print | close

An Evolution of Special Function Register Abstraction

Electronic Design

Odin Holmes

Tue, 2015-10-13 11:15

Special function registers (SFRs) provide an interface to core and peripheral hardware functionality of a microprocessor or microcontroller. They are commonly mapped into the processor's address space, similar to RAM. To a C or C++ compiler, SFRs are essentially indistinguishable from RAM.

On the other hand, SFRs are linked to functionality that, in contrast to plain RAM, is externally observable. The optimizer has no knowledge of this external observability; therefore, adding the volatile modifier to the type of SFR variables is usually required. The volatile modifier essentially turns off all optimizations acting on the affected address (loads and stores are not optimized away or reordered).

Related

Car-Hacked! Flaw in Jeep Revealed

What's the Difference Between Big Data and Big Content?

DIY Makers, Hobbyists, and Experimenters Get Professional Software

Enough with the theory talk. Let's look at a real world example. Say we want to disable the spi0 clock and enable the spi1 and i2c0 clocks on a LPC15xx Cortex M3 microcontroller. The simplest method often taught in school would be:

```
*(volatile int*)0x400740C8 &= ~0x200; //disable spi0 clock
*(volatile int*)0x400740C8 |= 0x400; //enable spi1 clock
*(volatile int*)0x400740C8 |= 0x2000; //enable i2c0 clock
```

This results in the following optimized assembler on GCC 4.8. It should be noted that optimization and code-generation results will vary depending on the surrounding code, compiler version, and settings used.

```
ldr r3, [pc, #24] ; load address of SYSAHBCLKCTRL1 to r3
ldr r2, [r3, #0] ; load *r3 in r2
bic.w r2, r2, #512 ; bit clear with mask of 0x200
str r2, [r3, #0] ; store r2 in *r3
ldr r2, [r3, #0] ; load *r3 to r2
orr.w r2, r2, #1024 ; bitwise or with mask of 0x400
str r2, [r3, #0] ; store r2 in *r3
ldr r2, [r3, #0] ; load *r3 to r2
orr.w r2, r2, #8192 ; bitwise or with mask of 0x2000
str r2, [r3, #0] ; store r2 in *r3
```

mmon Practice

Although this may be the method that's often taught in school, the majority of production code is written using a more sophisticated method. The above method surely violates Scott Meyers' most important design guideline: "*Make interfaces easy to use correctly and hard to use incorrectly*." It's also not as efficient as it could be. Can you spot the optimization potential?

A brief search of popular *github.com* repositories shows this to be the most common method:

```
//in a library header file
#define SYSAHBCLKCTRL0 *(volatile int*)0x400740C4
#define SYSAHBCLKCTRL1 *(volatile int*)0x400740C8
#define SPI0CLK 9
#define SPI1CLK 10
#define I2CClk 13
//user code
unsigned reg = SYSAHBCLKCTRL1;
reg |= (1< reg &= ~(1< SYSAHBCLKCTRL1 = reg;
```

(Ed. note: From here on in, bolded items embedded in the text are code terms/functions.) Although this method increases readability, it's still unsafe. One could easily mix up **SYSAHBCLKCTRL0** and **SYSAHBCLKCTRL1**, or use **SPI1CLK** as a value rather than an offset. The resulting assembler is similar except for the optimizations of merging **SPI1CLK** and **I2CClk**, and using a non-volatile temporary to allow the compiler to remove the store and reload in the middle.

```
ldr r3, [pc, #12] ; load address of SYSAHBCLKCTRL1 to r3
ldr r2, [r3, #0] ; load *r3 in r2
bic.w r2, r2, #512 ; bit clear with mask of 0x200
orr.w r2, r2, #9216 ; bitwise or with mask of 0x2400
str r2, [r3, #0] ; store r2 in *r3
```

A C Library Solution with Tradeoffs

There are several C Libraries out there. I have chosen LPCOpen from NXP as an example because it's quite comprehensive and popular. LPCOpen provides a:

```
        void Chip_Clock_EnablePeriphClock(int);
```

and

```
        void Chip_Clock_DisablePeriphClock(int);
```

function.

```
Chip_Clock_DisablePeriphClock(SYSCTL_CLOCK_SPI0);
Chip_Clock_EnablePeriphClock(SYSCTL_CLOCK_SPI1);
Chip_Clock_EnablePeriphClock(SYSCTL_CLOCK_I2C0);
```

We can no longer use the wrong register or use the bit offset macro as a value. We also no longer need to know if a 1 represents enabled (sometimes it's the other way around in hardware). On the other hand, since the function essentially takes any int (actually an enum, which is implicitly convertible from an int), we could still pass it garbage data. Also, the names still must be a bit ugly because they're macros.

