Blink Like A Pro

Getting the most out of your microcontroller

About $Presenter

Main interest: (Embedded) Systems software that fails safely

… or can not fail

I work at MEN/duagon: rugged Embedded Hardware up to SIL-4 certification

… focus on software functional safety

This job: implement trivial stuff
suddenly, stuff becomes overly complicated

Example: switch a relais, check if it really switched

About duagon/MEN

- Rugged computers for trains, ships, mining, network communication & more
- Vibration, temperature, dust/gas/fluid, humidity, EMI, radiation

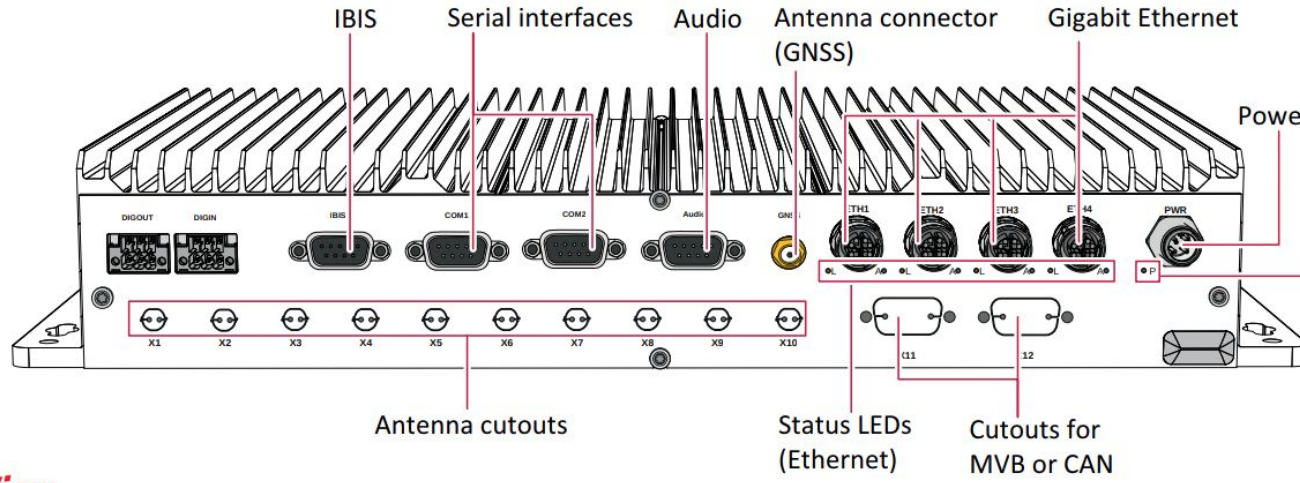
- Applications:
  - Real-time rail inspection at 200 Km/h (hi-res 3D model of rail)
  - Mobile entertainment, video streaming, on-board WiFi
  - Predictive maintenance
  - …

About duagon/MEN

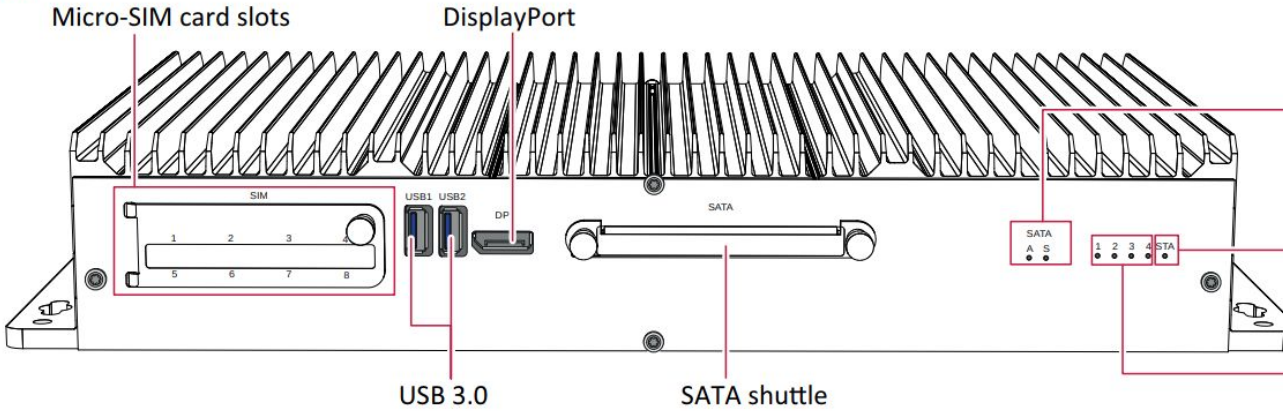
- Circuit design
- PCB design
- Manufacturing in-house
  - Soldering, assembly, testing
- System Design
  - Modules: Power supply, CPU card, storage units, communication
  - Ethernet, GPS, CAN, GPIO, LTE, WiFi, …
- Software (drivers(linux, windows, QNX, VxWorks, OS/2, solaris), middleware, BMC)
  - No application software!

Frontend Development at $Company

Front View



Rear View



Topic Today: Embedded Software!

Last time:
Embedded hardware, dev boards, circuits and debuggers

This time:
Software to run on our embedded targets!

Goals & Non-Goals

Goals:

- Go on a **Quick Safari** of microcontroller peripherals
- Gain more understanding of embedded software design principles
- Modern methods and tools for software development in ‘embedded’ targets

Not Goals:

- Teaching Assembler, C, C++, Linkers, Datasheets reading, …
- Make you an Embedded Software Developer in 45 minutes
  (we’ll do that another time)

Our goal: Make a LED blink!

- One bit of the real world
- Simplest tangible example for many peripherals

- 5 ways to blink an LED (from simple to powerful)

Basic Blocking Blinky: The ‘Hello World’

```
while (true) {
    digitalWrite(pin, ON);
    delay(500);
    digitalWrite(pin, OFF);
    delay(500);
}
```

Demo - Bare Minimum ARM Blinky

- Before we get to the nice stuff, we have to dive deep.

