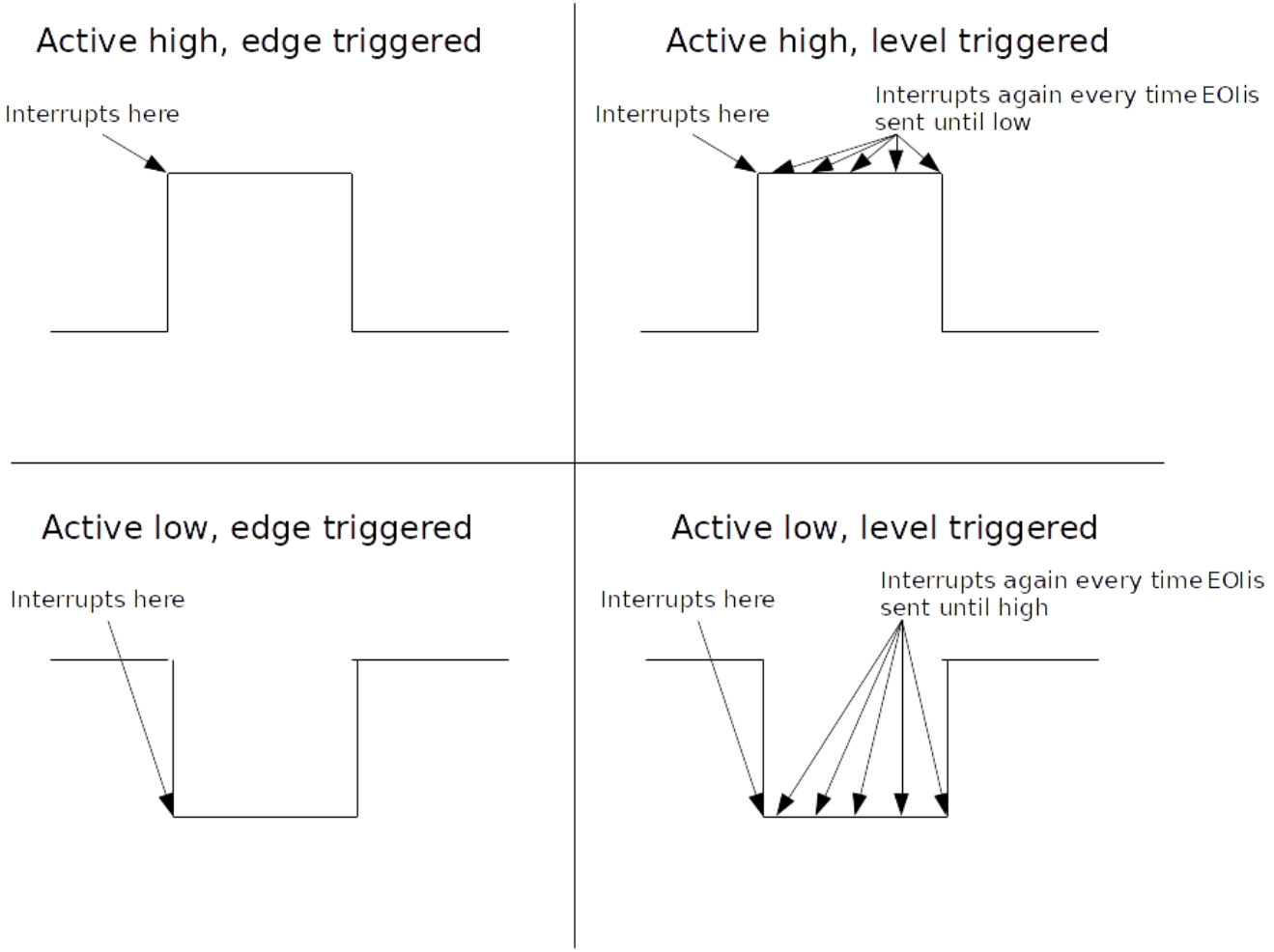
- A physical pin or a port generates an interrupt when it receives a signal.
- You can configure what should trigger an interrupt:
  - Signal **edge** (rising or falling)
  - Signal **level** (high / low)
- Examples causes of IO interrupts:
  - Button pressed.
  - A / D conversion ready
  - Communication:
    - USART received characters
    - USART transmission of characters completed

Edge vs Level Triggered Interrupts



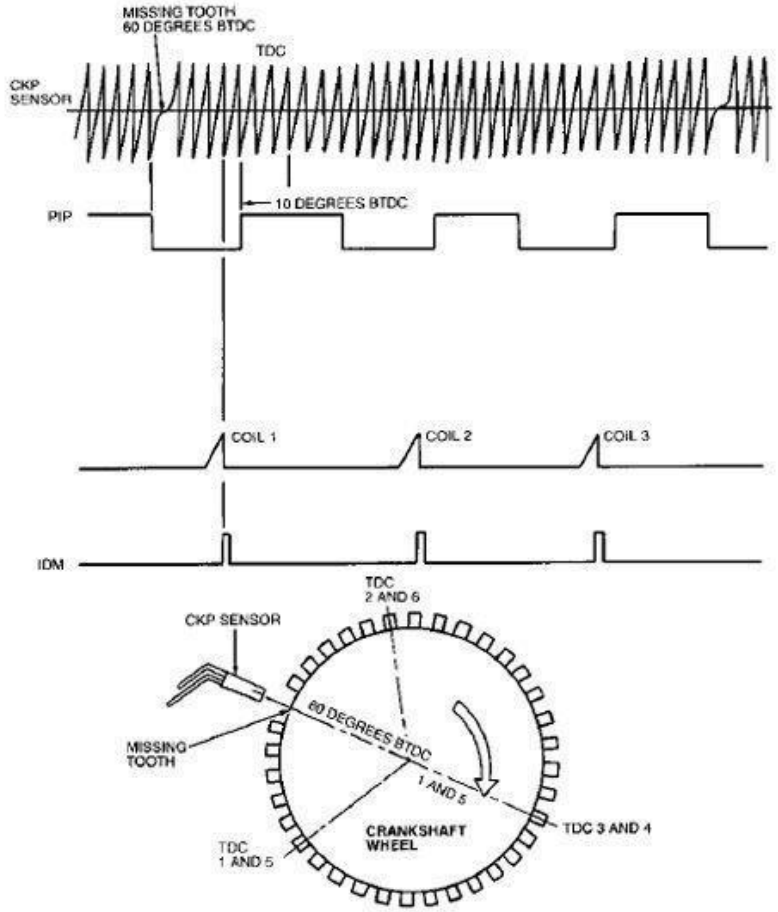
- Edge triggered interrupts signal as a one shot event!
- Level triggered interrupts are signaled as long as line is raised

Raising / Falling Edge and High / Low Level



To think about

- What would you use to measure rotations of a crank shaft?
  - Interrupts? Level or edge? Falling or raising?



