C++ Hardware Register Access Redux

Ken Smith

kgsmith@gmail.com

ABSTRACT

Everything about a hardware register is listed in the datasheet for the processor or peripheral. We know its address, its offset within the word if any, and we know whether the register is readable, writable, or both. Therefore, we know how to define a register or any of its subfields at compile time. Many register manipulation techniques exploit some of these pieces of a priori knowledge to ensure safety and efficiency but none I have seen so far exploit them all. This article presents an approach for hardware register access in C++ that puts all of this a priori knowledge to use to help ensure safety without sacrificing efficiency.

1. Contemporary Alternatives

Other articles explain novel approaches to defining register access techniques in C++ [Saks, 1998, Goodliffe, 2005]. I won't reiterate those approaches here but of the alternatives, I gained the most inspiration from Goodliffe's.

2. Suppose...

Let the following serve as our pretend datasheet. Suppose our 32-bit hardware platform defines a peripheral whose base address is `0xfffe0000` and has the following registers. Offset is the address offset added to `0xfffe0000` to determine the location in memory of the register. Reading a write-only register has no effect and yields all zeroes. Writing to a read-only register has no effect.

Offset	Register	Register Name	Access
0x00	Control Register	PERIPH_CR	Write-only
0x04	Mode Register	PERIPH_MR	Read/Write
0x08	Status Register	PERIPH_SR	Read-only

The following tables display the bit layout of each of the registers. Bit 0 is the least significant and bit 31 is the most significant. The text following each table describes the fields within the register and their effects.

PERIPH_CR (Write-only)

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	**22**	**21**	**20**	**19**	**18**	**17**	**16**
-	-	-	-	-	-	-	-
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**
-	-	-	-	-	-	-	-
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
-	-	-	-	-	-	DIS	EN

EN
0 = no effect
1 = enable peripheral
DIS
0 = no effect
1 = disable peripheral

PERIPH_MR (Read/Write)

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	**22**	**21**	**20**	**19**	**18**	**17**	**16**
-	-	-	-	-	-	-	-
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**
-	-	-	-	DELAY			
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
CLOCKDIV							

CLOCKDIV
0 - 255 Clock divisor
DELAY
0 - 16 Delay between consecutive transfers in clock cycles

PERIPH_SR (Read-only)

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	**22**	**21**	**20**	**19**	**18**	**17**	**16**
-	-	-	-	-	-	-	-
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**
-	-	-	-	-	-	-	-
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
-	-	-	-	-	-	-	EN

EN
0 = disabled
1 = enabled

3. A Policy Based Design

First, the code.

```
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
        );
    }
};
```

As I claimed earlier, everything about a register is known at compile time. It's address and location within a word are given by the template parameters, `address`, `mask`, and `offset`. `address`, as you would expect is the memory location of the register. `mask` is a 32-bit value with as many consecutive bits set to one starting with the least significant bit representing the width of the field. `offset` is how far to left-shift that mask to get to the precise location within the word of the field. The last, and most interesting template parameter, is `mutability_policy`. This parameter accepts a class that implements how to read and write this register. This is how we implement a registers readability, writability, or both. In fact, we can implement basically any specialized mutability policy that does automatic checking or writes magic values as you'll see later in this document.

Any class can be a mutability policy but there are really only four meaningful variations.

read-only

Alexandrescu's policies help us design a variety of register types with a single interface [Alexandrescu, 2001]. Here is a policy that implements the write-only concept. Note the lack of an implementation for `read`.

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

Finally, here is the definition for a write only register field.

```
typedef reg_t<0xfffe0000, 0x1, 0, wo_t> en;
```

This fully specifies our first field from the PERIPH_CR register. With this technique, we give each field its own name. This ultimately allows us to be indifferent to whether a field is defined as a whole or partial register. We can manipulate the fields independently. Now we can enable our peripheral like this.

```
en::write(1);
```

4. Clarity

With so many similarly named registers in any given datasheet, we can avoid name collisions with namespaces.

```
namespace periph
{
    namespace cr
    {
        typedef reg_t<0xfffe0000, 0x1, 0, wo_t> en;
    }
}
```

Now we can write this.

```
periph::cr::en::write(1);
```

It is easy to conceptualize what is going on here. I'm writing to the `EN` field of the `CR` register of `PERIPH`. By carefully matching register names to what is in the datasheet, we can program a peripheral just by looking in the datasheet. We can mentally compute the names of the register fields without looking at the header file.

5. Safety

Our first benefit in terms of safety is that we name each register field. We don't need to care about what is going on with the rest of the register and we don't need to concern ourselves with shifting properly to get access to the appropriate field without stepping on other fields.

Our first example `periph::cr::en` is a write-only register field. So what happens if we write this?

```
periph::cr::en::read();
```

Our static class, `reg_t`, implements a static read method so the above should work, right? Your compiler probably doesn't think so. My compiler tells me, `"'read' is not a member of 'wo_t'"`. It's easy to underestimate the utility of this assistance from the compiler. We have just been told that we tried to read a write-only register *at compile time*. The compiler has helped prevent us from doing something which is almost certainly an error.