Demo - Bare Minimum ARM Blinky

- Make, Linker, Compiler, Debugger need to cooperate
- Code looks almost like assembler - no abstraction

Demo - Bare Minimum ARM Blinky

It does not have to be like that!

Demo - Blinky with ST HAL and CubeMX

Demo - Blinky with ST HAL and CubeMX

- In the end, the HAL is just register accesses.
- It just offers a (somewhat) more convenient API

Registers?

In General: Memory-Mapped Peripherals

To set a bit in memory, you might use:

```
uint32_t *registerPointer = (uint32_t*) 0xf00dbeef;
*registerPointer |= 1 << 5;
```

But what looks like memory might actually be an LED or a serial port!

All peripherals (LED, Communication, ADC, Timers, CRC, …)
are accessible via Memory Mapped I/O

Demo - Blinky without a single line of code

Example: ADC control via registers

Calibration

```
ADC1->CR2 |= (1 << 3);          // reset calibration
while (ADC1->CR2 & (1 << 3)); // wait until reset finished

ADC1->CR2 |= (1 << 2);          // start calibration
while (ADC1->CR2 & (1 << 2)); // wait until calibration finished
```

Sampling

```
ADC1->CR2 |= 1 << 22;           // start SW conversion

while (ADC1->SR & (1 << 1));   // conversion finished?
value = ADC1->DR & 0x0FFF;     // read converted value
```

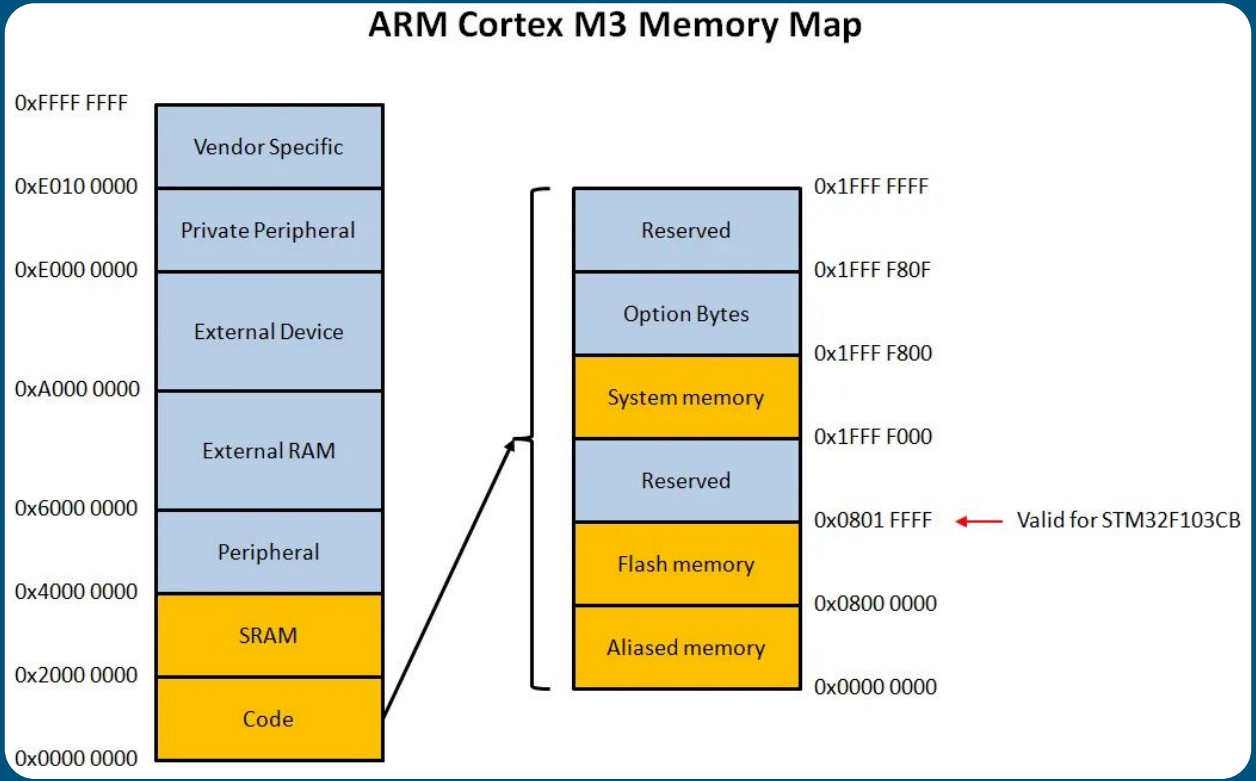
Registers: “Arbitrary” set of Contracts!

- Defined by cryptic names in datasheet
  - PDKEYR, PCROP1ENR, DHR8RD, RSTE1R, etc.

- Setting a bit in a register might brick your controller … or set your rocket on fire

  => everything is **globally mutable shared** state! 😱

STM32 Memory Map



STM32 Memory Map

Bus	Boundary address	Size (bytes)	Peripheral
AHB3	0x5000 0400 - 0x5000 07FF	1 K	ADC3 - ADC4
	0x5000 0000 - 0x5000 03FF	1 K	ADC1 - ADC2
	0x4800 1800 - 0x4FFF FFFF	~132 M	Reserved
AHB2	0x4800 1400 - 0x4800 17FF	1 K	GPIOF
	0x4800 1000 - 0x4800 13FF	1 K	GPIOE
	0x4800 0C00 - 0x4800 0FFF	1 K	GPIOD
	0x4800 0800 - 0x4800 0BFF	1 K	GPIOC
	0x4800 0400 - 0x4800 07FF	1 K	GPIOB
	0x4800 0000 - 0x4800 03FF	1 K	GPIOA
	0x4002 4400 - 0x47FF FFFF	~128 M	Reserved

Software asynchronicity considered harmful

`delay(int ms)` is not good! (why?)