Callbacks

"In computer programming, **a callback is a reference to executable code, or a piece of executable code, *that is passed as an argument* to other code**. This allows a lower-level software layer to call a subroutine (or function) defined in a higher-level layer."

Wikipedia

- Callbacks can be used as parameters when setting up an ISR.
  - setISR(ISR_N, callback)

Pointer to functions in C

- A **function pointer** is a variable that stores the address of a function that can later be called through that function pointer.
- This is useful because functions encapsulate behaviour. For instance, every time you need a particular behaviour such as handling an event, instead of writing out a bunch of code, all you need to do is call the function. But *sometimes you would like to choose different behaviours at different times in essentially the same piece of code*.

- Function pointers are used to pass **callbacks** in C.

Live coding

- Example of how to use a pointer to a function in C.

Interrupts in the ESP32

- ESP32 has **71 peripheral interrupt sources** in total.
- 67 of 71 ESP32 peripheral interrupt sources can be allocated to either CPU (named PRO and APP).
- The four remaining peripheral interrupt sources are CPU-specific, two per CPU.
  - GPIO_INTERRUPT_PRO and GPIO_INTERRUPT_PRO_NMI can only be allocated to PRO_CPU.
  - GPIO_INTERRUPT_APP and GPIO_INTERRUPT_APP_NMI can only be allocated to APP_CPU.

CPU interrupts

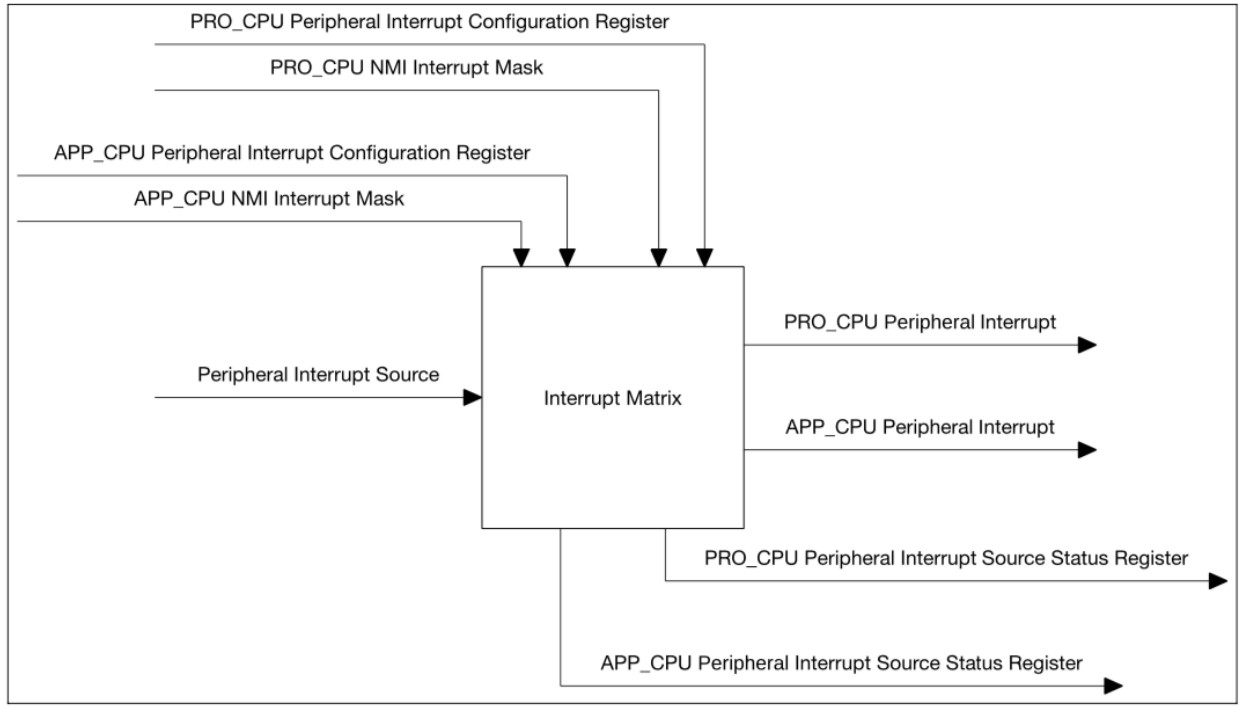
- Both of the two CPUs (PRO and APP) can handle 32 interrupts each, of which 26 are peripheral interrupts.
- The internal ones are:
  - Three timer comparators
  - A performance monitor
  - Two software interrupts.

- Interrupts are routed to the CPUs through a **configurable matrix**.

No.	Category	Type	Priority Level
0	Peripheral	Level-Triggered	1
1	Peripheral	Level-Triggered	1
2	Peripheral	Level-Triggered	1
3	Peripheral	Level-Triggered	1
4	Peripheral	Level-Triggered	1
5	Peripheral	Level-Triggered	1
6	Internal	Timer.0	1
7	Internal	Software	1
8	Peripheral	Level-Triggered	1
9	Peripheral	Level-Triggered	1
10	Peripheral	Edge-Triggered	1
11	Internal	Profiling	3
12	Peripheral	Level-Triggered	1
13	Peripheral	Level-Triggered	1
14	Peripheral	NMI	NMI
15	Internal	Timer.1	3
16	Internal	Timer.2	5
17	Peripheral	Level-Triggered	1
18	Peripheral	Level-Triggered	1
19	Peripheral	Level-Triggered	2
20	Peripheral	Level-Triggered	2
21	Peripheral	Level-Triggered	2
22	Peripheral	Edge-Triggered	3
23	Peripheral	Level-Triggered	3
24	Peripheral	Level-Triggered	4
25	Peripheral	Level-Triggered	4
26	Peripheral	Level-Triggered	5
27	Peripheral	Level-Triggered	3
28	Peripheral	Edge-Triggered	4
29	Internal	Software	3
30	Peripheral	Edge-Triggered	4
31	Peripheral	Level-Triggered	5

Interrupt matrix

- To allocate a peripheral interrupt X to a CPU peripheral interrupt Y you configure the routing matrix through the Peripheral Interrupt Configuration Registers.



