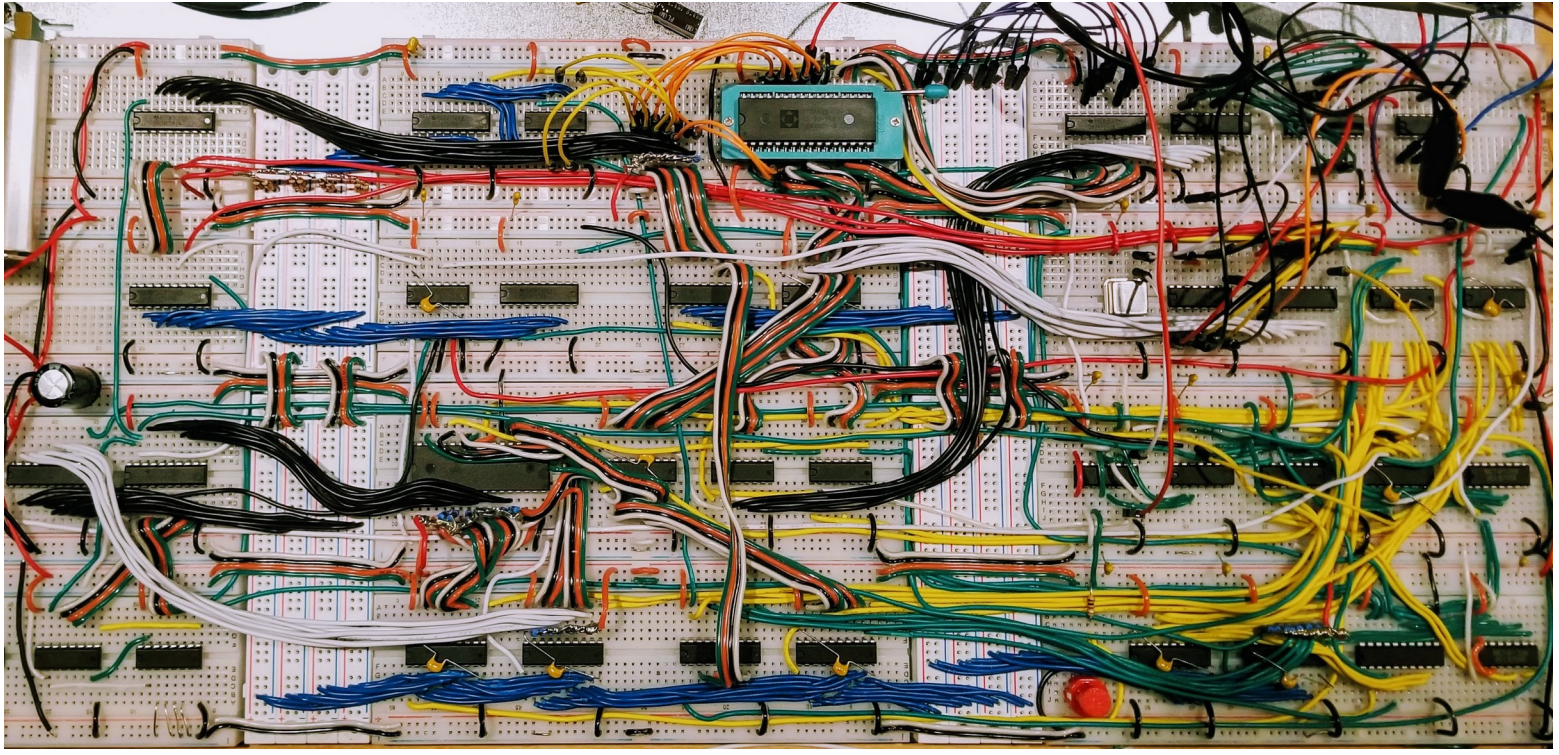
C++ and microcontrollers



Alexis Jeandet <alexis.jeandet@lpp.polytechnique.fr>
jeandet jeandet #irc freenode:jeandet matrix @jeandet:matrix.org