Looking at the assembler, we can see the tradeoff that accompanies this added safety. We can no longer use the

k of merging bits, and since the functions are contained in a separate library, the compiler can't easily inline m.

```
push {r3, lr}
movs r0, #41 ; 0x29
bl 0x228
movs r0, #42 ; 0x2a
bl 0x1f8
movs r0, #45 ; 0x2d
bl 0x1f8
pop {r3, pc}
movs r0, r0

Chip_Clock_EnablePeriphClock:
ldr r2, [pc, #40] ;
cmp r0, #31
mov.w r3, #1
itett hi
ldrhi.w r1, [r2, #200] ; 0xc8
ldr.w r1, [r2, #196] ; 0xc4
subs r0, #32
lslls r3, r0
itete ls
lslls r3, r0
orrs r3, r1
orrs r3, r1
str.w r3, [r2, #200] ; 0xc8
it ls ; unpredictable
strls.w r3, [r2, #196] ; 0xc4
bx lr
nop
ands r0, r0
ands r7, r0
ldr r2, [pc, #44] ; (0x258 cmp r0, #31
mov.w r3, #1
itett hi
ldrhi.w r1, [r2, #200] ; 0xc8
ldr.w r1, [r2, #196] ; 0xc4
subs r0, #32
lslls r3, r0
itete ls
lslls r3, r0
bic.w r3, r1, r3
bic.w r3, r1, r3
str.w r3, [r2, #200] ; 0xc8
it ls
strls.w r3, [r2, #196] ; 0xc4
bx lr
nop
ands r0, r0
ands r7, r0
```

If you have design constraints like legacy code, an old compiler, or specific coding guidelines like MISRA C that you must fulfill, LPCOpen is a good choice.

A C++11 Library Solution

Using the power of C++11's **constexpr** and template meta programming, one can provide a familiar C-like interface, complete type safety, and preform a number of optimizations behind the scenes at compile time before generating the code. Here's an example using the Kvasir library:

```
  ɔply(clear(AHBClock::Enabled::spi0),
  ɐt(AHBClock::Enabled::spi1),
set(AHBClock::Enabled::i2c0));
```

This is coming closer to meeting the “most important design guideline.” We no longer need to remember all of the optimization tricks we have learned. It’s surely very hard to make any mistakes. In fact, because of the library, one can essentially skip the corresponding part of the chip's user’s manual. It just works. So what is this complete type safety going to cost us?

```
ldr r3, [pc, #12] ; load address of SYSAHBCLKCTRL1 to r3
ldr r2, [r3, #0] ; load *r3 in r2
bic.w r2, r2, #9728 ; bit clear with mask of 0x2600
orr.w r2, r2, #9216 ; bitwise or with mask of 0x2400
str r2, [r3, #0] ; store r2 in *r3
```

Wow! The assembler code is just as efficient as the best example so far. It costs us nothing. To be fair, this needs to be compiled with at least C++11 enabled. However, for most projects, that should not be a problem. But wait, there’s still more optimization potential here! Can anyone spot it this time?

If you do take a look at the LPC15xx user’s manual, you will see that in this special case, all of the other bits in byte 1 of **SYSAHBCLKCTRL1** are reserved. It’s perfectly OK to write 0s to reserved bits, so instead of modifying the whole 32 bits of the register, we can just write to byte 1 in 8-bit mode and fill in all reserved bits with 0s.

```
ldr r3, [pc, #4] ; load address to r3
movs r2, #36 ; load literal 0x24
strb r2, [r3, #0] ; store in byte mode
```

This optimization should not be used when hand-coding—the efficiency versus readability tradeoff is not in its favor, not to mention programmer productivity. However, by using meta programming to divide interface from implementation, this and other optimizations can be fully encapsulated. Code readability and programmer productivity tradeoffs are no longer an issue.

Admittedly, this is a bit of a contrived example, where we just happened to be modifying all of the non-reserved bits in a particular byte. But if the register has larger bit fields, this special case is much more common. In other special cases, other optimizations are possible—it all adds up in the end.

What About Larger Fields?

So far we have been setting and clearing bits. Kvasir also supports writing to bit fields in a type safe manner.

```
        apply(write(Can::mode,Can::normalMode));
```

Here, **Can::mode** is a bit location that is two bits wide and **normalMode** is a compile time value of type **enum class Can::Modes**.

Where Do All of These Presets Come From?

Good embedded designs usually include a hardware abstraction layer (HAL) between the application code and low-level hardware access. The problem is, without zero cost encapsulation, many design purity versus efficiency tradeoffs need to be reconciled. The end result often allows some hardware details to creep up into the application code, often even in the form of ugly and error-prone preprocessor directives.

