Representing and Manipulating Hardware in Standard C and C++
ESC-243

Dan Saks

Introduction

This paper discusses the basics of hardware manipulation using C and C++. It focuses on the common idioms for controlling devices that you can write in Standard C or Standard C++.

Both C and C++ provide the following features that aid embedded programming:

- bitwise operators and bitfields for packing data and manipulating individual bits in hardware registers
- the const qualifier for identifying ROMable data
- the volatile qualifier for identifying objects, such as memory-mapped device registers, that may change due to events outside a program's control

C++ does not add any features specifically for embedded programming, but classes can be extremely helpful in hiding messy hardware details behind cleaner interfaces.

Touching Hardware

Nearly all computer architectures use at least one of the following methods for accessing the hardware of input/output devices:

- *Memory-mapped addressing* maps device registers into the conventional data space. To a C or C++ programmer, memory-mapped registers look much like ordinary data. Storing into a memory-mapped device register sends commands or data to a device. Reading from a memory-mapped device register retrieves status information or data from a device. For example, this is the approach used in the Motorola 68K processor family.
- *Separate device addressing* maps control and data registers into a separate (often small) data space. Separate addressing is similar to memory-mapping except that programs must use special instructions, such as the **`in`** and **`out`** instructions on the Intel x86 processor family, to move data to or from the device registers. Neither the C nor C++ standard says anything specifically about separate addressing, so it's almost impossible to write code for such devices without using non-standard language or library features, or assembly code.

Standard C and C++ are better at handling memory-mapped devices, so we will explore that first.

Representing Hardware Device Registers

A given hardware device often communicates through a collection of one or more device registers. Some registers communicate control and status information; others communicate data. A given device may use separate registers for input and output. Some registers might occupy just a byte; others may occupy a word or more.

The simplest representation for a device register is as an object of the appropriately sized and signed integer type. For example, you might declare a one-byte register as a **`char`** or a two-byte register as a **`short int`**. Then you can receive data from a memory-mapped device by reading a value from its input data register, and send data to the device by storing a value into its output data register.

A typical control/status register is really not an integer-valued object – it's a collection of bits. One bit may indicate that the device is ready to perform an operation, while another bit might indicate whether interrupts have been enabled for that device. A given device might not use all the available bits in its control/status register.

Using the bitmask approach to manipulate control/status registers, you use symbolic constants to represent masks for isolating bits in a register. Then you use those symbols in combination with bitwise operators to set, clear and test bits in the register.

For example, here's the bitmask representation for a simple 16-bit control/status register, assuming a **`short int`** occupies 16-bits:

```
typedef short int control;
#define ENABLE 0x0040       /* bit 6: enable interrupts */
#define READY  0x0080       /* bit 7: device is ready */
```

Neither Standard C nor Standard C++ lets you declare a variable so that it resides at a specified memory address. Rather, you access a device register by declaring a pointer whose value is the specified address. For example, a program can define:

```
control *const pcr = (control *)0xFF70;
```

which uses a cast expression to initialize **`pcr`** to point to the address of a memory-mapped control/status register. Although Standard C and C++ both allow cast expres-

sions that convert integers to pointers (such as the one above), the exact behavior of such casts vary across platforms.

C++ offers an alternative form for the cast syntax called a **reinterpret_cast**, as in:

```
control *const pcr = reinterpret_cast<control *>(0xFF70);
```

Of the two styles for casts, **reinterpret_cast** is the preferred one in C++.

Once **pcr** points to a memory-mapped control/status register, the program can communicate with the device by testing or setting the value of the register via **pcr**. For example,

```
*pcr &= ~ENABLE;              /* clear enable bit */
```

clears the control register's enable bit, so the device will not trigger interrupts. A loop such as:

```
while ((*pcr & READY) == 0) /* wait until ready */
    ;
```

repeatedly polls (tests) the control register's ready bit until that bit is non-zero. This loop makes the program wait until the device completes whatever it's doing.