6. Implementing the Read-only Policy

Here is a read-only policy.

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

This enables us to implement our simple status register.

```
namespace periph
{
    namespace sr
    {
        typedef reg_t<0xfffe0008, 0x1, 0, ro_t> en;
    }
}
```

Now we have a way to check to see if our peripheral is enabled.

```
bool enabled = periph::sr::en::read();
```

And, as you would expect, you can't call `::write()` on `periph::sr::en`.

7. Implementing the Read/Write Policy

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

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

The reader may notice that our `read` implementation is exactly the same as the read-only policy. The following is a better definition for `rw_t`.

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

Since we can't just write zeroes to a read/write register field and hope they are ignored, we must read the contents and mask in our new value.

8. A Complete Implementation

Let's get right to the code.

```
namespace my_organization
{
    namespace my_platform
    {
        namespace periph
        {
            namespace cr
            {
                typedef reg_t<0xfffe0000, 0x1, 0, wo_t> en;
                typedef reg_t<0xfffe0000, 0x1, 1, wo_t> dis;
            }
            namespace mr
            {
                typedef reg_t<0xfffe0004, 0xff, 0, rw_t> clockdiv;
                typedef reg_t<0xfffe0004, 0xf, 8, rw_t> delay;
            }
            namespace sr
            {
                typedef reg_t<0xfffe0008, 0x1, 0, ro_t> en;
            }
        }
    }
}
```

The new register fields, `clockdiv` and `delay` give us examples of register fields which occupy more than a single byte. The second template parameter is a mask to apply to any read or written values. We also have two examples of fields which don't begin at position zero. The third template parameter defines the least significant bit of the register field.

9. From the Datasheet Directly to Code

Once we have the machinery in place to implement the register fields and the mutability policies, defining a datasheet worth of registers is easy and straightforward. It is also easy to compare the definitions to the datasheet and see that they are correct. You can almost read the register definition as an English sentence.

```
typedef reg_t<0xfffe0000, 0x1, 1, wo_t> dis;
```

Translation: This register field called "dis" lives at 0xfffe0000 and occupies one bit at offset one and is write-only.

Also, if you use cscope, doing a definition search for a field name shows you the complete definition immediately without having to actually open the source file. You can quickly double check that you're dealing with the right register.

10. Dealing With Long Register Names

With the additional namespaces, the register names can be rather lengthy.

```
my_organization::my_platform::periph::cr::en::write();
```

Ameliorate this with namespace aliases.

```
namespace cr = my_organization::my_platform::periph::cr;
cr::en::write(1);

// or

namespace periph = my_organization::my_platform::periph;
periph::cr::en::write(1);
bool is_enabled = periph::sr::en::read();
```

11. Password Protected Registers

Occasionally you'll encounter a register whose value must be written with a password or key simultaneously with some kind of command. Let's fabricate one for our peripheral.

Offset	Register	Register Name	Access
0x80	Reset Register	PERIPH_RST	Write-only

PERIPH_RST (Write-only)

31	30	29	28	27	26	25	24
KEY							
23	**22**	**21**	**20**	**19**	**18**	**17**	**16**
-	-	-	-	-	-	-	-
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**
-	-	-	-	-	-	-	-
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
-	-	-	-	-	-	-	RESET

RESET
0 = no effect
1 = enable peripheral
KEY
Must be written with 0xac when RESET == 1 to reset the peripheral.

This

```
typedef reg_t<0xfffe0080, 0x1, 0, wo_t> reset;
```

is not sufficent to define this register field because KEY will be written with zeroes nullifying the action.

12. Policies to the Rescue (again)

Here is a policy to enable writing registers like our new reset register.

```
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;
    }
};
```

And here is the register definition.

```
namespace rst
{
    typedef reg_t<0xfffe0080, 0x1, 0, keyed_wo_t<0xff, 24, 0xac> > reset;
}
```

Every time you write to RESET, KEY will get the value 0xac per the specification.

13. Efficiency

So far, I've been selling this idea on the value of the added safety due to all the static checking. I've also been touting the improved readability of code using this technique. Now let me sell you on how performant the code this technique generates will be.

To set a baseline for performance, we'll compare the generated assembly for this technique against that generated by the following C.

```
#define PERIPH_CR ((volatile unsigned*) 0xfffe0000)

void enable_peripheral(void)
{
    *PERIPH_CR = 1;
}
```

This function compiles into the following four lines of assembly on my platform using ARM ELF GCC 4.1.1. The compile command I used optimizes for size.

```
mov r2, #1
mvn r3, #126976
str r2, [r3, #-4095]
bx  lr
```

And here is `enable_peripheral` using our technique also compiled with optimization for size.

```
void enable_peripheral()
{
    namespace cr = my_organization::my_platform::periph::cr;
    cr::en::write(1);
}
```

Here's the assembly.

```
mov r2, #1
mvn r3, #126976
str r2, [r3, #-4095]
bx  lr
```

They're completely identical. (I am as surprised as you are. I thought they'd be at least a little different.) All of the bit shifting rigamarole our `reg_t` and its policies perform are done at compile time. Heavy use of this approach may lengthen your compile times but your execution times will be just as fast as if you had written everything in C.