What does `delay(ms)` do?

```
void delay(int count) {
    count *= INSTR_PER_MS;
    while(count > 0) {
        count--;
    }
}
```

We use the processor to … count? This seems wasteful.

We can do MUCH better than NOPping!

async yield style

`millis()`: milliseconds since system start.
What happens on overflow?

This is manually implemented collaborative multitasking!

```
int last = millis();
loop {
    int now = millis();
    if now - last > THRESHOLD {
        do_thing();
        last = now;
    }
    // free to do anything else
}
```

Peripheral: Interrupt

- Real world is asynchronous and parallel.
  - => Systems need to react to async events!
- Triggers: Event out- or inside uC
  - ADC done, rising edge on pin, division by zero, bus fault, …
- Interrupt Handler:
  - preempts regular control flow
  - does something
  - resumes regular control flow

Peripheral: Interrupt

- Handled by NVIC: Nested Vectored Interrupt Controller
  - Just another memory mapped peripheral.
  - Monitors interrupt sources
  - Remembers interrupt requests
- Interrupt priorities/preemption levels
  - Interrupts cannot back-preempt
  - Preemption level determines which request is handled first

Peripheral: Interrupt

- Shared data (between interrupts and main loop)
  - Potential for bad data races
  - You can get away with turning interrupts off while reading (but this reduces responsiveness)
- The `volatile` qualifier is very important
  - “Dear compiler, this value may change in ways you cannot see. Please do not optimize it at all.”

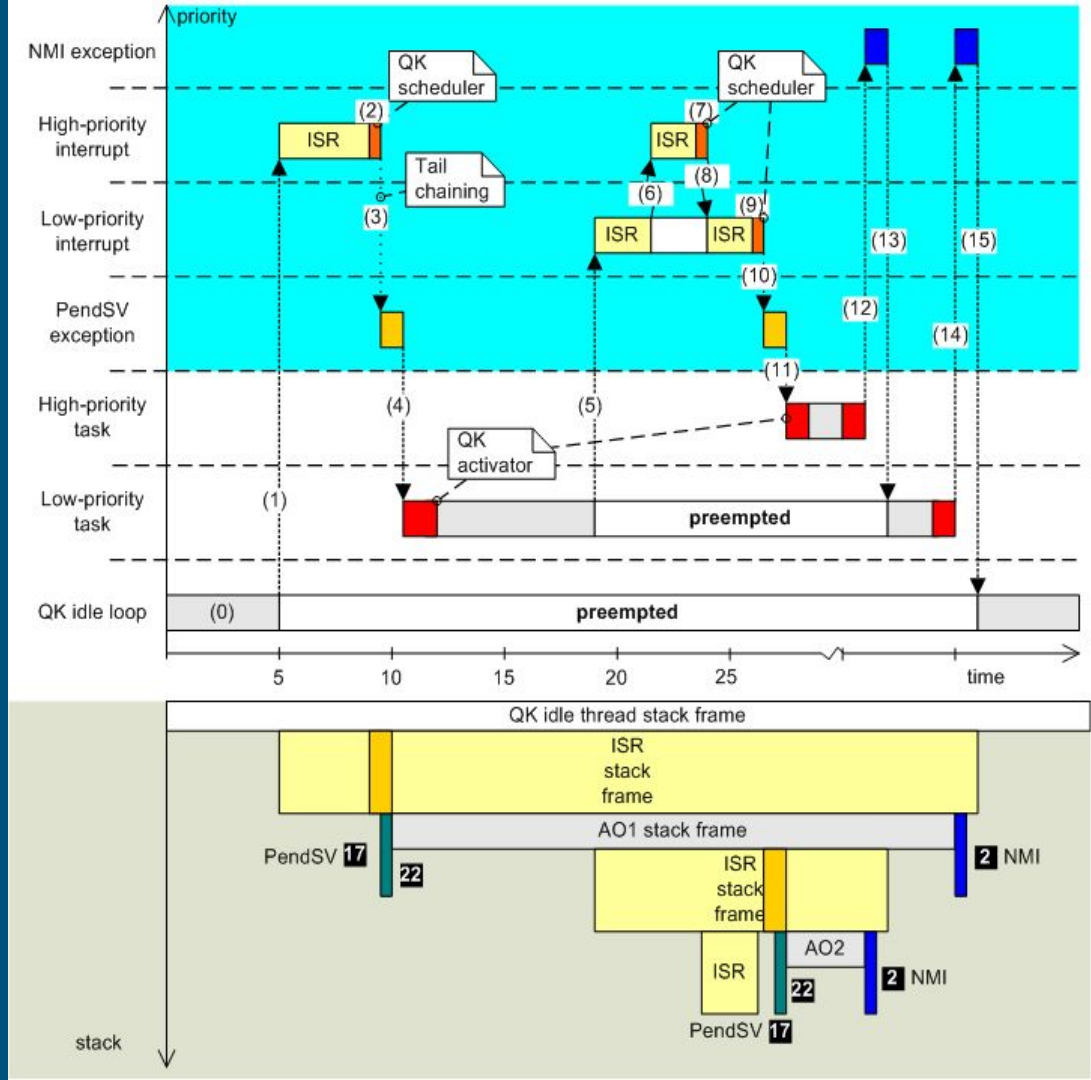
Why volatile?

```
global bool flag = false;

main() {
  registerInterrupt(dataReady, PIN_12_RISING);
  while(1) {
    if (flag) {
      result = readSensor();
    }
}

void dataReady() { // never called?
  flag = true;
}
```

Why volatile?