In C, you can make a control register look more like an object by using a macro as follows:

```
#define cr (*pcr)
```

Then you can manipulate the device register by writing expressions such as:

```
cr &= ~ENABLE;
```

In C++, you can use a reference (rather than a macro), as in:

```
control &cr = *pcr;
```

Again, then you can write expressions such as:

```
cr &= ~ENABLE;
```

Many devices use a control/status and a data register in tandem. In that case, using a struct is a good way to show the close relationship between the registers. The declarations for the device registers might look like:

```
typedef short int control;
typedef short int data;

typedef struct port port;
struct port
    {
    control cr;
    data dr;
    };

port *const p = (port *)0xFF70;
```

The **typedef**:

```
typedef struct port port;
```

defines **port** as a type name that's equivalent to **struct port**. This allows you to refer to the structure type **port** without also writing the keyword **struct** in front of it. In C++, a structure, union or enumeration name is automatically a type name, so you don't need the **typedef** in C++. But it doesn't hurt. For the remainder of this paper, I will assume for convenience that a structure, union or enumeration name is a type name, even in C.

A bi-directional device (supporting both input and output) may use a pair of registers for input and another pair for output. The declaration for such a pair might look like:

```
struct ioport
    {
    port in, out;
    };
```

Using the declaration above,

```
ioport *const piop = (ioport *)0xFF70;
```

declares **piop** to point to an **ioport**, and:

```
piop->out.cr &= ~ENABLE;
```

disables output interrupts for that port. An assignment such as:

```
piop->out.dr = c;
```

sends the value of character **c** to the output device. It immediately clears the ready bit in the output control/status register to indicate that the device is busy. A loop such as:

```
while ((piop->out.cr & READY) == 0)
    ;
```

or:

```
while (!(piop->out.cr & READY))
    ;
```

waits until the output device is ready to receive another character. A function such as:

```
void put(char const *s, ioport *p)
    {
    for (; *s != '\0'; ++s)
        {
        while ((p->out.cr & READY) == 0)
            ;
        p->out.dr = *s;
        }
    }
```

sends the characters from null-terminated string **s** to the output device controlled by ***p**.

The Volatile Qualifier

The previous function might not work exactly as expected. Here's an example that illustrates why.

Using the previous declaration for **ioport**, the following sequence of code writes **'\r'** (carriage return) and **'\n'** (newline or line feed) to the output device of an **ioport** (based on Plauger [1988]):

```
ioport *const p = (ioport *)0xFFFC4;
...
while (!(p->out.cr & READY))
    ;
p->out.dr = '\r';
while (!(p->out.cr & READY))
    ;
p->out.dr = '\n';
```

This code is supposed to wait until the output device's **READY** bit indicates that the device is ready to perform an operation. Then it writes **'\r'** to the device's data register. The code waits again until the device is ready to perform another operation. Then it writes **'\n'** to the device's data register.

From this code, the compiler has no way of knowing that **p->out.cr** is actually a device register whose value changes spontaneously in response to external events such as a change in state for a peripheral device. The compiler's optimizer might therefore conclude that the value of the **READY** bit in **p->out.cr** never changes. It's always 1 or it's always 0.

If the **READY** bit never changes, then there's no need to test the loop condition more than once. The program can test the bit once and then either loop forever or skip the loop entirely. Thus, the compiler's optimizer can transform:

```
while (!(p->out.cr & READY))
    ;
```

into:

```
if (!(p->out.cr & READY))
    for (;;)
        ;
```

In fact, the compiler can transform both loops in the same way. After this optimization, the driver code becomes:

```
if (!(p->out.cr & READY))
    for (;;)
        ;
p->out.dr = '\r';
if (!(p->out.cr & READY))
    for (;;)
        ;
p->out.dr = '\n';
```

Again, the compiler can't see any code that changes the **READY** bit. It concludes that the **READY** bit is either always set or always clear. Consequently:

- If the **READY** bit is always set, the program always skips the first loop. When the execution reaches the second loop, the bit is still set and so the program skips the second loop as well.