Disclaimer

- I’m not a C++ expert
- This is not about a finished product
- This is more about C++ experiments
- We care about optimizations from -O1 optimization level
- Used to think that C++ wasn’t for microcontrollers

C++ trolls

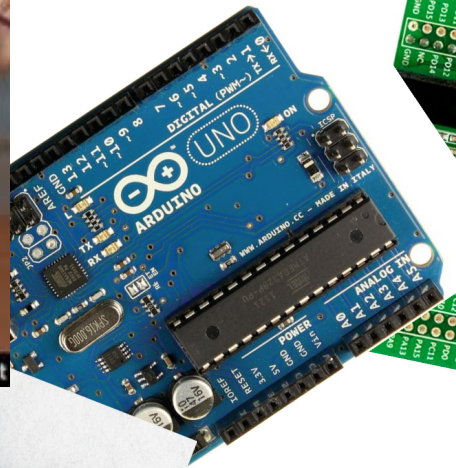
C++ leads to really really bad design choices. You invariably start using the "nice" library features of the language like STL and Boost and other total and utter crap, that may "help" you program, but causes:

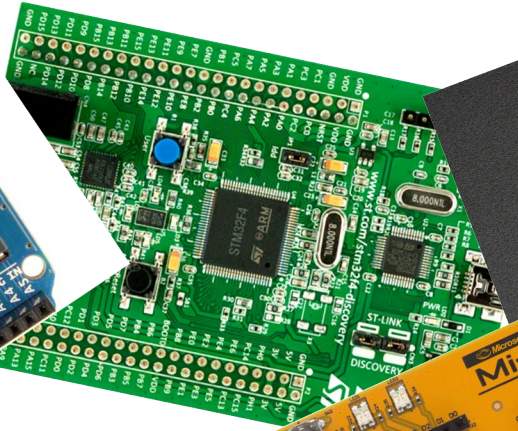
In other words, the only way to do good, efficient, and system-level and portable C++ ends up to limit yourself to all the things that are basically available in C. And limiting your project to C means that people don't screw that up, and also means that you get a lot of programmers that do actually understand low-level issues and don't screw things up with any idiotic "object model" crap.

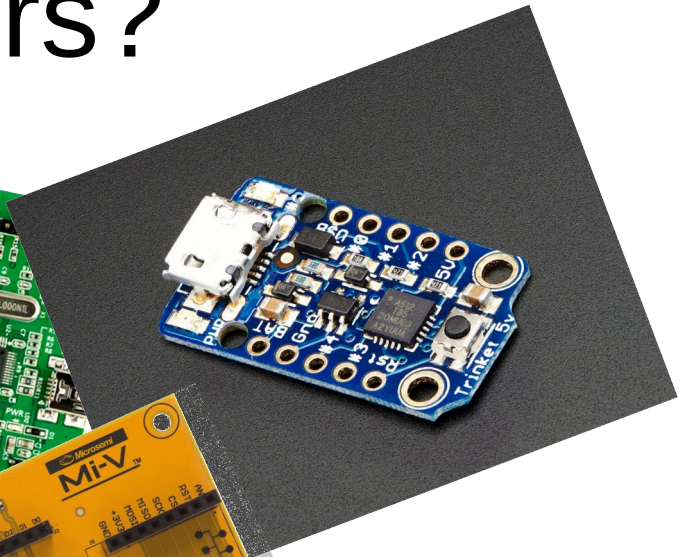
Pros: none

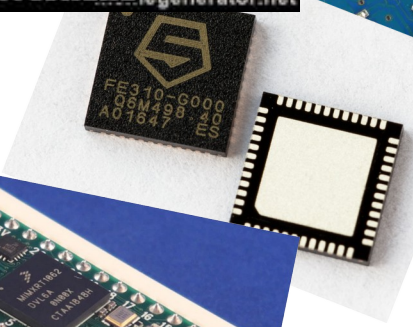
Cons: ill-defined, far too big, Object Oriented Programming, loads of baggage, ecosystem that buys into its crap, enjoyed by bad programmers.