ıddition, the design of the HAL is typically optimized for efficiency with the specific application. This ninates the possibility of a generic HAL; most designs involve hand-coding the HAL as well as the application.

With the introduction of meta programming, zero-cost encapsulation is possible and many of the design tradeoffs are eliminated. The Kvasir library includes a small but growing number of “chip abstraction” files that contain all of the necessary **BitLocations** and device-specific information for a given chip. This is an attempt, for the first time, to provide a zero-cost HAL to the user once and for all.

Examples in this article assume a chip abstraction is present that defines things like Can::mode. You can check if someone has made one, or build your own. Ideally, the chip manufacturer would provide these files.

What About Reading?

Reading with Kvasir::Register also works as expected:

```
if(UART0_CONFIG & STOP_LENGTH_MASK == 1){/*...*/}
//becomes
if(apply(read(Uart0::Config::stopLength)) == 1){/*...*/}
```

But wait, this will not do what you think! According to the LPC15xx user’s manual, a stop length value of 0 means 1 stop bit and a value of 1 means 2 stop bits. If we modify the type of the BitLocation **stopLength** to make its value type a strongly classed enum, it’s no longer possible to make the same error:

```
//in a header
namespace Uart0{
struct Config{
enum class StopBits{one, two};
static constexpr Register::RWLocation<
0x40040000, //address
(1<<6), //bit field mask
(1<<1)|(5<<8)|(1<<13)|(3<<16)|(0x0F << 24), //unwritable
StopBits //value type
> stopLength{};
};
}
//in user code
using namespace Uart0;
if(apply(read(Config::stopLength)) == 1){/*...*/}
//^ error, StopBits is not convertible to int
if(apply(read(Config::stopLength))==Config::StopBits::one){/*...*/} //works
```

What About Clear on Read?

If you are not familiar with this problem, consider this piece of code:

```
if(REGISTER & 0x01){/*...*/}
else if(REGISTER & 0x04){/*...*/}
```

If **REGISTER** is clear on read, we have a very subtle bug in our code that may be missed in code review:

```
auto temp = REGISTER;
if(temp & 0x01){}
else if(temp & 0x04){}
```

s would solve the problem, but how should the code reviewer know that this register is clear on read?

ısir::Register allows a BitLocation to store access information. If the chip abstraction implementer did a good job when making the bit-field **register**, then this information is known by the library:

```
if(apply(read(register)) & 0x01){/*...*/}
//^ compiler issues an error telling you to use readAndClear
if(apply(readAndClear(register)) & 0x01){/*...*/}
//^ potential danger can be easily seen in code review
```

What About Reading Multiple Values?

In most cases, there's no real reason to read multiple values in one apply statement. However, if the bit fields you want to read are all in the same register, it can be more efficient to read once and then mask off the different parts later. Since we want to be at least as efficient as C in all circumstances, we should solve this problem.

More importantly, though, some registers contain multiple bit fields but are clear on read. Without multiple read support, it would be impossible to access them correctly. Therefore, Kvasir also supports multiple reads in one apply statement and returns a kind of tuple containing all results:

```
auto result = apply(read(field1),read(field2));
if(get<0>(result) == 4){/*...*/} //access by index like std::tuple
if(get(result, field2) == 8){/*...*/}
//^ access by bit location also possible and can be more clear
```

One can also mix reads and writes into one apply statement. However, this is only advantageous when both act on the same register.

What About Writing Run-Time Known Values?

Thus far, I have only shown examples where the values going into the registers are known at compile time. It may be counterintuitive, but I have found that the majority of values written in small processor embedded programs are actually known at compile time. Nevertheless, there are bound to be some that are not. The syntax is actually identical, although a completely different overloaded **write()** function is used "under the hood."

```
void f(Can::Modes m){
apply(write(Can::mode,m));
}
```

Needless to say, we can't make the same optimizations on run-time known values as in the above compile-time examples. However, there are a few corner cases that will allow the optimizer to do more for us (by marking less variables volatile than the user otherwise would). Kvasir can provide type safety (e.g., when using strongly classed enums, etc.), though, and eliminate off-by-one shifting errors. Kvasir can also guarantee that if the chip abstraction is correct, no bit-field access will overflow into other fields, or reserved bits if the value written is larger than the field.

There's an inherent tradeoff here. By masking and shifting the values at run time, we may be less efficient than equivalent C code in the special case when the user knows that the value is within range or that the value is already correctly shifted. When choosing between safety and efficiency, I would always argue for safety, but then again, as stated by Bjarne Stroustrup, "no one knows what most C++ programmers do."