- If the **READY** bit is always clear, the program enters the first loop and never leaves. It never gets a chance to execute the second loop.

In either case, the program never executes the second loop.

Therefore, the optimizer can eliminate the second if-statement entirely. The optimized driver code becomes:

```
if (!(p->out.cr & READY))
    for (;;)
        ;
p->out.dr = '\r';
p->out.dr = '\n';
```

Now, as far as the compiler can see, the first assignment writes **'\r'** into a location, only to have the next assignment overwrite the same location with **'\n'**. Therefore, only the last assignment is worth keeping. The final "optimized" code looks like:

```
if (!(p->out.cr & READY))
    for (;;)
        ;
p->out.dr = '\n';
```

It does the wrong thing, but much more efficiently than the original code!

The way to prevent this overly aggressive optimization is the use the volatile qualifier, as in:

```
ioport volatile *const p = (ioport *)0xFFFC4;
```

This declares **p** as a "**const** pointer to a **volatile ioport**". The volatile qualifier indicates that an object, such as a memory-mapped device register, may change even though the program didn't change it. The volatile qualifier prevents the compiler from "optimizing away" references to an object that seems unchanging, but doesn't inhibit other optimizations.

In a declaration such as:

```
ioport volatile *const p = ... ;
```

volatile is not part of the type of **ioport**. This is appropriate only if some **ioport**s are non-volatile.

If volatility is inherent in **ioport**s, **volatile** should be part of the **ioport** type. It may be that only some registers within an **ioport** are volatile:

```
struct ioport
    {
    control volatile cr;
    data dr;
    };
```

More likely, every member of an **ioport** will be volatile. In that case, you must declare each member so, as in:

```
struct ioport
    {
    control volatile cr;
    data volatile dr;
    };
```

If you want to declare the entire **ioport** as a volatile type, this won't work:

```
volatile struct ioport
    {
    control cr;
    data dr;
    }; // error: missing declarator
```

To declare the entire **ioport** as a volatile type, you should use a **typedef**, as in:

```
typedef struct /* no tag */
    {
    control cr;
    data dr;
    } volatile ioport;
```

Just as pointer conversions can add but not remove a const qualifier, they can add but not remove a volatile qualifier:

```
T *p;
T volatile *pv;
pv = p;         // OK
p = pv;         // error
p = (T *)pv;    // OK
```

Believe it or not, the "new style" cast for removing **volatile** from a pointer type is **const_cast**. There is no such thing as a **volatile_cast**:

```
T *p;
T volatile *pv;
p = const_cast<T *>(pv);       // yes
p = volatile_cast<T *>(pv);    // no
```

You must preserve the volatility of arguments in parameter passing as well:

```
ioport volatile *const piop = ...;
void put(char const *s, ioport *p);

put("...", piop);   // error
```

The call is an error because the compiler can't convert **piop** from "pointer to **volatile ioport**" into "pointer to **ioport**". One way to eliminate the error is to declare **put**'s second parameter as **ioport volatile *p**, as in:

```
void put(char const *s, ioport volatile *p);
```

Objects can be both const and volatile, meaning "I promise I won't change it, but I'll assume that something else might." This is appropriate for read-only device registers.

Representing Hardware Registers as Bitfields

Bitfields offer an approach that's more elegant than bitmasks for modeling hardware registers. Rather than declare the register type as an alias for some integer type, you declare the register as a structure with each bit as a named bitfield member. Then you can set, clear and test bits in registers by manipulating the named members almost as if they were objects.