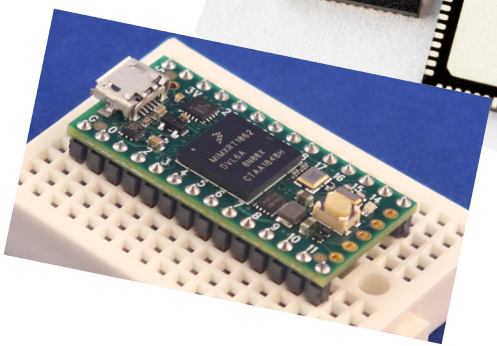
Microcontrollers?

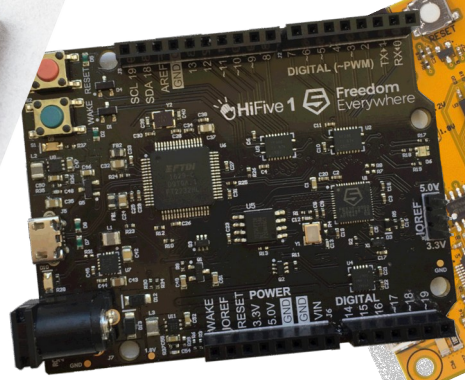


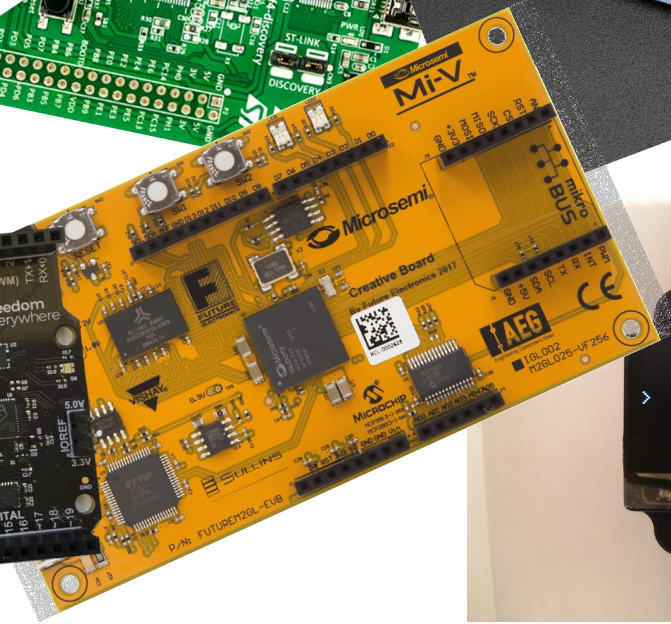




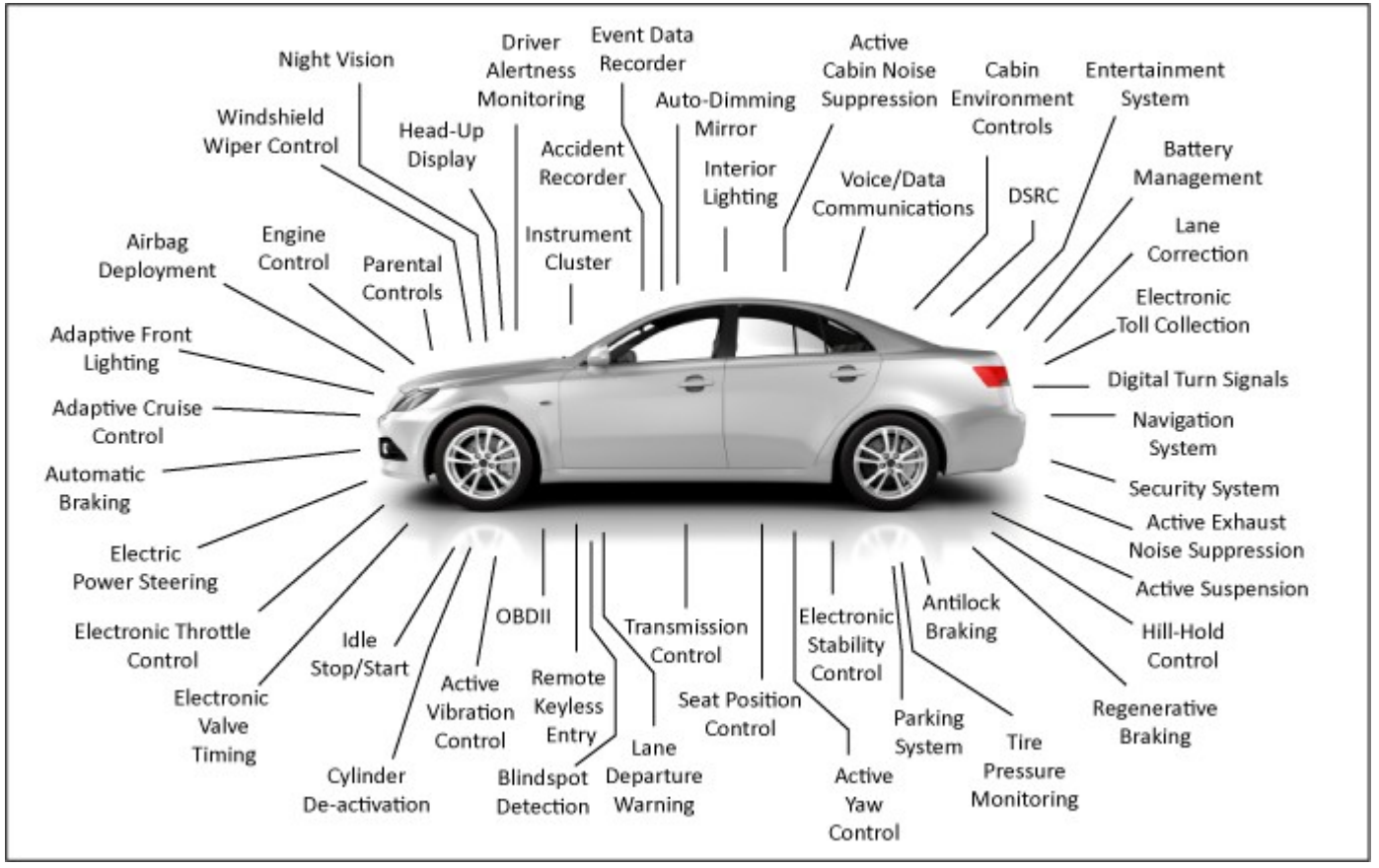








MCU in cars


Night Vision
Driver Alertness Monitoring
Event Data Recorder
Active Cabin Noise Suppression
Cabin Environment Controls
Entertainment System
Windshield Wiper Control
Head-Up Display
Auto-Dimming Mirror
Battery Management
Accident Recorder
Interior Lighting
Voice/Data Communications
DSRC
Airbag Deployment
Engine Control
Parental Controls
Instrument Cluster
Lane Correction
Electronic Toll Collection
Adaptive Front Lighting
Digital Turn Signals
Adaptive Cruise Control
Navigation System
Automatic Braking
Security System
Active Exhaust Noise Suppression
Electric Power Steering
Active Suspension
Electronic Throttle Control
OBDII
Transmission Control
Electronic Stability Control
Antilock Braking
Hill-Hold Control
Idle Stop/Start
Electronic Valve Timing
Active Vibration Control
Remote Keyless Entry
Seat Position Control
Parking System
Regenerative Braking
Cylinder De-activation
Blindspot Detection
Lane Departure Warning
Active Yaw Control
Tire Pressure Monitoring