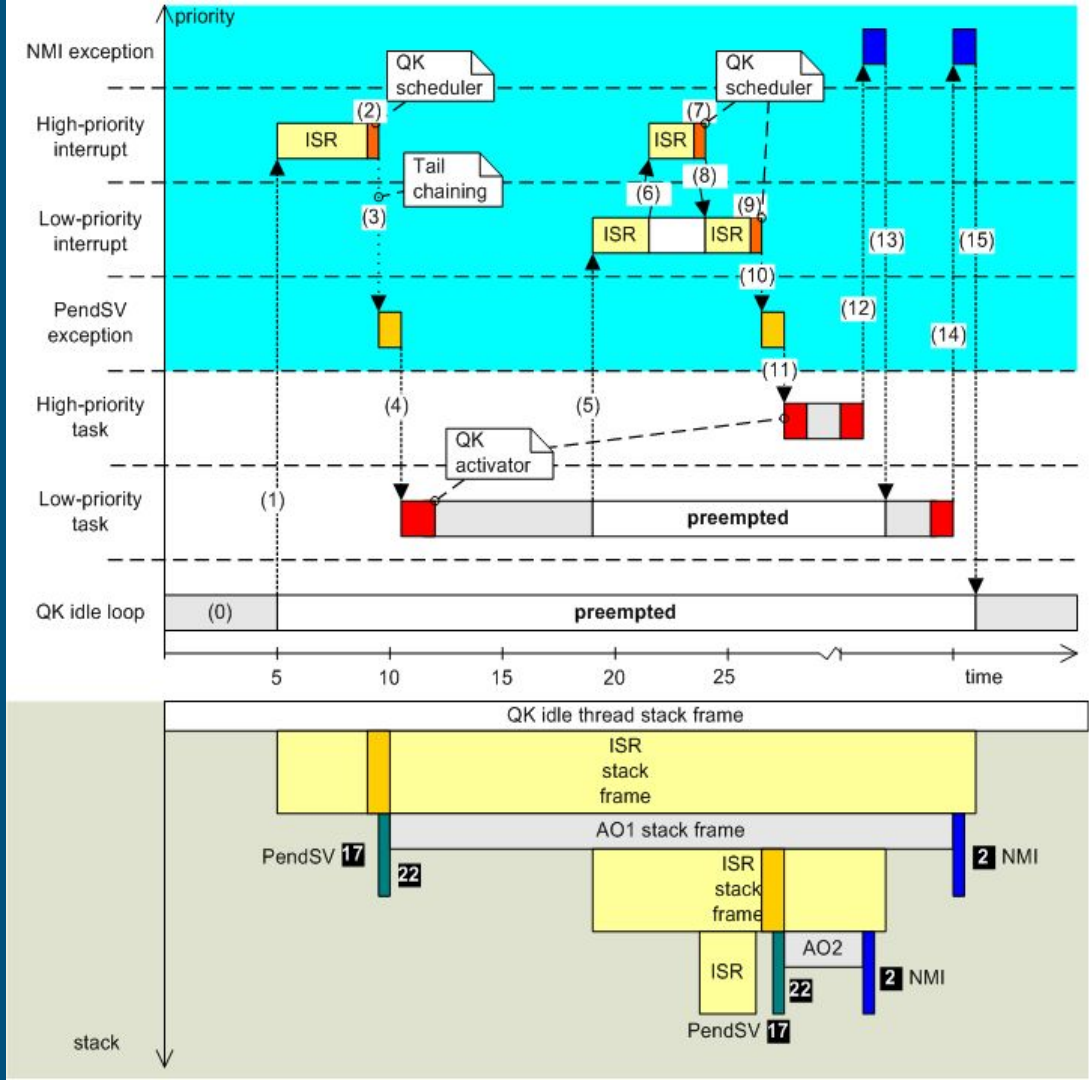
```
global volatile bool flag = false;

main() {
  registerInterrupt(dataReady, PIN_12_RISING);
  while(1) {
    if (flag) {
      result = readSensor();
    }
}

void dataReady() {
  flag = true;
}
```

Interrupt Priorities



Interrupt Priorities

Let’s not get into it …



Peripheral: Timer

- Timer: Counts up to n at given rate, then performs action:
  - Trigger: Interrupt, ADC conversion, slave timer, RTOS task, …
- Timer applications
  - PWM, periodic tasks, duty cycle measurement

- Timer is completely parallel to CPU!

Important Timer Parameters

- Period Value
  - Value up to which to count
- Prescaler Value
  - Skip m clock pulses to slow down counting