External / internal peripherals

- **External peripherals**, within the ESP32 but outside the Xtensa cores themselves. Most ESP32 peripherals are of this type.
  - Both CPU cores are wired to an interrupt multiplexer, which can be used to route any external interrupt source to any of these interrupt slots.
  - Allocating an external interrupt will always allocate it on the core that does the allocation.
  - Freeing an external interrupt must always happen on the same core it was allocated on.
  - Disabling and enabling external interrupts from another core is allowed.
- **Internal peripherals**, part of the Xtensa CPU cores themselves.
  - These peripherals can only be configured from the core they are associated with. When generating an interrupt, the interrupt they generate is hard-wired to their associated core.

Masking and querying

- Interrupts can be temporarily *masked* through the **Non-Maskable Interrupt** (NMI) register.
- The current interrupt status of a peripheral interrupt source can be read via the bit value in the **Peripheral Interrupt Source Status** registers.

Interrupts in the ESP-IDF

- Each interrupt has a certain priority level, most (but not all) interrupts are connected to the interrupt mux.
- Because there are more interrupt sources than interrupts, sometimes it makes sense to share sources in a single ISR.
- The **esp_intr_alloc()** abstraction exists to hide all these implementation details.
  - One can use the flags passed to this function to set the type of interrupt allocated, specifying a specific level or trigger method. The interrupt allocation code will then find an applicable interrupt, use the interrupt mux to hook it up to the peripheral, and install the given interrupt handler and ISR to it.

Interrupts in the ESP-IDF

- **Non-shared interrupts**: a separate interrupt is allocated per esp_intr_alloc call and this interrupt is solely used for the peripheral attached to it, with only one ISR that will get called.
  - Non-shared interrupts can be either level- or edge-triggered.
- **Shared interrupts** can have multiple peripherals triggering it, with multiple ISRs being called when one of the peripherals attached signals an interrupt. Thus, ISRs that are intended for shared interrupts should check the interrupt status of the peripheral they service in order to see if any action is required.
  - Shared interrupts can only be *level interrupts*.
    - The logic behind this: DevA and DevB share an int. DevB signals an int. Int line goes high. ISR handler calls code for DevA -> does nothing. ISR handler calls code for DevB, but while doing that, DevA signals an int. ISR DevB is done, clears int for DevB, exits interrupt code. Now an interrupt for DevA is still pending, but because the int line never went low (DevA kept it high even when the int for DevB was cleared) the interrupt is never serviced.

Abstractions for GPIOs

- esp_err_t *gpio_isr_register*(void (*fn)(void *), void *arg, int intr_alloc_flags, gpio_isr_handle_t *handle,)
  - Register GPIO interrupt handler, the handler is an ISR. The handler will be attached to the same CPU core that this function is running on.
  - This ISR function is called whenever **any GPIO interrupt** occurs.
- esp_err_t *gpio_install_isr_service*(int intr_alloc_flags)
  - allows **per-pin GPIO interrupt handlers**
  - the ISR service provides a global GPIO ISR and individual pin handlers are registered via esp_err_t *gpio_isr_handler_add*(gpio_num_tgpio_num, gpio_isr_tisr_handler, void *args)
  - For flags, use 0 for default
- The 2 functions are **incompatible**.

https://docs.espressif.com/projects/esp-idf/en/v4.4.1/esp32/api-reference/peripherals/gpio.html

Live coding

- Example of how to setup IO interrupts for the ESP32 using ESP-IDF.

Volatile

- C's **volatile** keyword is a qualifier that is applied to a variable when it is declared. It tells the compiler to **avoid certain optimizations** because the value of the variable may change at any time.
- A variable should be declared volatile whenever its *value could change unexpectedly*. In practice, only three types of variables could change:
  1. Memory-mapped peripheral registers
  2. Global variables modified by an interrupt service routine
  3. Global variables accessed by multiple tasks within a multi-threaded application
- Examples:
  - Volatile integer:
    - volatile uint16_t x; or uint16_t volatile y;
  - Pointer to volatile variable:
    - volatile uint8_t * p_reg; or uint8_t volatile * p_reg;

ESP32 and IRAM

- The ESP32 follows the Harvard architecture.
- RAM is divided into 2 types:
  - Instruction RAM: contains instructions
  - Data RAM: contains data and operands
- During Write and Erase operations the FLASH memory is slow -> the ISR is unable to be loaded at the time that it is needed because flash is busy -> crash
- ISR and all functions called inside need to be placed in IRAM or ROM, not Flash.
- All data used inside an ISR must be located in DRAM, not Flash.
- **gpio_isr_handler_add** already allocates the ISR in IRAM, otherwise a the **IRAM_ATTR** compiler flag will ensure that the ISR function is placed in the IRAM area instead of the flash area.
- For data, use the **DRAM_ATTR** attribute.

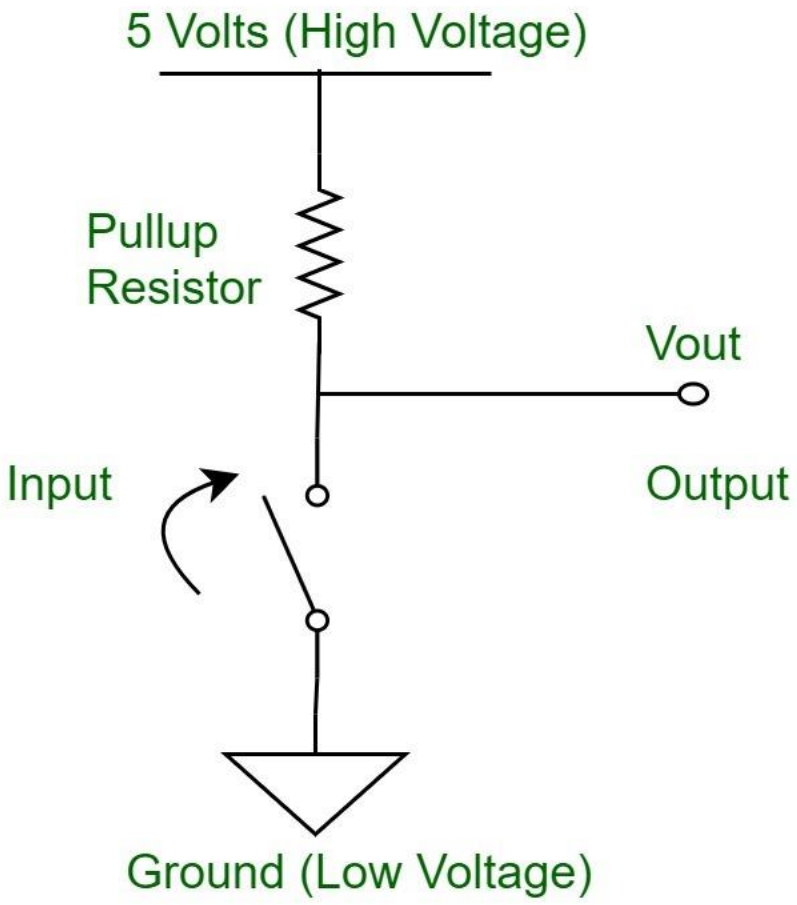
https://espressif-docs.readthedocs-hosted.com/projects/esp-idf/en/latest/api-guides/general-notes.html