For example, you can declare the control register and port types as:

```
struct control
    {
    unsigned int /* unused */ : 6;
    unsigned int enable : 1;
    unsigned int ready : 1;
    unsigned int /* unused */ : 8;
    };
typedef short int data;
```

```
struct ioport
    {
    control volatile cr;
    data volatile dr;
    };

ioport *const piop = ...;
```

Using these declarations,

```
piop->out.cr.enable = 0;
```

disables output interrupts for that port and:

```
while (piop->out.cr.ready == 0)
    ;
```

waits until the output device is ready.

If you prefer, you can even think of each single-bit bitfield as a boolean, and write:

```
piop->in.cr.enable = false;
```

to disable output interrupts and:

```
while (!piop->in.cr.ready)
    ;
```

to waits until the output device is ready.

In C++, you can use the predefined boolean constants **`false`** and **`true`**. In C99, you can get declarations for these symbols by including **`<stdbool.h>`**. Using earlier dialects of C, you must define these symbols yourself.

A word of caution: Neither C nor C++ specifies whether a compiler must allocate bits starting with the most- or the least-significant bit. Therefore, although this technique is generally applicable across platforms, the code itself is not likely to be portable.

Mixing Hardware Register Metaphors

You can sometimes get the most compact and efficient code by employing different representations for a single register. For example, suppose **pcr** points to a 16-bit control register and the **READY** bit happens to be the high-order bit in the register. You could define the bit mask for **READY** as:

```
#define READY 0x8000
```

Then you can test the **READY** bit using:

```
if (*pcr & READY)
```

If the control register has a signed integer type then:

```
if (*pcr < 0)
```

does the same thing, often with smaller, faster machine instructions.

Some hardware designers have a habit of placing the most often used bit of a control/status register as the sign bit so that software can use a signed comparison to test that bit. This encourages you to treat a control/status register as a signed integer sometimes, and as a set of bits at other times.

A union can provide alternative views of a single device register:

```
struct byte_pair
    {
    signed char hi, lo;
    };

union control
    {
    short int word;
    byte_pair byte;
    };
```

This union assumes that **sizeof(short int)** is 2, which is not true on all platforms.

Anyway, using the union, here's a (possibly) efficient way to test the high-order bit of the low-order byte of a register:

```
control volatile *const pcr = (control *)0xFF70;
...
if (pcr->byte.lo < 0)
    ...
```

Again, you must be careful to adapt these coding techniques to match the byte ordering of your architecture. For example, the union above overlays **hi** with the high-order byte of **word** only if the machine is "big-endian". On a "little-endian" machine, you should reverse the order of the bytes in the **byte_pair**, as in:

```
struct byte_pair
    {
    signed char lo, hi;
    };
```

Once you get the byte ordering correct, then for control register **cr**, the expressions **cr.byte.hi** and **cr.byte.lo** refer to the high- and low-order bytes of the register, respectively. **cr.word** refers to the entire register.

Remember, no matter how you mix representations, you must rely on implementation-dependent storage representations.

Unless you really need to save a few bytes or a few nanoseconds, you're better off writing the code as clearly and simply as possible, but it's nice to know the alternatives. With C++, you can have the best of both worlds – speed and clarity – if you package it properly...

Hiding Messy Details in a C++ Class

A C++ class can hide most of the messy details described above, such as the choice of bitmasks vs. bitfields (including any bitmask constants) or dependency on implementation-defined byte ordering. For example, you might define a class **control_status** for control/status registers as:

```
class control_status
    {
public:
    void enable();
    void disable();
    bool enabled() const;
    bool ready() const;
```

```
private:
    enum { ENABLE = 0x0040, READY = 0x0080, ... };
    union
        {
        short int word;
        byte_pair byte;
        } u;
    };
```

This class represents a control/status register as a union wherein you can view the register as either a (**signed**) **short int** or a pair of **signed char**. The class assumes that **sizeof(short int)** is 2, and that the low-order byte of a **short int** has the same address as the entire **short int**. Only the union member **u** occupies storage. The enumeration definition defines a type and constants, but no data that occupies space in **control_status** objects. Thus, the size of a **control_status** object is just the size of its **u** member, which is **sizeof(short int)**.