$$updateRate[Hz] = \frac{freq_{clk}[Hz]}{(prescaler + 1) * (period + 1)}$$

Important Timer Parameters

- Example: generate a 1-second update rate

$$freq_{clk} = 16Mhz$$

$$updateRate = 1$$

$$updateRate[Hz] = \frac{freq_{clk}[Hz]}{(prescaler + 1) * (period + 1)}$$

$$\frac{updateRate}{freq_{clk}} = \frac{1}{(prescaler + 1) * (period + 1)}$$

$$\frac{freq_{clk}}{updateRate} = (prescaler + 1) * (period + 1)$$

$$16Mhz = (1599 + 1) * (9999 + 1)$$

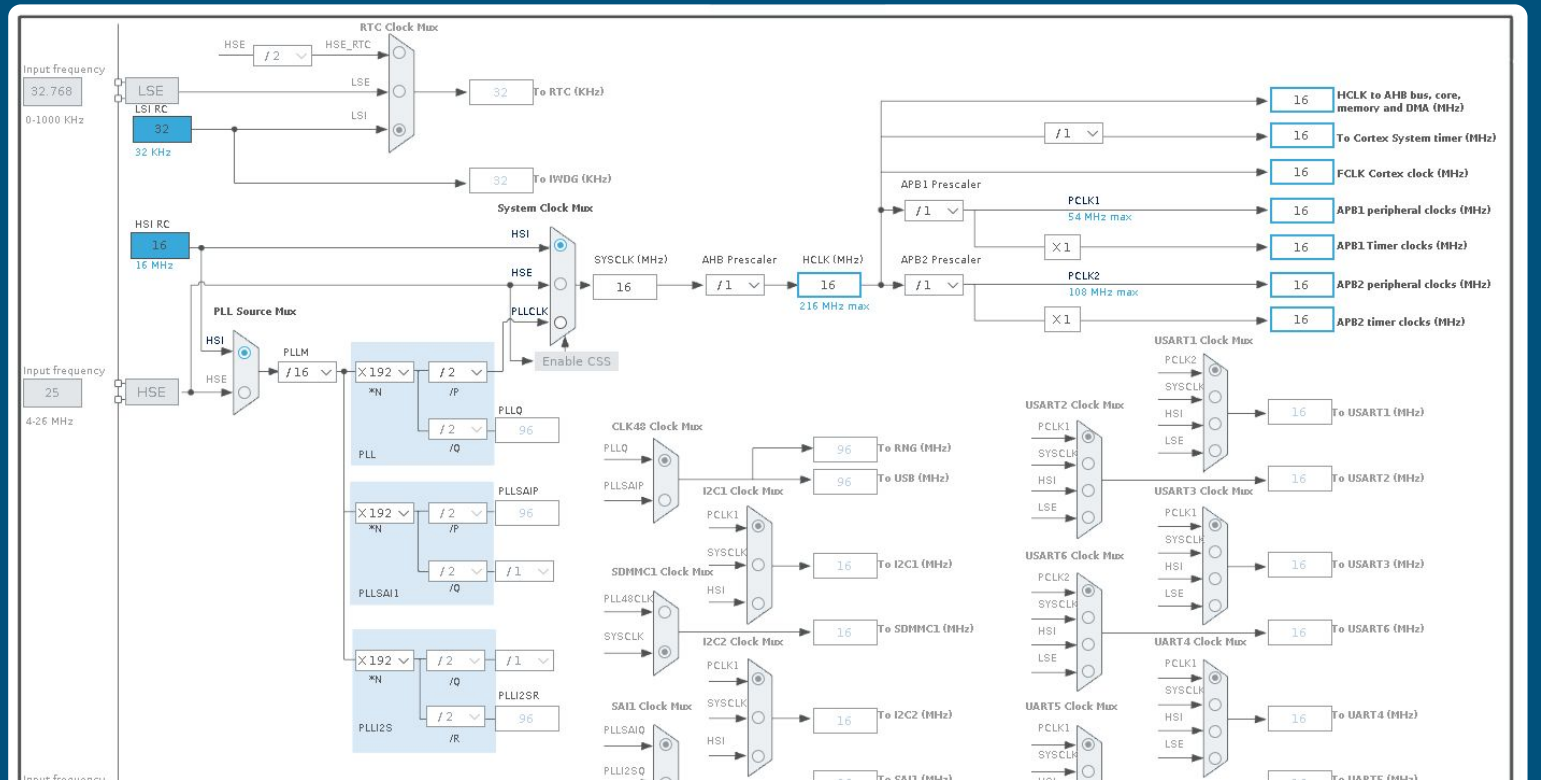
Demo - Blinky with Timer and Interrupt

Demo - Blinky with Timer and Interrupt

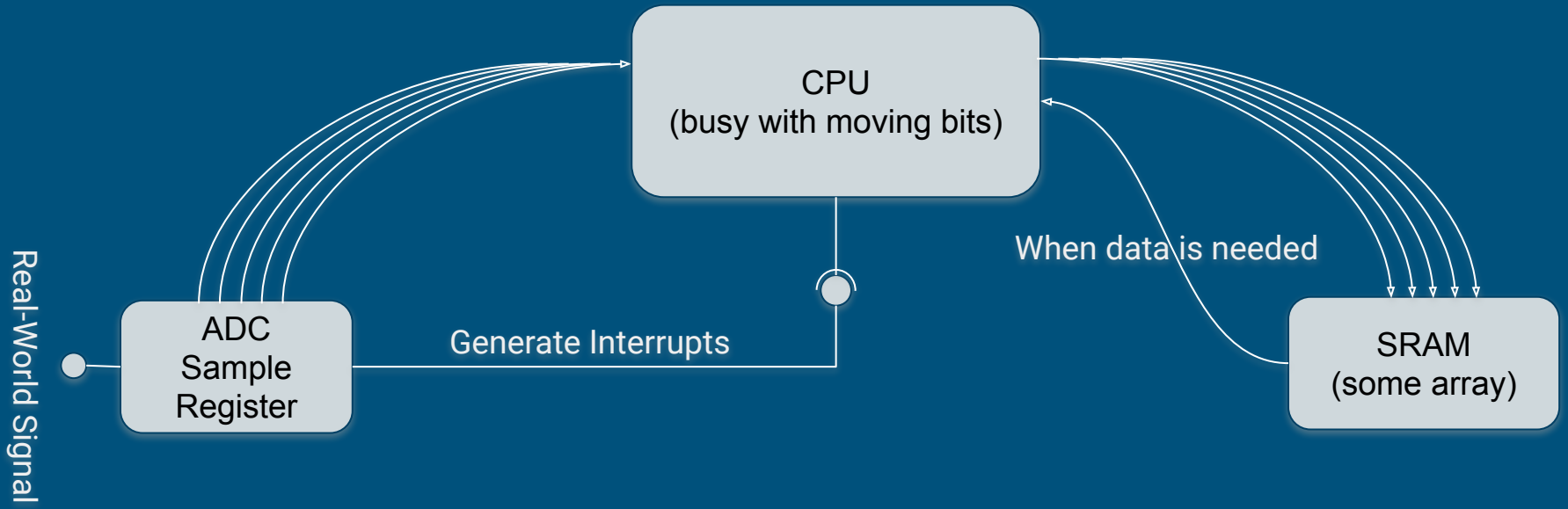
Processor busy moving bits, still.

But only when necessary!

