C++ Hardware Registers

Using compile time techniques to provide guarantees without sacrificing performance

Ken Smith

kgsmith@gmail.com

Overview

- State of the art in MMIO
- A C++ register model
- Policies
- Performance and benefits

The state of the art: macros

```
#define REG_VAL_TYPE (volatile uint32_t)
#define REG_TYPE (REG_VAL_TYPE*)
#define DEV_REG ((REG_TYPE)0xffff0000)
```

The state of the art: macros

```
#define REG_VAL_TYPE (volatile uint32_t)
#define REG_TYPE (REG_VAL_TYPE*)
#define DEV_REG ((REG_TYPE)0xffff0000)

*DEV_REG |= 1;
REG_VAL_TYPE get_val = *DEV_REG;
```

The state of the art: macros

- Advantages
  - Lots of people are comfortable with the idiom
  - Generates efficient code

The state of the art: macros

- Advantages
  - Lots of people are comfortable with the idiom
  - Generates efficient code
- Disadvantages
  - Easy to get it wrong
    - Eg., writing to a read-only register compiles and runs
  - Meaning can be opaque

The state of the art: bitfields

```
struct   periph_status_reg_t
{
     volatile uint32_t enabled : 1;
     volatile uint32_t flag : 1;
     volatile uint32_t bitrate : 6;
};
```

The state of the art: bitfields

```
struct  periph_status_reg_t
{
     volatile uint32_t enabled : 1;
     volatile uint32_t flag : 1;
     volatile uint32_t bitrate : 6;
};

struct periph_status_reg_t* periph_status_reg =
     (struct periph_status_reg_t*) 0xffff0000;
```

The state of the art: bitfields

```
struct  periph_status_reg_t
{
     volatile uint32_t enabled : 1;
     volatile uint32_t flag : 1;
     volatile uint32_t bitrate : 6;
};

struct periph_status_reg_t* periph_status_reg =
     (struct periph_status_reg_t*) 0xffff0000;

if (periph_status_reg->enabled)
{
     // react
}
```

The state of the art: bitfields

- Advantages
  - More readable than masking / anding / oring to access values
  - Also generates efficient code

The state of the art: bitfields

- Advantages
  - More readable than masking / anding / oring to access values
  - Also generates efficient code
- Disadvantages
  - Not necessarily portable
  - You can still accidentally write to a read-only register, etc.

C++ to the rescue

- Catch common errors at compile time
- Preserve the readability of bitfields
- Preserve the platform independence of masking / anding / oring with macros

A register model

```
template
<
    uint32_t address,
    uint32_t mask,
    uint32_t offset,
    class mutability_policy
>
```

A register model

```
template
<
    uint32_t address,
    uint32_t mask,
    uint32_t offset,
    class mutability_policy
>
struct reg_t
{
    static void write(uint32_t value)
    {
        mutability_policy::write(
            reinterpret_cast<volatile uint32_t*>(address),
            mask, offset, value
        );
    }
```

A register model

```
template
<
    uint32_t address,
    uint32_t mask,
    uint32_t offset,
    class mutability_policy
>
struct reg_t
{
    static void write(uint32_t value)
    {
        mutability_policy::write(
            reinterpret_cast<volatile uint32_t*>(address),
            mask, offset, value
        );
    }

    static uint32_t read()
    {
        return mutability_policy::read(
            reinterpret_cast<volatile uint32_t*>(address),
            mask, offset
        );
    }
};
```

Mutability policies

- Mutability policies implement the static read and write methods
- The way they implement those methods defines the behavior of the register

A read-only policy

```
struct ro_t
{

    static uint32_t read(
        volatile uint32_t* reg,
        uint32_t mask,
        uint32_t offset
    )
    {
        return (*reg >> offset) & mask;
    }
};
```

A read-only register

```
struct scs
{
    typedef reg_t<0xe01fc1a0, 1, 6, ro_t> oscstat;
};
```

A read-only register

```
struct scs
{
    typedef reg_t<0xe01fc1a0, 1, 6, ro_t> oscstat;
};

do {} while (scs::oscstat::read() == 0);
```

A read-only register

```
struct scs
{
    typedef reg_t<0xe01fc1a0, 1, 6, ro_t> oscstat;
};

do {} while (scs::oscstat::read() == 0);

scs::ocsstat::write(1); // doesn't compile
```

A write-only policy

```
struct wo_t
{
    static void write(
        volatile uint32_t* reg,
        uint32_t mask,
        uint32_t offset,
        uint32_t value
    )
    {
        *reg = (value & mask) << offset;
    }
};
```

A write-only register

```
struct fioset
{
    typedef reg_t<0x3fffc018, 1, 21, wo_t> led1;
};
```

A write-only register

```
struct fioset
{
    typedef reg_t<0x3fffc018, 1, 21, wo_t> led1;
};

fioset::led1::write(1);
```