Instances in which the user knows that the value has been masked and shifted correctly are not as common as it would seem. If the value in question did not come from outside the program (through an input port etc.), it's

hly likely that it's known at compile time. If it did come in from the outside world, then who's to say that's out of bounds and needs to be masked?

At times, there's a corner case where we want to mask and shift a variable once and then write it to the register multiple times. In that case, it's possible to store result of the **write()** function.

Are there more common corner cases? Should Kvasir also provide an **unsafeWrite()**, which doesn't perform masking or shifting? Should we also provide an **unsafeRead()**, too? I would like to hear your input in the comments.

Interface Implementation

So what is this **spi0** variable, and how exactly do I make my own chip abstractions? The Kvasir::Register module uses the concept of a **Kvasir::Register::BitLocation**, which stores the address, access rights (e.g., read only), special functionality (e.g., clear on read), mask, associated type (e.g., strongly classed enum), and writable (e.g., non-reserved) bits of a particular bit field in a register.

Kvasir::Register::apply is a variadic function template that takes parameters of type **Kvasir::Register::Action**. It should be noted that because **BitLocation**, **Action**, and **apply()** are all in the same namespace, Argument Dependent Lookup (otherwise known as ADL or könig lookup) will find the correct functions without namespace qualification.

Since the data must be available for compile-time evaluation, the parameters to **BitLocation** and **Action** are encoded into the type of the object. If you are not familiar with C++ template meta programming, you may find this section confusing—don't worry, you can still use the library. After all, few C++ programmers understand how **std::regex** or **std::tuple** are implemented, but they're used every day.

```
using Address = Register::Address<0x400740C8,
Register::maskFromRange(8,8,12,11,16,14,20,20,22,22,31,24)>;
//^ mask of reserved bits
constexpr Register::RWBitLocT spi0;
constexpr Register::RWBitLocT spi1;
```

Here **RWBitLocT** is a handy alias for creating read writable **BitLocations**. Making a chip abstraction file may be tedious, though. As mentioned above, it only has to be done once by one person (and properly open-sourced).

The **set()** and **clear()** functions take a **BitLocation** as a parameter and return an **Action**.

Two **Actions** can be merged (combined) if they act on different bit fields of the same register. For example, a write to a given bit field probably consists of:

• A read of the whole register

• The application of a mask to clear only the related bits

• A bitwise of the value to be written

• A write back to the register

If we have two writes to different bit fields of the same register, we can combine the masks and the values and only perform one read-modify-write.

nstexpr Meta Functions

Changing one type to another is usually the domain of meta functions. But these have odd syntax and, after all, a library is often only as good as its public interface. So we must find a better way. Luckily, **constexpr** variables cost nothing and one can use templates with **constexpr** functions. Therefore, rather than writing:

```
        typename Clear::Type{}
```

we can declare a **constexpr** function to do that for us:

```
template
constexpr typename Clear::Type clear(T){ return {}};
```

Kvasir makes heavy use of this “constexpr meta function” pattern to improve public interface syntax. Functions like **set()**, **clear()**, **toggle()**, **read()**, **write()**, **blindWrite()**, **readAndClear()**, **push()**, and **pop()** take a bit location (sometimes only those with certain access policies or field sizes) and return an **Action**. Others like **atomic()** and **isolated()** take an **Action** and return a modified version. In Kvasir::IO, we use a **PinLocation** template with **contexpr** meta functions like **makeOpenDrain()**, **makeOutput()**, or overloads of **set()**, **clear()**, and **read()**, which take a **PinLocation** and return an **Action**.

The apply() Function

Within a list of **Action**s, the order of execution is not defined because this allows more optimization. In cases where a strict order is desired, one can break the list up and call **apply()** multiple times or use a **sequencePoint**.

The implementation of **apply()** is beyond the scope of this article and will surely evolve over time. If you are interested, have a look at the Kvasir library at kvasir.io. As of this writing, it essentially does the following:

• Tags every **Action** with the index of its possible runtime parameter

• Flattens any sub lists into the main list

• Splits this list of parameters by **sequencePoint**

• Sorts each piece by register address

• Merges reads and writes to the same address wherever possible

• Preforms any other known optimizations

• Masks, merges and passes eventual runtime parameters to their respective **Actions**

• Executes each (merged) **Action**

• Collects the results of reads

• Stores and returns the results of reads into a tuple if necessary

• In case of only one return, the value is masked appropriately and properly typecasted

• Returns the tuple or read value if needed

If you can think of other possible optimizations or handy features, please comment, contact me, or fork on

ıub. We are currently working on optimizations that take advantage of the **LDM** and **STM** (load multiple l store multiple) commands, which the optimizer is often not allowed to do in hand-coded C due to **volatile** constraints.