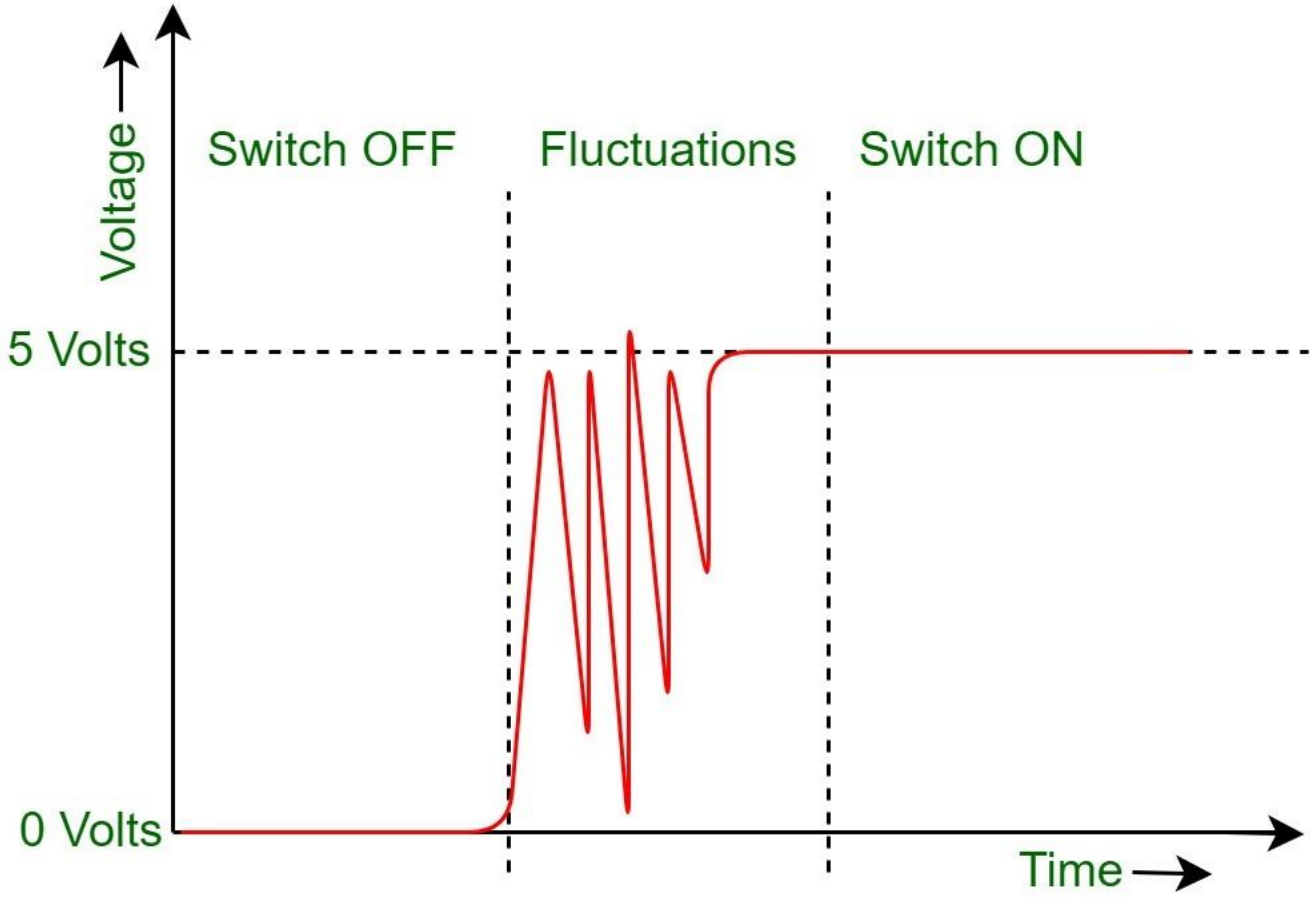
https://espressif-docs.readthedocs-hosted.com/projects/esp-idf/en/latest/api-reference/system/intr_alloc.html#iram-safe-interrupt-handlers

Printf()

- ISR may only call functions placed into IRAM or functions present in ROM.
  - -> many functions of the ESP-IDF not available!
- Some functions may be still available in RAM though!
  - Try with esp_rom_printf() instead of regular printf()
  - Otherwise try the Logging library, particularly ESP_DRAM_LOG**X**(…) where X is the log level (**E**rror, **W**arn, **I**nfo, **D**ebug, **V**erbose).
  - Not much documentation available!
- Also strings must be in DRAM (not Flash)!
  - Use DRAM_STR(”my message") to ensure that they are there