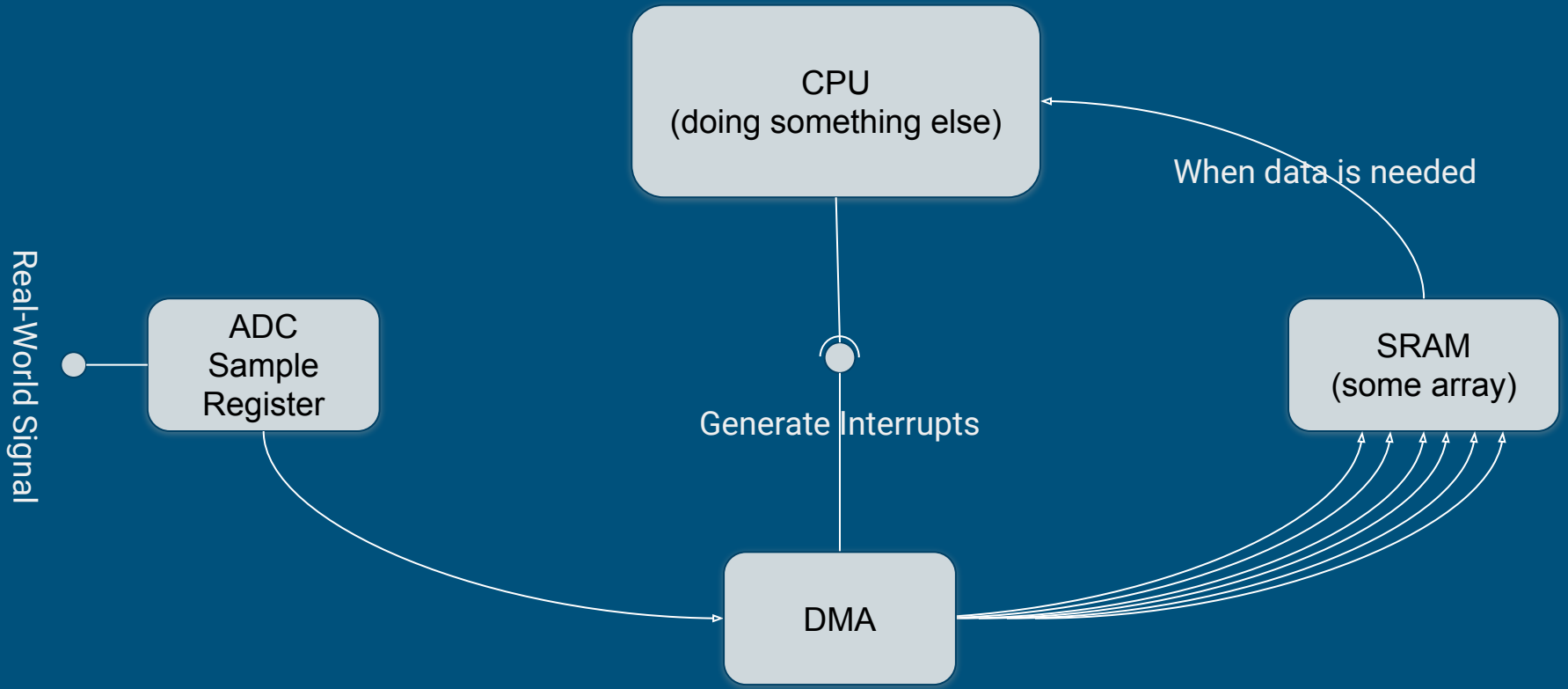
Demo - CubeMX Clock Tree Config



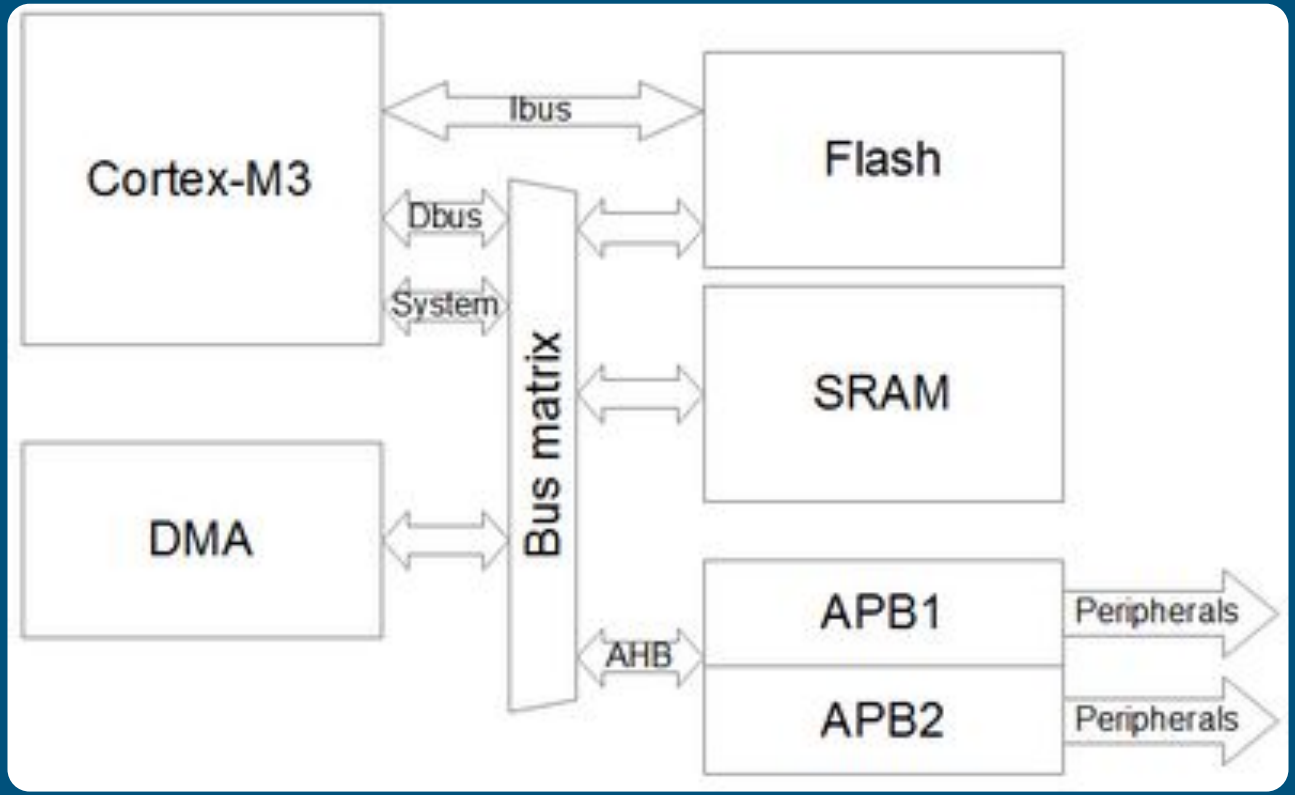
Moving bits manually



DMA - Async Data Pump



DMA - Bus Diagram



DMA Notes

- Sharing XOR Mutation? Builtin data race.
  - Careful to access memory according to DMA callbacks
- `volatile` qualifier necessary?
- Careful for out-of bounds write/read - no C compiler will warn
- Minimize bus arbitration by not using the DMA at full rate
  - Rarely necessary anyway