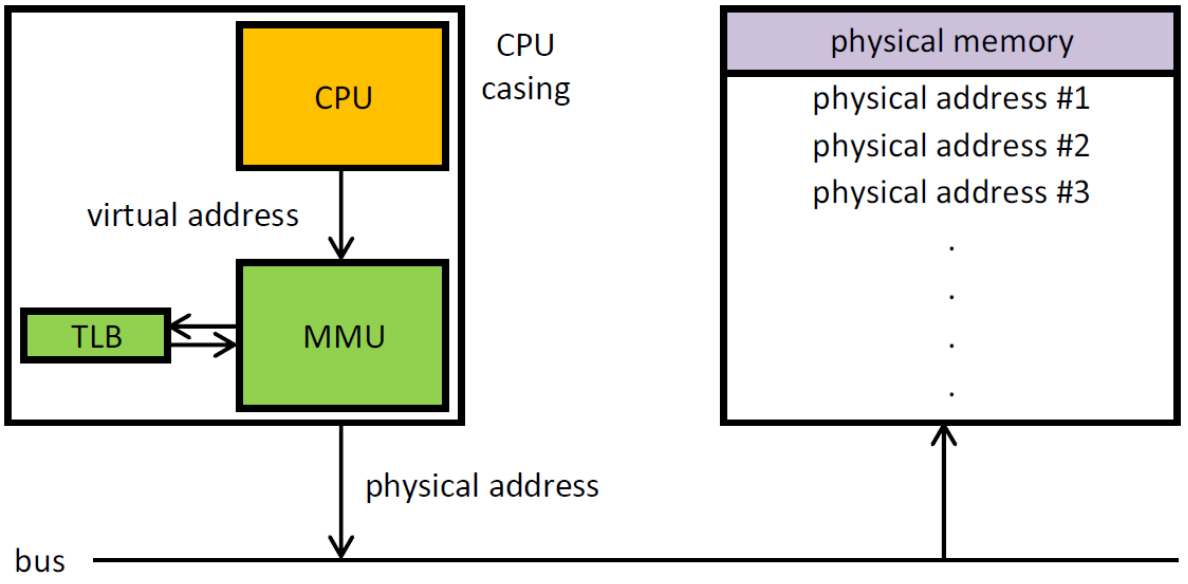
Microcontrollers?

- A microcontroller (MCU for microcontroller unit) is a small computer on a single metal-oxide-semiconductor (MOS) integrated circuit chip. In modern terminology, it is similar to, but less sophisticated than, a system on a chip (SoC); an SoC may include a microcontroller as one of its components. A microcontroller contains one or more CPUs (processor cores) along with memory and programmable input/output peripherals. Program memory in the form of ferroelectric RAM, NOR flash or OTP ROM is also often included on chip, as well as a small amount of RAM. Microcontrollers are designed for embedded applications, in contrast to the microprocessors used in personal computers or other general purpose applications consisting of various discrete chips.

Microcontroller Vs Computer

	MCU	Computer
Power	~1µW to ~1W	~1W to ~1kW
RAM	32B to ~1MB	~32MB to TB
FLASH	512B to ~2MB	GB to TB
Core frequency	~1MHz to ~1GHz	~1GHz to ~4GHz
Cores	1 to N	1 to N
MMU	**NO**	**YES**

Memory Management Unit ?



CPU: Central Processing Unit
MMU: Memory Management Unit
TLB: Translation lookaside buffer