Switch bouncing

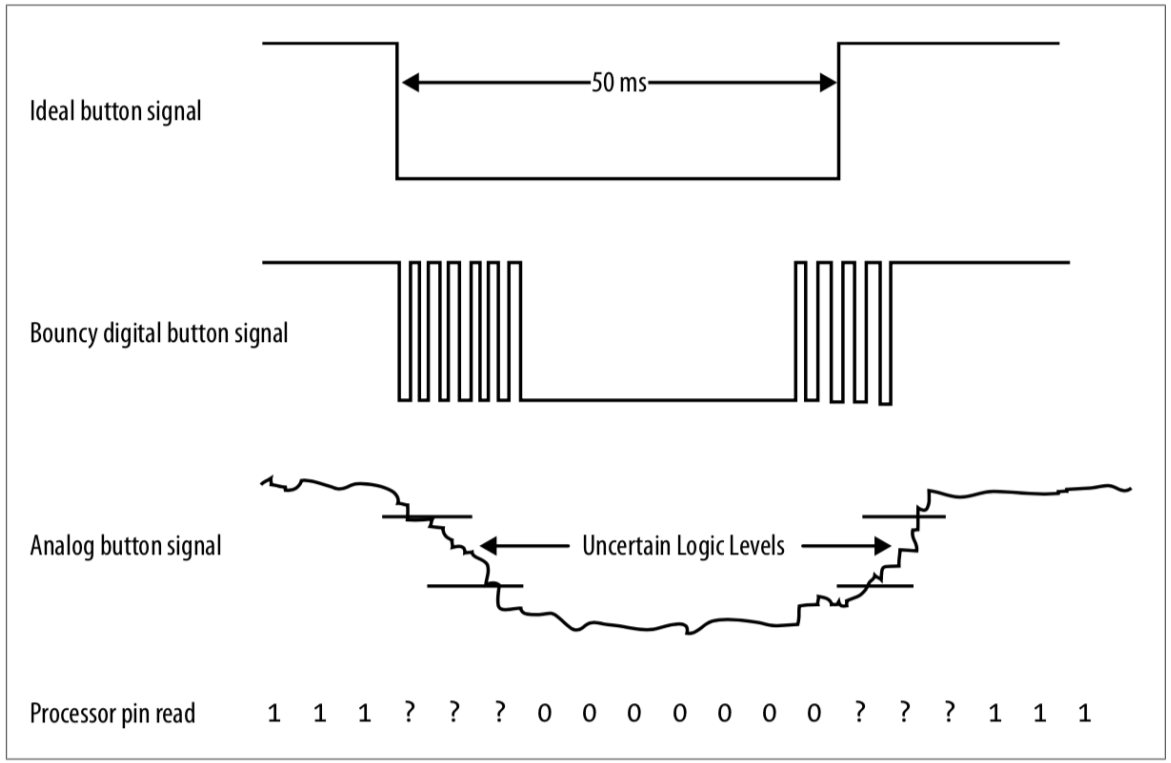




Not this bouncing…

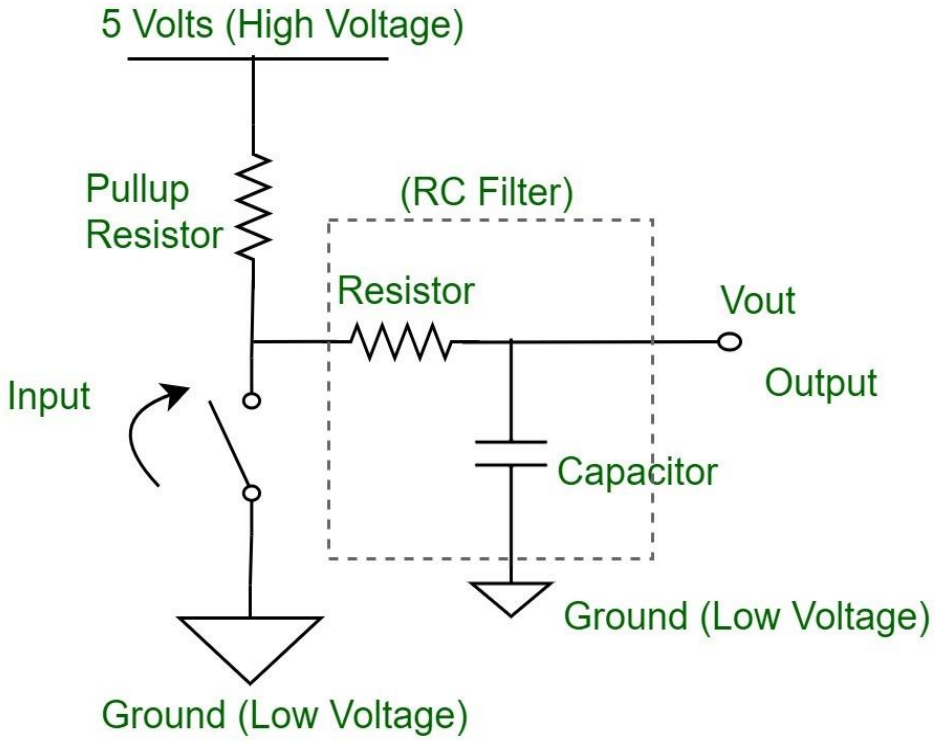
Bouncing

- *Switch bouncing* is due to an **inductive effect** caused by the switch as the state of the signal transitions from one to another, and hence there is an associated change in the electric field. The switch is, in effect, *acting as an inductor*. Coupled with transmission line effects in the signal trace, this is the source of the anomaly.
- **It is not a mechanical bounce.**



De-bouncing

Hardware:



Software:

```
PUSH_TIME = reasonable amount of time needed to push
counter = 0;
lastPush = -PUSH_TIME;

ISR_handler()
{
   now = get_time();
   if (now - lastPush > PUSH_TIME_US)
   {
      lastPush = now;
      counter++;
   }
}
```

IRAM and DRAM

- The ESP32 follow the *Harvard architecture*
- Internally, there is one bus for instructions and one bus for data
  - -> Optimization: the next instruction can be fetched while data for an operation is retrieved!
- RAM is split into 2:
  - IRAM: for instructions (addresses >= 0x4000 0000)
    - Can only retrieve 32 bits at the time
  - DRAM: for data (addresses < 0x4000 0000)
    - Can retrieve single bytes
- You can ask the loader to place a function into IRAM with the IRAM_ATTR
- **It is recommended to load ISRs into IRAM**
  - Because otherwise the ISR would need to be loaded from the external flash which might be temporarily unavailable, and IRSs need to act **fast**!

https://esp32.com/viewtopic.php?t=5102

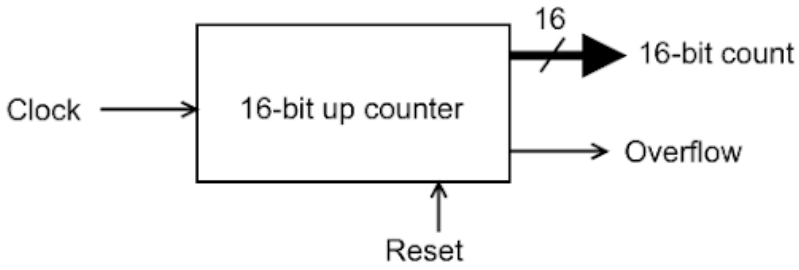
Live coding

- Example of debouncing a pushbutton and an ISR in IRAM.
- Documentation of the API: https://docs.espressif.com/projects/esp-idf/en/v5.0.2/esp32/api-reference/peripherals/gpio.html