14. Unit Testing Your Register Implementation

No implementation is complete without good unit tests. It would be time consuming to write a standard program to iterate through all the possibilities. Since registers are configured with compile time constants, you'd have to write each case by hand. Luckily, Abrahams and Gurtovoy have a solution [Abrahams, 2005]. Let's start by defining a special version of `ro_t` that we can prime with an arbitrary value.

```
template<unsigned initialized_to>
struct soft_ro_t
{
    static unsigned read(
        unsigned volatile* reg,
        unsigned mask,
        unsigned offset
    )
    {
        (void) reg;
        unsigned volatile soft_register = initialized_to;
        return ro_t::read(&soft_register, mask, offset);
    }
};
```

Now, here's the unit test itself using Boost.Test [Rozental, 2010].

```
template<unsigned mask, unsigned offset>
struct ro_test_t
{
    static void run()
    {
        typedef reg_t<0, mask, offset, soft_ro_t<mask << offset> > on;
        BOOST_ASSERT(on::read() == mask);
        typedef reg_t<0, mask, offset, soft_ro_t<~(mask << offset)> > off;
        BOOST_ASSERT(off::read() == 0);
    }
};
```

Now how do we check all possible read-only register field definitions to ensure that, no matter what values we give to the `reg_t` template, we'll have a working register? We'll use template metaprogramming to generate them.

```
template<template <unsigned, unsigned> class test>
struct generate_tests_t
{
    static void run()
    {
        generate_masks_t<test, 0x1>::run();
    }
};
```

We can now write this.

```
generate_tests_t<ro_test_t>::run();
```

It won't compile yet because we don't have a definition for `generate_masks_t`.

```
template<template <unsigned, unsigned> class test, unsigned mask>
struct generate_masks_t
{
    static void run()
    {
        generate_offsets_t<test, mask, num_shifts_t<mask>::value>::run();
        generate_masks_t<test, (mask << 1) | 1>::run();
    }
};

template<template <unsigned, unsigned> class test>
struct generate_masks_t<test, 0xffffffff>
{
    static void run()
    {
        generate_offsets_t<test, 0xffffffff, num_shifts_t<0xffffffff>::value>::run();
    }
};
```

This construct recursively generates all the possible masks from 0x1 through 0xffffffff calling into the not yet implemented `generate_offsets_t` and `num_shifts_t`. Here's `num_shifts_t.`

```
template<unsigned mask>
struct num_shifts_t
{
    static const unsigned value = 1 + num_shifts_t<(mask << 1) | 1>::value;
};

template<>
struct num_shifts_t<0xffffffff>
{
    static const unsigned value = 0;
};
```

`num_shifts_t<mask>::value` is an integer from 0 to 31 representing the number of times you could shift `mask` before you overflow. This ensures that we only generate tests for registers that could actually exist. For example, you wouldn't want to have an eight bit register field that starts at bit position 30. Finally, here is how we generate the field offsets.

```
template<template <unsigned, unsigned> class test, unsigned mask, unsigned offset>
struct generate_offsets_t
{
    static void run()
    {
        test<mask, offset>::run();
        generate_offsets_t<test, mask, offset-1>::run();
    }
};

template<template <unsigned, unsigned> class test, unsigned mask>
struct generate_offsets_t<test, mask, 0>
{
    static void run()
    {
        test<mask, 0>::run();
    }
};
```

`generate_offsets_t` runs backward from the most significant possible bit position to zero. It takes all of the statically generated register parameters and runs the test. It ends up running the test 528 times with all possible combinations of mask and offset. When the tests succeed, you can use your read-only register implementation with confidence.

Tests for the other register types can use the same generation framework. You just have to implement a wrapper for the mutability policy class that gives you the visibility that you require. For example, you may want to wrap `wo_t` and provide it with `read` so you can check the results of the operation.

You can also pepper `BOOST_STATIC_ASSERT` or C++0x's `static_assert` to ensure register definitions themselves don't violate the constraints we tested for above.

References

Saks, 1998.

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

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).

Alexandrescu, 2001.

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

Abrahams, 2005.

David Abrahams, Aleksey Gurtovoy, *C++ Template Metaprogramming*, Addison-Wesley and Pearson Education, Inc. (2005).

Rozental, 2010.

Gennadiy Rozental, *Boost.Test*, http://www.boost.org/doc/libs/1_41_0/libs/test/doc/html/index.html (2010).