A write-only register

```
struct fioset
{
    typedef reg_t<0x3fffc018, 1, 21, wo_t> led1;
};

fioset::led1::write(1);

uint32_t result = fioset::led1::read(); // doesn't compile
```

Read-write policy

```
struct rw_t : ro_t
{
    static void write(
        volatile word_t* reg,
        word_t mask,
        word_t offset,
        word_t value
    )
    {
        *reg =
            (*reg & ~(mask << offset))
            |
            ((value & mask) << offset);
    }
};
```

Exotic policy

- Keyed Register
  - Every time you write to this register, you have to write a special value to the most significant byte

A keyed register policy

```
template
<
	uint32_t key_mask,
	uint32_t key_offset,
	uint32_t key_value
>
struct keyed_wo_t
{
	static void write(
		volatile uint32_t* reg,
		uint32_t mask,
		uint32_t offset,
		uint32_t value
	)
	{
		volatile uint32_t tmp = (value & mask) << offset;
```

A keyed register policy

```
template
<
	uint32_t key_mask,
	uint32_t key_offset,
	uint32_t key_value
>
struct keyed_wo_t
{
	static void write(
		volatile uint32_t* reg,
		uint32_t mask,
		uint32_t offset,
		uint32_t value
	)
	{
		volatile uint32_t tmp = (value & mask) << offset;
		tmp &= ~(key_mask << key_offset);
```

A keyed register policy

```
template
<
	uint32_t key_mask,
	uint32_t key_offset,
	uint32_t key_value
>
struct keyed_wo_t
{
	static void write(
		volatile uint32_t* reg,
		uint32_t mask,
		uint32_t offset,
		uint32_t value
	)
	{
		volatile uint32_t tmp = (value & mask) << offset;
		tmp &= ~(key_mask << key_offset);
		tmp |= (key_value & key_mask) << key_offset;
```

A keyed register policy

```
template
<
	uint32_t key_mask,
	uint32_t key_offset,
	uint32_t key_value
>
struct keyed_wo_t
{
	static void write(
		volatile uint32_t* reg,
		uint32_t mask,
		uint32_t offset,
		uint32_t value
	)
	{
		volatile uint32_t tmp = (value & mask) << offset;
		tmp &= ~(key_mask << key_offset);
		tmp |= (key_value & key_mask) << key_offset;
		*reg = tmp;
	}
};
```

A keyed register

```
struct sys {
    typedef reg_t
    <
        0xfffe0080,
        1,
        0,
        keyed_wo_t<0xff, 24, 0xa5>
    > reset; };

sys::reset::write(1); // writes 0xa5000001
```

More uses for policies

- Log all register accesses
- Gather metrics about register accesses
- Add delay or timing code
- Perform mutual exclusion
- Anything else that makes use of compile-time information

Other possibilities

- remote_reg_t
  - Access registers of off-chip (SPI, I2C, RS232) devices using the same idiom
- Collect and flush several writes at once
  - Use expression templates to collapse a series of writes into a single write to allow maximum performance while preserving readability
  - Useful in conjunction with off-chip register access to minimize bus transactions
- Mock registers for unit testing or desktop simulator

Performance

- Much of the masking / anding / oring is computed at compile time
- The generated assembly is as fast as using C macros for the test platform (arm-none-eabi-gcc) when compiling with -Os
    - The compiler collapses and inlines the static function calls (no impact on the runtime stack)

Limitations

- Pushback from collaboraters unused to the technique
- You may have to write a lot of register definitions
- Naive use may introduce a performance penalty, esp. with read-write registers
    - One read-modify-write per field rather than per register unless you develop the idea mentioned in "Other possibilities"
- Very different unoptimized vs. optimized performance
    - Code essentially needs to be optimized
    - Some uses cannot run unoptimized, eg. stack pointer initialization

Benefits

- Namespaces or nested structs make it easy to guess register names.
  - Namespace or template aliases can lessen the typing burden
- Register access statements look like actions
- It’s clear when you’re reading or writing
- Compile time safety
- Speed
- Once you get the masking / shifting / anding / oring right in the policy classes, you don’t have to do it again

Questions?

Original paper at
http://yogiken.wordpress.com/2010/02/10/on-publishing/

Code and Slides at
https://github.com/kensmith/embsys

References:

Alexandrescu, 2001. Andrei Alexandrescu, Modern C++ Design, Addison-Wesley

Goodliffe, 2005. Pete Goodliffe, Register Access in C++: Exploiting C++'s features for efficient and safe hardware register access,
http://www.ddj.com/cpp/184401954
http://accu.org/index.php/journals/281

Meyers, 2010. Scott Meyers, Effective C++ in Embedded Systems,
http://www.aristeia.com/c++-in-embedded.html

Moen, 2010. Martin Moene, One Approach to Using Hardware Registers in C++,
http://accu.org/index.php/journals/1606

Saks, 1998. Dan Saks, Representing and Manipulating Hardware in Standard C and C++,
http://www.open-std.org/jtc1/sc22/wg21/docs/ESC_SF_02_465_paper.pdf