To think about

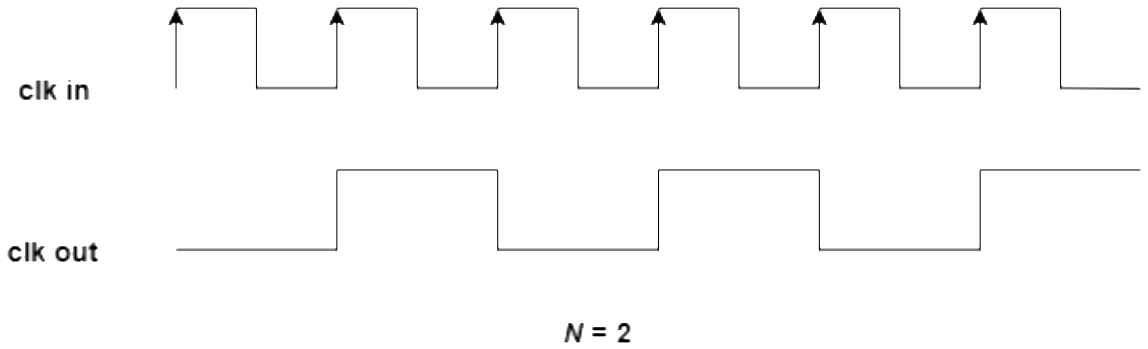
- The debouncing approach is very basic. What happens if the fluctuation is random?
- Can you think of a more robust software approach?
  - See “Making embedded systems” for a good answer.

Timers



- A timer is a simple counter measuring time by accumulating the number of clock ticks in a counter register.
- Each time the counter reaches a given value (set in a register), or when it *overflows*, an **action** is taken (an interrupt, a pin is toggled etc.)
- The more deterministic the master clock is, the more precise the timer can be.
- Timers operate independently of software execution, acting in the background without slowing down the code at all.

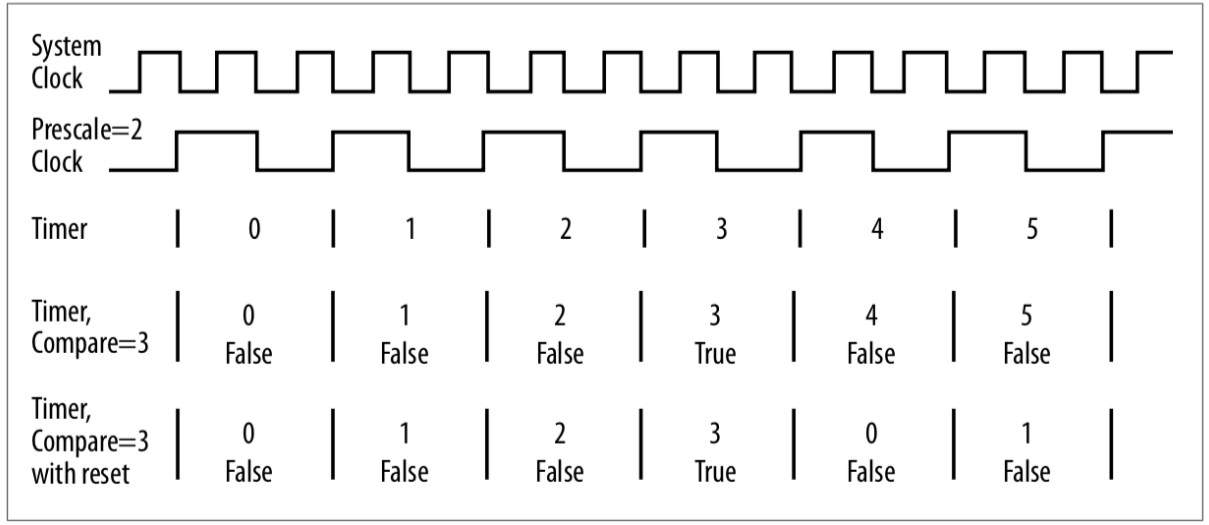
Prescalers



- The frequency of the timer is determined by the clock input. This may be the processor clock (*system clock* or *master clock*) or it may be a peripheral clock.
- If the clock of the timer is very fast, the counter will overflow very frequently, but in some applications you may need slower frequencies.
- Chips usually include a *prescaler*, which divides the clock so that the counter increments at a slower rate.
  - The prescaler outputs a signal with the frequency divided by N.

Compare register

- In addition to the prescaler, you can control the frequency with a *compare register*.
  - The timer counts up. The processor notes when the timer register (counter) matches the compare register.
  - When the timer matches, it will fire an action (alarm)
  - After: it may continue counting up or reset.

