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 unsigned int)
#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
- Disadvantages
  - Easy to get it wrong
    - 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 unsigned enabled : 1;
  volatile unsigned flag : 1;
  volatile unsigned 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 get at 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
<
    unsigned long address,
    unsigned mask,
    unsigned offset,
    class mutability_policy
>
struct reg_t
{
    static void write(unsigned value)
    {
        mutability_policy::write(
            reinterpret_cast<volatile unsigned*>(address),
            mask,
            offset,
            value
        );
    }
    static unsigned read()
    {
        return mutability_policy::read(
            reinterpret_cast<volatile unsigned*>(address),
            mask,
            offset
        );
    }
```

Mutability policies

- Mutability policies implement the static read and write methods
- The way they implement those methods defines the policy

Read-only policy

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

A read-only register

```
namespace periph
{
    namespace sr
    {
        typedef reg_t<0xfffe0008, 0x1, 0, ro_t> en;
    }
}
bool enabled = periph::sr::en::read();
periph::sr::en::write(1); // compile time error
```

Write-only policy

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

A write-only register

```
namespace periph
{
    namespace cr
    {
        typedef reg_t<0xfffe0000, 0x1, 0, wo_t> en;
    }
}
periph::cr::en::write(1);
periph::cr::en::read(); // compile-time error
```

Read-write policy

```
struct rw_t : ro_t
{
    static void write(
        volatile unsigned* reg,
        unsigned mask,
        unsigned offset,
        unsigned 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 high order byte.

Keyed register policy

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

A keyed register

```
namespace rst
{
    typedef reg_t<
        0xfffe0080,
        0x1,
        0,
        keyed_wo_t<0xff, 24, 0xac>
    > reset;
}
rst::reset::write(1); // writes 0xac000001
```

More uses for policies

- Log all register accesses
- Gather metrics about register accesses
  - These two are probably more interesting for a software test harness than for production
- Add delay or timing code
- Perform mutual exclusion

Performance

- Much of the masking/anding/oring is computed at compile time
- The generated assembly is just as fast as using C macros for the test platform (arm-elf-gcc) when using -Os
  - The compiler collapses and inlines the static function calls (no impact on the runtime stack)

Limitations

- Writing multiple values simultaneously requires a policy for that purpose
- You may have to write a lot of register definitions despite probably having a premade header from your toolchain vendor (unless they use this technique!)

Benefits

- Namespaces make it easy to guess register names.
  - Namespace aliases can relieve some of the typing burden.
- Register access statements look like actions.
- It is clear when you are reading or writing.

Benefits

- Once you get the masking/ shifting/anding/oring right in the policy classes, you don’t have to do it again.

Questions?

Based on the original work, *C++ Hardware Register Access Redux*, http://yogiken.files.wordpress.com/2010/02/c-register-access.pdf

References:

- Alexandrescu, 2001. Andrei Alexandrescu, *Modern C++ Design, Addison-Wesley (2001).*
- 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 and http://accu.org/index.php/journals/281* (2005).
- Moene, 2010. Martin Moene, *One Approach to Using Hardware Registers in C++,* http://www.eld.leidenuniv.nl/~moene/Home/publications/accu/overload95-register/ (2010).
- Saks, 1998. Dan Saks, *Representing and Manipulating Hardware in Standard C an C++,* http://www.openstd.org/jtc1/sc22/wg21/docs/ESC_SF_02_465_paper.pdf (1998).