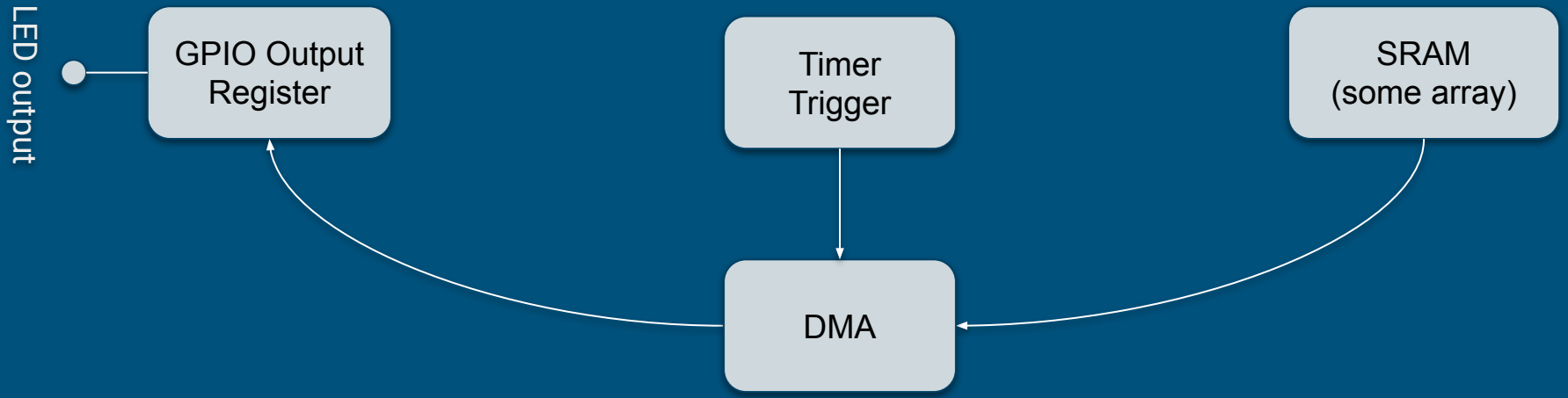
DMA Blinky

- DMA can push bits from memory to peripheral
- So: push bits from bit-buffer to GPIO output register!
- Triggered by timer in regular intervals

Processor is COMPLETELY free!

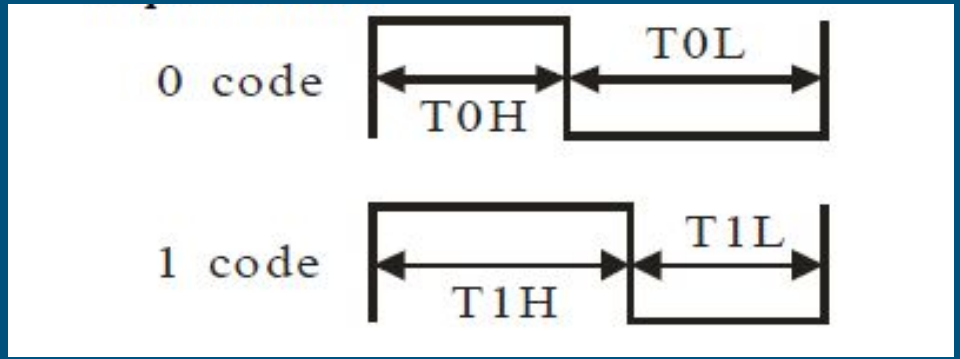
DMA Blinky

CPU
(doing something else)



DMA Example: Hubacek WS2812 driver

- WS2812 LED strips use a serial protocol to control brightness and intensity of each LED - ideal for generation by DMA.
  - generate 0/1 waveforms depending on bit buffer
- 3 DMA actions:
  - ON (always)
  - OFF-0 (conditional)
  - OFF-1 (always)



DMA Example: Hubacek WS2812 driver

- Code, benchmarks and good explanations by Martin Hubacek

https://github.com/hubmartin/WS2812B_STM32F3

DMA Example: Parallel GPIO Port

- Applications:
  - Drive 16 WS2812 strips efficiently
  - Drive parallel port displays
  - Parallel communication
  - Arbitrary pulse generation

https://github.com/barafael/DMA-Parallel-Port

DMA Example: Parallel GPIO Port

- In general: use DMA to automate FIFO copying
  - Receive/Transmit data via SPI
  - Supervise ADC sample values
  - Copy data around (mem2mem mode)

Lets keep overdoing it - RTOS

- Explicit resource sharing (time, cpu, memory, …)
- Enables decoupling of software modules
- USB, TCP/IP, Ethernet, File Systems, …

- Loss of low-level control
- Requires Scheduling and Preemption
- Requires mutex, semaphore, lock, queue, …

Demo - FreeRTOS and CubeMX

Demo - FreeRTOS and CubeMX

Couldn’t be simpler.

But used to be MUCH more difficult!

Shout Out to CubeMX!

- Code generation for peripheral initialization
  - Respecting cross-peripheral constraints
  - Emit code that humans can read
  - Generated code usually quite good

How do they do it!?

