1. Terms and Conditions :) 
2. C-World Problems 
3. C++ Solutions 
4. Restraints in the Industry 
5. Conclusions and Discussion
Terms and Conditions :)  
Defining ”Embedded”

- interact with physical environment via sensors and actuators
- cost and resource constrained
- talk assumes typical MCUs
  - < 1MB internal program memory
  - AVR, ARMv7M based (e.g. STM32, LPC5), etc.
  - bare-metal or small RTOS
  - or true kernel space programming on larger systems
The Register Access Layer
The Register Access Layer

Example Code for STM32F107
https://github.com/alibabashack/cxx-device-models
C-World Problems
Typical Register-Level Mistakes

- Intention: Set Pin B5 high
- ODR = Output Data Register
- IDR = Input Data Register

```c
# define GPIO_ODR_ODR_5 (( uint32_t )0x00000020)
# define GPIO_IDR_IDR_5 (( uint32_t )0x00000020)
GPIOB -> ODR |= GPIO_IDR_IDR_5;
```
C-World Problems
Typical Register-Level Mistakes

- Intention: Set Pin B5 high
- ODR = Output Data Register
- IDR = Input Data Register

Mask Confusion 1

```c
#define GPIO_ODR_ODR_5 ((uint32_t)0x00000020)
#define GPIO_IDR_IDR_5 ((uint32_t)0x00000020)
#define GPIOB ((GPIO_t *) GPIOB_BASE)

GPIOB->ODR |= GPIO_IDR_IDR_5;
```
C-World Problems
Typical Register-Level Mistakes

Mask Confusion 1

```c
#define GPIO_ODR_ODR_5 ((uint32_t)0x00000020)
#define GPIO_IDR_IDR_5 ((uint32_t)0x00000020)
#define GPIOB ((GPIO_t *) GPIOB_BASE)

GPIOB->ODR |= GPIO_IDR_IDR_5;
```

✓ compiles
✓ works (by accident)
✗ but it is not correct!
✗ And it is non-optimal (on STM32)
Intention: Enable clock for blocks SPI2 and GPIOB

RCC = Reset and clock control

APB = Advanced Peripheral Bus
Intention: Enable clock for blocks SPI2 and GPIOB

RCC = Reset and clock control

APB = Advanced Peripheral Bus

Mask Confusion 2

1. `RCC->APB1ENR |=`
2. `(RCC_APB1ENR_SPI2EN | RCC_APB2ENR_GPIOBEN);`
C-World Problems
Typical Register-Level Mistakes

Mask Confusion 2

1  RCC->APB1ENR |=
2    (RCC_APB1ENR_SPI2EN | RCC_APB2ENR_GPIOBEN);

✓ compiles
✗ but does not work
✗ may introduces side effects on existing behavior
✗ may invoke undefined hardware behavior
   (e.g. unexpected power consumption)
C-World Problems
Resulting Issues

✗ Creates lots of problems for future-Homer!
C-World Problems
Resulting Issues

- time wasted for
  - flashing to target
  - creating stimuli/test fixtures
  - lots of manual work

- remaining hard-to-find non-functional bugs

- undetected usage of undefined behavior
  - catastrophic if the chip die changes
  - YES, this happens!
C-World Problems

Resulting Issues

- time wasted for
  - flashing to target
  - creating stimuli/test fixtures
  - lots of manual work
- remaining hard-to-find non-functional bugs
- undetected usage of undefined behavior
  - catastrophic if the chip die changes
  - YES, this happens!

Our aim:
The compiler shall detect illegal/non-optimal register usage!
GPIO memory map definition

```c
typedef struct
{
    volatile uint32_t CRL;
    volatile uint32_t CRH;
    volatile uint32_t IDR;
    volatile uint32_t ODR;
    //...
} GPIO_TypeDef;
```

- everything is of the same data type (integer)
- thus, we can assign apples to oranges
GPIO Output Data Register bit definition

```c
#define GPIO_ODR_ODR0 ((uint16_t)0x0001)
#define GPIO_ODR_ODR1 ((uint16_t)0x0002)
#define GPIO_ODR_ODR2 ((uint16_t)0x0004)

// ...
```

- bits of the same field are unrelated language-wise
C-World Problems
Reasons for Compiler Blindness

ODR, IDR: different bits for the same logical pin

1. \#define GPIO_ODR_ODR_5 ((uint32_t)0x00000020)
2. \#define GPIO_IDR_IDR_5 ((uint32_t)0x00000020)

- no differentiation between
  - logical and physical structures
  - mechanism and policy\(^1\)

\(^1\)see Linux Device Drivers – Corbet et al.
C-World Problems
Reasons for Compiler Blindness

- mechanism
  - the intention of an action

- policy (Verfahrensweise)
  - how the action is achieved

- no differentiation between
  - logical and physical structures
  - mechanism and policy

- self-documenting code
C-World Problems

Let's Discuss

What can we do to engage the compiler?