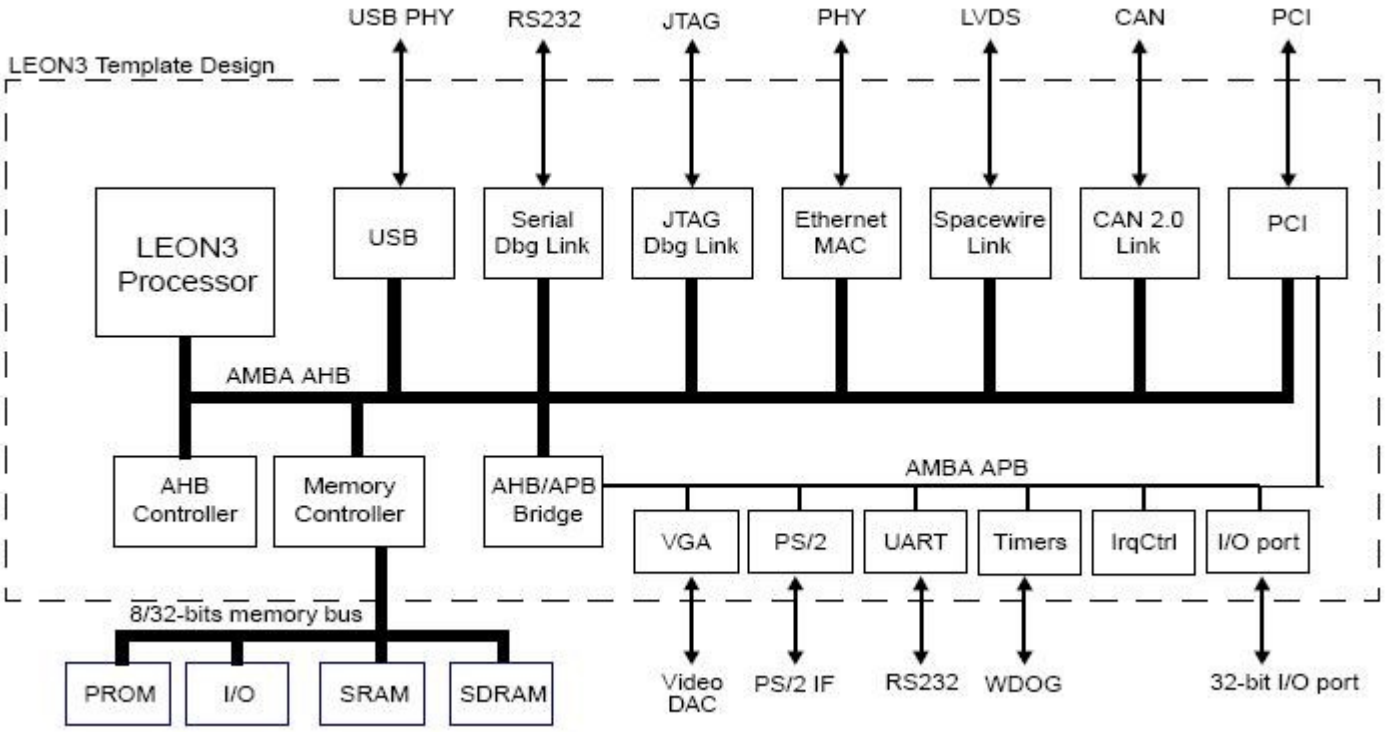
Memory **M**anagement **U**nit ?

- No MMU → no Process (mostly)
- No MMU → heap memory fragmentation
- No MMU → no Segfaults
- No MMU → different operating systems

Programming models

- Operating system
  - Good for complex multiprogramming problems
- Bare metal
  - Faster and simpler for basic control loops

So what does it looks like?



Let’s dive into a datasheet

Let’s consider a simple example

- LED blink, AKA the MCU hello world
- With C and C++
- C++ shouldn’t produce more assembly than C
- Should be more expressive and safer
- There shouldn’t be anything done a run time that can be done at compile time