It should be noted that at the time of this writing, the Kvasir library is still young and therefore may have a volatile public interface, have bugs, or miss certain minor features. In writing about it at this beta stage, I hope to solicit comments and suggestions from other experts before freezing the public interface or making commitments to bad design patterns.

Other Uses of Kvasir::Register

General-Purpose IO

Configuring GPIO pins is an easy task for Kvasir::IO. As mentioned above, Kvasir provides the concept of a **PinLocation**, which the user can easily create with a factory function as well as a bunch of **Action** factories that take a **PinLocation** as a parameter:

```
using namespace Kvasir::IO;
constexpr auto alarmLed = pinLocation(port0,pin4);
constexpr auto alarmHorn = pinLocation(port0,pin8);
//configuration
apply(makeOutput(alarmLed, alarmHorn),makeOpenDrain(alarmHorn));
//use
apply(toggle(alarmLed),clear(alarmHorn));
```

Thread Safety

One may think that when not using a scheduler there is no need for thread safety. However, when different interrupt priorities access the same SFR, the potential exists for a race condition as is commonly the case with different threads. If a thread is interrupted in the middle of a read-modify-write cycle and the same register is modified in the interrupting thread or ISR, the modification will be lost even if it only affects unrelated bits. There are essentially three possible ways to solve this problem:

1. Modify all **Actions** to be atomic except for those in the ISR with the highest priority.

2. Modify all **Actions** to use 8- or 16-bit access mode and only access a specific 8- or 16-bit piece at one priority

3. Disable interrupts or introduce other locking mechanisms.

Option 1 is only possible in cases where we can either use bit banding or when we modify all of the writable bits (excluding read only and reserved) in an 8-, 16-, or 32-bit section. We can modify an **Action** or multiple **Actions** to be atomic using the atomic() modifier function. If the **Action**, or **Action**s, passed to **atomic()** cannot be made into an atomic **Action** (after merging), the library will issue a static assertion (plain text error message at compile time).

Option 2 is actually more applicable then one might think. Of all the bit fields in special function registers, over half of them are either reserved, never modified, or only modified during startup. The remaining fields are often loosely ordered by function, which typically matches how one would divide them between threads or ISR priorities. Thus, chances are high that if a bit field is larger than 1 bit (in which case bit banding is probably possible), it doesn't share a byte with another bit field that's modified at a different priority. Kvasir provides an **Action** modifier function called **isolated()**, which will constrain SFR access to the narrowest possible width.

:ion 3 should really only be a last resort, since it has run-time cost and cannot be done automatically for the r. **Kvasir::Atomic** will eventually provide more tools for this, but at the time of this writing, this module is too young.

Merging and Managing Initialization

As mentioned earlier, many special function registers are modified only at startup. Because of potential race conditions or initialization order bugs, it would be nice to increase static checking here as well. If we choose to group different parts of our program into modules or objects (i.e., classes), we may think of this as part of the constructor.

I argue that actually using a constructor is false, since the SFRs in question are not part of the class and are actually often shared with other classes. This is a special architecture that we don't encounter during classical application development. On the other hand, it's good design to group code into modules where possible. It's also a unique situation in the embedded world, where we would like to have the majority of initialization done before interrupts are turned on.

Using Kvasir::Register, we can express this initialization in the form of **Action**s. But in this case, we don't want to apply them at the site of their definition. We would rather save them in a **constexpr** variable named **init**, which is publicly accessible:

```
using namespace Kvasir::IO;
class ArmageddonWarning {
static constexpr auto alarmLed = pinLocation(port0,pin4);
static constexpr auto alarmHorn = pinLocation(port0,pin8);
public:
//configuration
static constexpr auto inti =
list(makeOutput(alarmLed, alarmHorn),
makeOpenDrain(alarmHorn));
//use
void warn(){
apply(toggle(alarmLed),set(alarmHorn));
}
}
```

We must pass all of our objects to the variadic **KVASIR_START()** macro, which will extract all **init** lists, merge them, and apply them all before interrupts are enabled or main is entered. This is handy for detecting initialization clashes; if two objects modify the same bit field with different values, you get a compiler error (no more initialization order bugs). It's also considerably more efficient—instead of 20 different modules all turning on power, clock, etc., it needs only one merged read-modify-write. Using merged initialization can shrink a small program more than 10-fold compared with LPCOpen or mbed!

Source URL: http://electronicdesign.com/dev-tools/evolution-special-function-register-abstraction