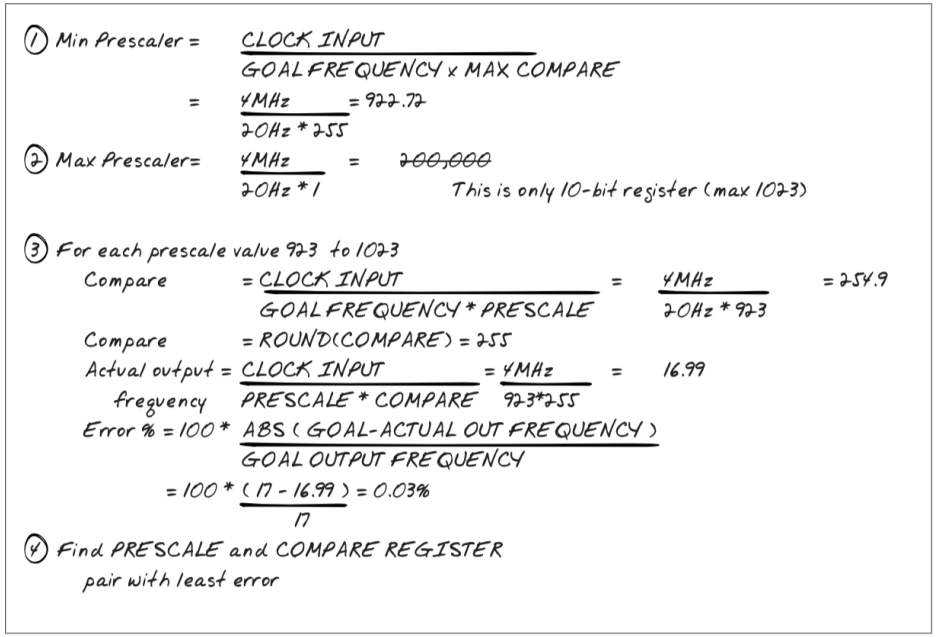


Producing a wanted frequency

- Given that the clock frequency is fixed, thanks to both the prescaler and the compare register you have some flexibility to trigger an interrupt at the frequency you want:
  - frequency = clock/(prescaler * compareReg)
- You need to deal with the limits in size! Example:
  - Suppose we have a clock of 4MHz and a compare register of 1 byte and a prescaler register of 10 bit.
  - We want an output frequency of 20Hz.
  - prescaler * compareReg = 4Mhz/20Hz = 200,000
  - This can be solved with:
    - Prescaler = 1000 (fits into 10 bits)
    - CompareReg = 200 (fits into 8 bits)

The solution is not always obvious!

1. Detect minimum and maximum prescaler (by setting compareReg to 1 and to its max).
2. Within those ranges, find the compareReg value that minimises the difference between wanted freq. and actual freq.
3. To further fine-tune it, you can deal with it programmatically in the ISR.
   - For example, if you need an even lower frequency than the minimum you can generate with max prescaler and compareReg



Typically involved registers with timers

- *Clock configure register (optional)*
  - This register tells a subsystem which clock source to use, though the default may be the system clock. Some processors have timers that even allow a clock to be attached to an input pin.
- *Timer counter*
  - This holds the changing value of the timer, the number of ticks since the timer was last reset.
- *Control register*
  - This sets the timer to start counting once it has been configured. The control register also often has a way to reset the timer.
- *Prescale register*
  - This divides the clock so that it runs more slowly, allowing timers to happen for relatively rare events.
- *Compare (or match) register*
  - When the timer counter equals this register, an action is taken. There may be more than one compare register for each timer.
- *Action register*
  - This register sets up an action to take when the timer and compare register are the same.
    - Interrupt (or not)
    - Stop or continue counting
    - Reset the counter (or not)
    - Set an output pin to high, low, toggle, or nothing
- *Interrupt register (may be multiple)*
  - If you have timer interrupts, you will need to use the appropriate interrupt register to enable, clear, and check the status of each timer interrupt.

Timers in ESP32

- The ESP32 chip contains **two hardware timer groups**.
- Each group has **two general-purpose hardware timers**.
- They are all **64-bit generic timers** based on 16-bit prescalers and 64-bit up / down counters which are capable of being auto-reloaded.
- Timers can be associated to an interrupt using the interrupt matrix.

Clocks

- The ESP32 can use an external crystal oscillator, an internal PLL or an oscillating circuit as a clock source.
- Internal clocks:
  - PLL_CLK: internal PLL clock with a frequency of 320 MHz or 480 MHz (usually used for the CPU and timers).
  - RTC8M_CLK: internal low-power clock with a default frequency of 8 MHz (adjustable).
  - RTC8M_D256_CLK: derived by RTC8M_CLK dividing its frequency by 256.
  - RTC_CLK: internal, low power, at 150 KHz (adjustable).
  - APLL_CLK: internal Audio PLL clock with a frequency range of 16 ~ 128 MHz

Programming timers

- First of all, you initialize the timer with **gptimer_new_timer()** by giving it a **gptimer_config_t** struct containing:
  - **clk_src**: the clock to be used. Default: GPTIMER_CLK_SRC_DEFAULT (this works at 80 MHz)
  - **direction**: GPTIMER_COUNT_UP or GPTIMER_COUNT_DOWN
  - **resolution_hz**: frequency in Hz
- The function will take care of finding an available hardware timer and setup the prescaler.
- The function needs a pointer to a **gptimer_handle_t** as second argument, which will pass the instance of the timer (needed later to interact with the timer)

Programming alarm

- To configure the “action”, you need to use the **gptimer_set_alarm_action()** passing it a **gptimer_alarm_config_t** with:
  - **alarm_count**: at what value of the counter should the alarm fire (compare register)
  - **auto_reload_on_alarm**: if true, timer restarts after alarm
  - **reload_count**: value set at restart
- Add a function to be executed as an ISR with **gptimer_register_event_callbacks()**, this requires the callback to be passed in a struct with field **on_alarm**.
  - The callback must be of type **gptimer_alarm_cb_t**, which has a specific signature
  - Better placing the callback in the IRAM, because it is executed in an ISR

Starting/stopping the timer

- Start the timer with **gptimer_enable()** and **gptimer_start()**.
- Stop a timer with **gptimer_disable()** and get rid of an unused timer with **gptimer_del_timer()**.

Live coding

- Example of how to setup a timer with alarm and ISR.
- Documentation of the API: https://docs.espressif.com/projects/esp-idf/en/v5.0.2/esp32/api-reference/peripherals/gptimer.html

Application: esp_timer

- It’s a library that simplifies the use of timers and offers common use cases.
- It uses a specific timer, configured in the project.
- See https://docs.espressif.com/projects/esp-idf/en/v5.0.2/esp32/api-reference/system/esp_timer.html
- Examples:
  - **esp_timer_get_time**(): gets the number of μs since initialisation
  - **esp_timer_create**(): creates a timer to be used with this API
  - **esp_timer_start_periodic**() for periodic or **esp_timer_start_once**() for one-shot callbacks
  - **esp_timer_stop**(): stops the timer
  - **esp_timer_delete**(): deletes the timer