Raw C version

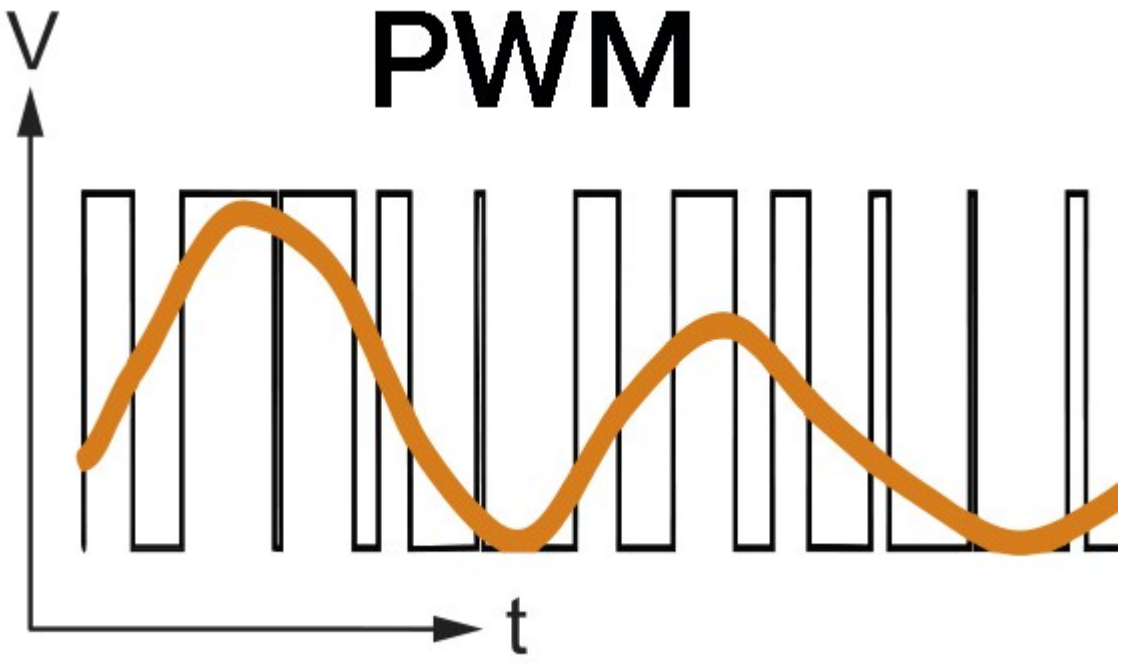
```
int main(void)
{
    // enable GPIOB
    *((volatile uint32_t*)(0x40023830)) |= (1 << 1);
    // configure GPIO PB8 as output
    *((volatile uint32_t*)(0x40020400)) |= (1 << 16);
    for (;;)
    {
        //toggle led
        *((volatile uint32_t*)(0x40020414)) = *((volatile uint32_t*)(0x40020414)) xor (1 << 8);
        for (volatile int i = 0; i < 1024 * 1024 * 2; i++);
    }
}
```

C++ Version

```
int main(void)
{
    rcc::enable_clock(stm32f7, stm32f7.GPIOB);
    set_direction(stm32f7, GPIOB8, stm32::gpio::mode::output);
    for (;;)
    {
        stm32f7.GPIOB.od.get<8>() = !stm32f7.GPIOB.od.get<8>();
        for (volatile int i = 0; i < 1024 * 1024 * 2; i++)
            ;
    }
}
```

Overhead?

Can we make this more interesting?



Can we make this more interesting?

- Let’s make a software PWM led blink
- We want to decouple how we set LED value from PWM code
- We want to decouple how we compute duty cycle from PWM code

Strong types

Let’s step back a little

- We’ll have to deal a lot with peripherals and registers
- Registers have many bit fields
- Registers or bit fields can be read-only
- Embedded systems are deterministic (no hot-plug..)
- Debug and Tests are easier on desktop PC
- Prefer enum classes to literals

Registers

```
uint32_t tmp = *((volatile uint32_t*)(0x40023830));
tmp &= ~(1u << 1);
tmp |= (1u << 1);
*((volatile uint32_t*)(0x40023830)) = tmp;
```

Registers (Structures)

```
typedef struct device_id
{
	uint32_t VID;
	uint32_t PID;
}device_id;
```

```
volatile device_id_t* id = (device_id_t*)0x12345678;
id->VID = 32;
```

- Not convenient with spaced registers
- Bit field access not much better
- User has to “hack” memory layout of structures
- User has to ensure that structure is packed

Registers (simplified)

```
template <typename T, const uint32_t address>
struct reg_t
{
    inline reg_t& operator=(const T& value) noexcept
    {
        *reinterpret_cast<T*>(address) = value;
        return *this;
    }
    constexpr operator T&() noexcept { return *reinterpret_cast<T*>(address); }
    constexpr operator const T&() const noexcept { return *reinterpret_cast<T*>(address); }
};
```

```
reg_t<uint16_t, 0x1FF0F442> flash_size; // declare some register
std::cout << flash_size;                // you can access its value
flash_size = 123;                       // you can set its value
```

