# Modern special function register abstraction

Beating hand coded C in bare metal designs.

By Odin Holmes

Immer eine zündende Idee.

Contact: holmes@auto-intern.de

# Bare Metal is bug prone!

- Lots of bit manipulations
- No zero cost unit testing*
- No zero cost libraries*
- Single processor *race conditions* because of ISRs

AUTOINTERN
Immer eine zündende Idee.

*there are some tools, but not generic and not zero cost

# Special Function Register

- Looks like RAM to the Compiler
  - Observable!
- Often contain multiple bit fields and reserved bits
- Odd behavior
  - set to clear / clear on read
  - FIFO buffer
  - bit banding / BME (Bit Manipulation Engine)

```
template<int Address, int Mask,typename TType=int /*other rules */>
struct FieldLocation {
    using DataType = TType;
};
constexpr FieldLocation<0x100, (1 << 4)> myBit{};

template<typename TLocation, typename TLocation::DataType Value>
struct FieldValue {};

//turn a field location into an action which sets the corresponding bits
template<typename T>
constexpr Action<T, WriteTag, 1> set(T) { return{}; }
// other factory functions like write, read, clear etc.

template<typename... T>
constexpr void apply(T...) { /*template magic*/ }

//use case example
apply(set(myBit));
```

# constexpr meta functions

```
namespace Uart1 {
    static constexpr FieldLocation<0x1234, (1 << 3)> cfgEnable{};
    enum class cfgStopBVal { one, two };
    static constexpr
        FieldLocation<0x1234, (1 << 3), cfgStopBVal> cfgStopB{};
    namespace cfgStopBValC {
        static constexpr
            FieldValue<decltype(cfgStopB), cfgStopBVal::one> one{};
        static constexpr
            FieldValue<decltype(cfgStopB), cfgStopBVal::two> two{};
    }
    enum class CfgDataBVal { eight, nine };
    static constexpr FieldLocation<0x1234, (1<<5), CfgDataBVal> cfgDataB{};
    namespace cfgDataBValC {
        static constexpr
            FieldValue<decltype(cfgDataB), CfgDataBVal::eight> eight{};
        static constexpr
            FieldValue<decltype(cfgDataB), CfgDataBVal::nine> nine{};
    }
}
```

**Library code**

```
apply(set(Uart1::cfgEnable));
apply(write(Uart1::cfgStopBValC::one,
        Uart1::cfgDataBValC::nine));

if (apply(read(cfgEnable)) == true) {}

auto temp = apply(read(Uart1::cfgStopB, Uart1::cfgDataB));
if (temp == Uart1::cfgDataBValC::nine) {}
else if (temp == Uart1::cfgStopBValC::one) {}

apply(atomic(set(Uart1::cfgEnable)));
```

**User code**

# Atomic tricks

- Any chip with BME
- Bit banding for all single bit manipulations within the bit banding address range
- If all bits within a 8, 16 or 32 bit range are either written to or have a default (reserved, set to clear etc.)
- Make use of `ldrex` and `strex`
- If all of them fail we can still turn off interrupts

# Efficiency tricks

- load multiple / store multiple
- Localize adressing
- Use of BME
- Use of bit banding
- Work on narrower data
- Turn read modify write into blind writes
  - Reserved bit awareness!
- Merge all initialization sequences

# If you'd like to know more


Contact me on GitHub
@porkybrain

write me an email
holmes@auto-intern.de

or join me at lunch today!

AUTO INTERN
Immer eine zündende Idee.