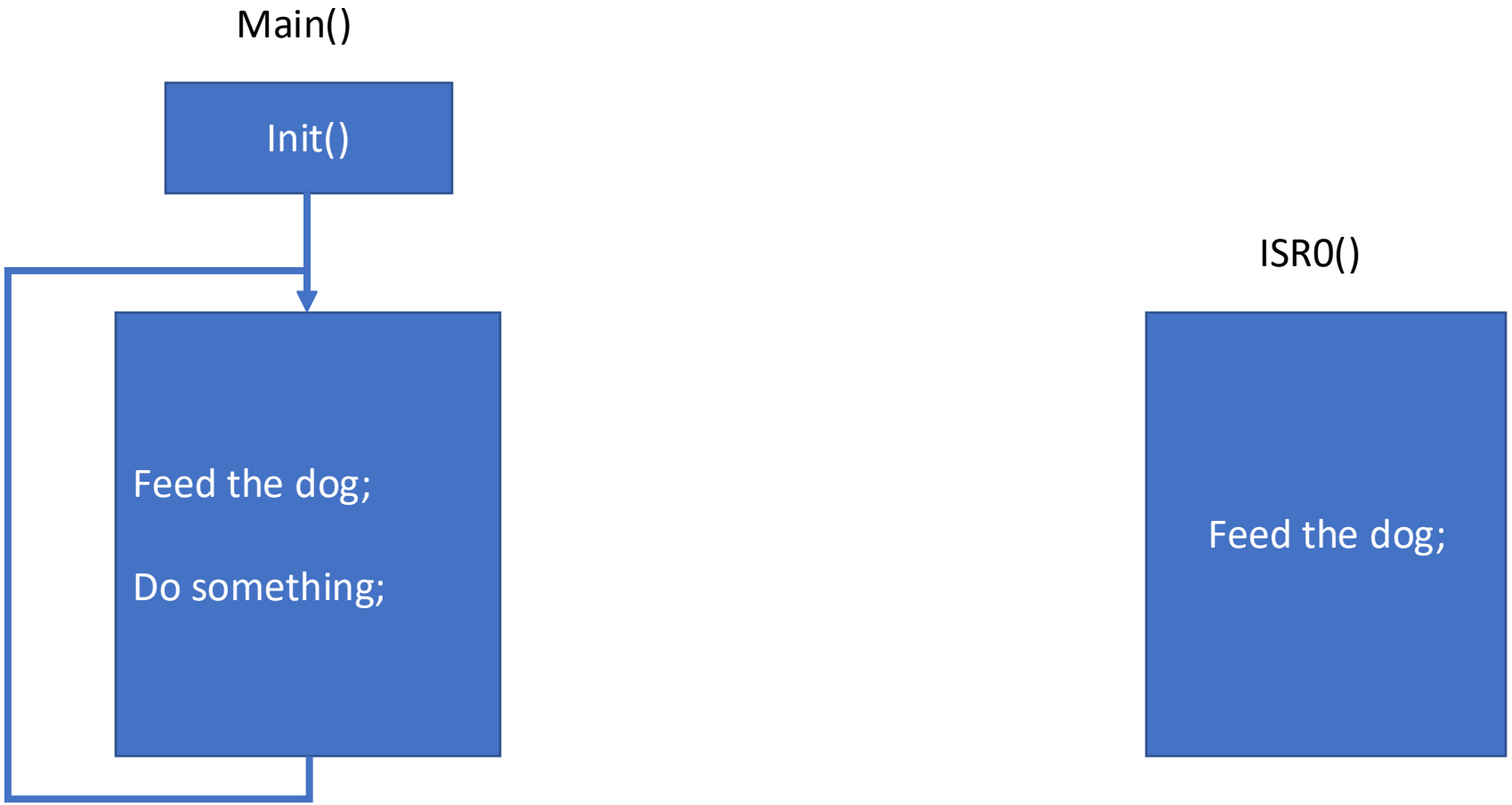
Application: time keeping

- The ESP-IDF comes with some handy functions to keep track of time.
- Based on a 150kHz oscillator and the RTC timer, which allows keeping the system time during any resets and sleep modes (but not power reset).
- The API is compatible with the standard POSIX ones used in many desktop OS (like Linux). Examples:
  - **gettimeofday()** retrieve the current date and time
  - **settimeofday()** sets the current data and time
- There is also SNTP time synchronisation, but you need Internet connection for that.

Watchdog

- A watchdog (timer) is a device that has the task of **resetting the processor** if the program seems to have hanged.
- To not reset, the program must "feed" the watchdog regularly to show that it works.
- Most modern controllers contain a watchdog device.

Feed the dog!



How to feed the watchdog

- Security checks can be spread throughout the code.
- The main program checks the flags regularly and if ever flag is OK the watchdog is fed.
- For even better security, the watchdog can be external to the MCU.

Watchdog timers in ESP32

- The ESP32 has 3 32-bit watchdog timers:
  - 1 in each of the 2 timer modules (called MWDT) clocked by the APB clock via a 16-bit prescaler.
  - 1 in the RTC module (called RWDT) clocked by RTC slow clock (32 KHz).
- A watchdog timer has **four stages**. Each stage may take one out of three or four actions upon the expiry of a programmed period of time for this stage, unless the watchdog is fed or disabled. The actions are: interrupt, CPU reset, core reset and system reset.

Watchdog in ESP-IDF

- The ESP-IDF has support for two types of watchdogs: The Interrupt Watchdog Timer and the Task Watchdog Timer (TWDT).
  - The **Interrupt Watchdog** is responsible for detecting instances where *FreeRTOS task switching* is blocked for a prolonged period of time. A blocked task switching interrupt can happen because a program runs into an infinite loop with interrupts disabled or hangs in an interrupt. The interrupt watchdog is built around the hardware watchdog in timer group 1.
  - The **TWDT** is responsible for detecting *instances of tasks* running without yielding for a prolonged period. Each watched task must 'reset' the TWDT periodically to indicate that they have been allocated CPU time. The TWDT is built around the Hardware Watchdog Timer in Timer Group 0.

Live coding

- We’ll skip it!
  - We need to learn about RTOS to be able to understand it.

Example exam questions

- Describe the difference between polling and using interrupts when reacting to events.
- What is the difference between hardware and software interrupts?
- Describe what happens when an interrupts occurs and an ISR is called.
- What is the “volatile” keyword used for in C?
- Why does “bouncing” of a pushbutton occur?
- Describe the two solutions to avoid the bouncing of a pushbutton.
- What is a “race condition” and how can I avoid it during the execution of an ISR?
- What is a prescaler and what is it used for?
- I have a master clock of 16Mhz, a prescale register of 10 bits and a compare register of 8 bits. What is the smallest frequency this setup can output without using additional code?
- What registers are typically used in a microcontroller to deal with timers?
- Describe the role of a watchdog timer.
- You are programming an MCU for a toy that flashes an LED following a given pattern. The user can press a button to switch to select the next pattern in a list of 10. Provide a pseudo-code example (no need to suppose we are using the ESP-IDF) of how your program would look like given a function *int nextDelay(int pattern)* that gives you the delay you need to wait to toggle the light for a given pattern. You will need to, in parallel, make the LED flash and react to the button when it’s pressed.