Bit field

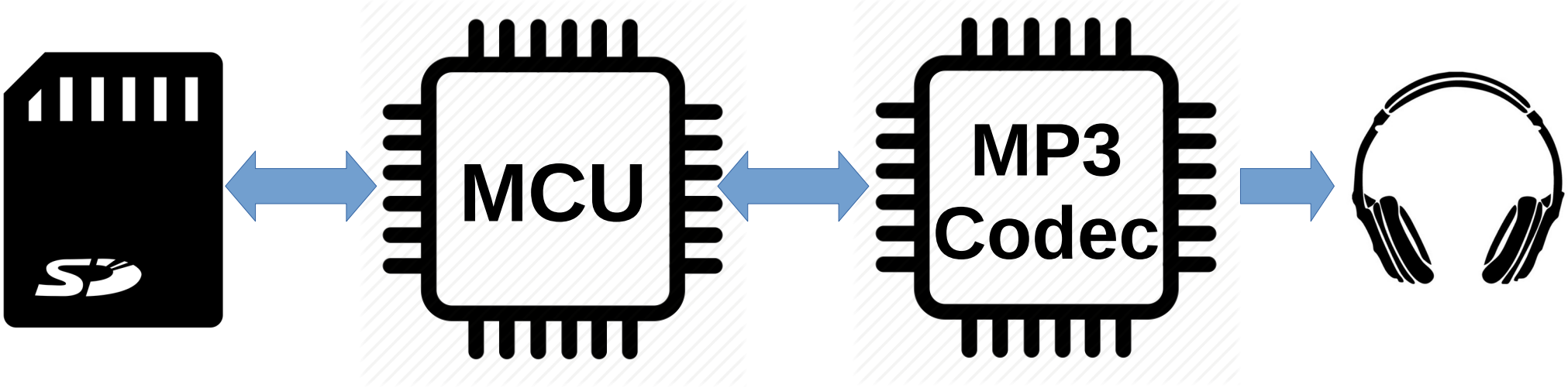
```
template <typename reg_t, int start_index, int stop_index=start_index, typename value_t=int>
struct bitfield_t
{

    constexpr bitfield_t operator=(const value_t& value) const noexcept
    {
        reg_t::value() = (reg_t::value() &  ~mask)  | shift(value);
        return bitfield_t<reg_t,start_index,stop_index,value_t>{};
    }

    constexpr operator value_t() noexcept{
            return value_t((int(reg_t::value())& mask)>>start); }
    constexpr operator value_t() const noexcept {
            return value_t((int(reg_t::value())& mask)>>start); }
};
```

C++ peripherals/register/bit-fields

- Use types instead of values
- Can mix with enum classes
- Generates compile time error on read-only register/field write
- 0 or tiny overhead
- Way more expressive
- Allow function specialization

Now let’s make a basic MP3 Player

Now let’s make a basic MP3 Player



Now let’s make a basic MP3 Player

- No filesystem
- Simple infinite loop
- Made with modern C++
- Strong decoupling between layers
- Use SD bus but should work with SPI

Basic MP3 player

- Clocks, IO, peripheral init
- SD card init
- MP3 codec init
- Main loop
  - Get next data block
  - Give data block to codec

Let’s look the code

Simple MP3 player

- Main loop can easily work on any system
- SD card SW protocol decoupled from HW layer
- Source code looks simple and expressive
- Almost no SW error possible
- We can easily develop/test a filesystem driver on computer

Conclusions

- Writing no-overhead C++ on MCU is possible
- Const const const!!!
- C++ allow modular code with no run-time costs
- C++ can be safer than C
- C++ doesn’t help to solve HW bugs/issues
- C++ isn’t “C with classes”
- C++ templates are “Pay for what you use”
- Templates debug is a big pain
- Computers have std::, microcontrollers’ll have mst::