Shout Out to CubeMX!

- Correct initialization code for all peripheral combinations
- SIL-2 certified HAL in the works
- Middlewares, uC libraries, Clock Tree
- “System View Description” (SVD) files
  - Provided by platform vendor/implementor
  - Register definitions, read/write modes, valid values

SVD Format: Peripheral Description

```
<peripheral>
  <name>TIMER0</name>
  <description>32 Bit Timer, counting up or down
  </description>
  <baseAddress>0x40010000</baseAddress>
  <access>read-write</access>
```

SVD Format: Peripheral Description

```
<interrupt>
  <name>TIMER0 Interrupt</name>
  <description>Timer 0 interrupt</description>
</interrupt>
```

SVD Format: Peripheral Description

```
<registers>
  <register>
    <name>CR</name>
    <description>Control Register</description>
    <access>read-write</access>
    <resetMask>0x1337F7F</resetMask>
```

SVD Format: Register Description

```
<fields>
  <!-- EN: Enable -->
  <field>
    <name>EN</name>
    <bitRange>[0:1]</bitRange>
    <access>read-write</access>
    <enumeratedValues>
      <enumeratedValue>
        <name>Disable</name>
        <description>Timer is disabled and does not operate</description>
        <value>0</value>
      </enumeratedValue>
      <enumeratedValue>
        <name>Enable</name>
        <description>Timer is enabled and can operate</description>
```

SVD Format: Use cases

- CubeMX
- C or CPP header file generation
- Specification generation for static analysis/formal verification
  - WHY is nobody doing this?!
- HAL generation
- Simulator environment generation
- Peripheral mocking generation
- Probably lots more …

Just generate the HAL!

- Generate type-safe HAL from SVD definitions
  - Register bits can only be accessed (read/write/execute) like specified
  - Guarantee all registers are written or read valid values
  - Higher-level generic HAL can be constructed given the generated HAL

https://github.com/rust-embedded/svd2rust

https://github.com/rust-embedded/embedded-hal

Rust - Embedded HAL

Generated HAL has no runtime cost - this is Rust.

Generic HAL can be mocked or simulated

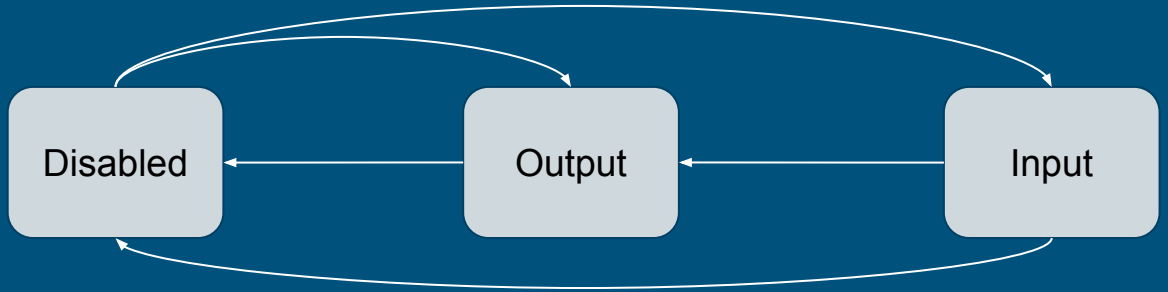
Facilitation of code reuse across families and architectures

Raspberry Pi or Arduino can use the same drivers!

Typestate Programming

- Objective: Encode real-world state in static types.
- Static guarantees for:
  - Only valid state transitions (resource management)
  - Available operations depend on state

GPIO State Transitions



Available Operations

into Output
into Input

Available Operations

into Disabled
read
write

Available Operations

into Disabled
read

Demo - Typestate Programming

Demo - Typestate Programming

Objective:

Ensure correct state transitions.
Everything else is a compile-time type error.

In simple terms: ensure an input pin cannot be written to!

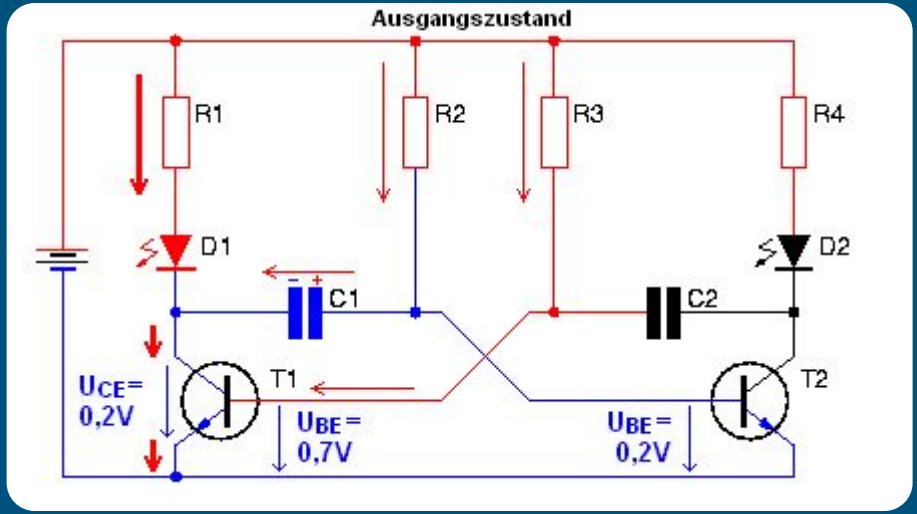
Demo - Typestate Programming

This is functional safety at compile time!

Downsides of Rust

- Almost as challenging as getting C code right
- Mutability, Ownership, Borrowing, Lifetimes, Algebraic Datatyp
- Not a great job market right now
- Ecosystem, libraries, tooling is young
- Compiler can be slow

- Memory Mapping
- Timers
- Interrupts
- DMA
- RTOS
- Software Tools
- Rust



https://upload.wikimedia.org/wikipedia/commons/9/90/Astabile-kippstufe.gif

That’s all I got!
Questions?