- one type per register
- cluster related bits
- separate mechanism from policy
- check invalid value combinations
C-World Problems
The True Challenge

- `enum` and `typedef` are unsafe: implicit conversions
- bit fields also suffer from implicit conversions
- no mechanism to convert types (except casting)
- non-trivial value checks must be performed at runtime
C-World Problems
The True Challenge

- `enum` and `typedef` are unsafe: implicit conversions
- bit fields also suffer from implicit conversions
- no mechanism to convert types (except casting)
- non-trivial value checks must be performed at runtime

Conclusion
The C type system is to weak for our purpose.
- `enum class` (Scoped Enumerations)
  - define types without default operators or conversions
- user-defined operators
  - define exactly the operations you need
- `constexpr` — running code at compile-time
- `static_assert` — checking conditions at compile-time
GPIO registers with individual types

```cpp
struct GpioRegs {
    PortConfigLowReg CRL;
    PortConfigHighReg CRH;
    InputDataReg IDR;
    OutputDataReg ODR;
    SetResetMaskReg BSRR;
    // ...
};
```
class OutputDataReg {

public:

    OutputDataReg& operator=(PinId pid) {
        rawReg = (1<<static_cast<size_t>(pid));
        return *this;
    }

    OutputDataReg& operator=(PinMask pm) {
        rawReg = static_cast<uint32_t>(pm);
        return *this;
    }

private:

    volatile uint32_t rawReg;
};
```cpp
enum class PinId {
    Px0, Px1, Px2, Px3, Px4, Px5, Px6, Px7, Px8,
    Px9, Px10, Px11, Px12, Px13, Px14, Px15,
};

enum class PinMask: uint32_t {
    /// PinMasks are created indirectly from
    PinId using logic operators
};

gpioERegs->ODR = PinId::Px2;
gpioERegs->ODR = (PinId::Px2 | PinId::Px3);
```
constexpr PinMask operator |(PinId l, PinId r) {
    return static_cast<PinMask>(
        (1 << static_cast<size_t>(l)) | 
        (1 << static_cast<size_t>(r))
    );
}

constexpr PinMask operator |(PinId l, PinMask r) {
    return static_cast<PinMask>(
        (1 << static_cast<size_t>(l)) | 
        static_cast<size_t>(r)
    );
}
template <PinId p, PinMode m, PinConfig c>
constexpr PinSetupHigh createPinSetupHigh () {
    static_assert (
        static_cast<size_t>(p) > 7,
        "selected PinId invalid");
    // ...
}

gpioERegs->CRL = createPinSetupLow<
    PinId::Px2,
    PinMode::Output2Mhz,
    PinConfig::GeneralPushPullPullOutput>();
Compiler can throw errors for

✓ mixed-up bit masks
✓ illegal value ranges
✓ suboptimal register access

✓ impossible to use reserved/undefined behavior
✓ zero runtime overhead

✗ Need to invest in a lot of new code
Restraints in the Industry
C as Dominating Language

- 70% of Embedded Software is written in C
- 23% is written in C++
- Industry is highly focused to support C
- everything else is optional
- primary solution: building on top of C

2Barr Group – Embedded Systems Safety & Security Survey 2018
Restraints in the Industry
Building on Top of C

- Peripheral Library with assertions (e.g. ST)
  - assertions only thrown at runtime
  - runtime overhead
- Automatic Code Generation (e.g. ST CubeMx)
  - vendor-dependent tools in build
  - integration of automatic and manual code
  - support stops at the chip boundary
Conclusions and Discussion

- **C++**
  - is also suitable for lowest code levels
  - can do powerful things without runtime overhead
  - can offer advanced diagnostics at compile time

- The industry will likely not support this approach.
- Let's make it free software then!
Conclusions and Discussion

Q/A
Conclusions and Discussion

- Q/A
- possible future talks:
  - Test-Driven Development for Embedded Systems
  - Using Smart Pointers with Memory Pools
  - Device Driver Architecture