This example defines the constants **ENABLE** and **READY** as enumeration constants rather than as the macros used in earlier examples. The advantage that enumerations have over macros is that:

- Macros ignore the normal scope rules of C and C++. That is, they ignore **{** and **}**, so macros might substitute in places you don't want them to. Enumerations observe the scope rules.
- Some compilers don't preserve macro names among the symbols they pass on to their debuggers.

Class **control_status** defines its member functions as **inline** to avoid run-time performance penalties:

```
inline
void control_status::enable()
    {
    u.word |= ENABLE;
    }

inline
void control_status::disable()
    {
    u.word &= ~ENABLE;
    }
```

```
inline
bool control_status::enabled() const
    {
    return u.word & ENABLE;
    }

inline
bool control_status::ready() const
    {
    return u.byte.lo < 0;
    }
```

As is often the case with unions, we are not interested in the union itself, but rather only in the members in the union. That is, we never refer to **control_status** member **u** by itself. We always refer to **u.word** or **u.byte**. C++ lets you eliminate the name **u** by using an *anonymous union*, as in:

```
class control_status
    {
public:
    ...
private:
    ...
    union
        {
        short int word;
        byte_pair byte;
        } /* no name here */;
    };
```

This, in turn, simplifies the member functions. For example, using the anonymous union, the definition for **control_status::ready** simplifies from:

```
inline
bool control_status::ready() const
    {
    return u.byte.lo < 0;
    }
```

to:

```
inline
bool control_status::ready() const
    {
    return byte.lo < 0;
    }
```

Class **control_status** restricts access to hardware so that the rest of the program can manipulate that hardware only in proscribed ways. That is, all manipulation of a **control_status** must be through public member function calls, as in:

```
struct port
    {
    control_status csr;
    data dr;
    };

port *const p = ...;
...
p->csr.disable();
while (!p->csr.ready())
    ;
```

If **control_status** objects can be declared volatile, as in:

```
struct port
    {
    control_status volatile csr;
    data volatile dr;
    };
```

then **control_status** member functions such as **disable** must be declared **volatile**, as in:

```
class control_status
    {
public:
    void enable() volatile;
    void disable() volatile;
    bool enabled() const volatile;
    bool ready() const volatile;
    ...
    };
```

The keyword **volatile** also must appear in the corresponding member function definition:

```
inline
void control_status::enable() volatile
    {
    u.word |= ENABLE;
    }
```

A *volatile member function* treats the object it applies to (***this**) as a volatile object. You cannot apply a member function to a volatile object unless the member function is a volatile member function. A member function can be both const and volatile.

Declaring all the public members as volatile can be a nuisance. The simpler alternative is to declare the members of **control_status**' anonymous union as volatile, as in:

```
class control_status
    {
    ...
    union
        {
        short int volatile word;
        byte_pair volatile byte;
        };
    };
```

This builds volatility right into the **control_status** type so that all **control_status** objects exhibit volatile semantics, even those not declared volatile.

You can use structures and macros in C to organize hardware manipulation as if you were using a class. However, structures and macros cannot prevent erroneous operations on a device as well as a class can. Classes leave the program with fewer ways to misuse the hardware.

Write-Only Registers

Some machines have control/status registers that are write-only. Write-only registers pose a problem for functions such as:

```
inline
void control_status::enable() volatile
    {
    word |= ENABLE;
    }
```

The statement in the function body is shorthand for:

```
    word = word | ENABLE;
```

The reference to **word** on the right-hand side of the **=** is a read access. If **word** is write-only, this function won't work properly.

For a write-only register, the program must maintain a separate read-write copy of the register. The place to keep that read-write copy is in a class.

The previous definition for **control_status** provides a class for objects that overlay bus addresses corresponding to hardware registers. The following **control_status** class uses a different approach that's more appropriate when dealing with write-only registers. Here, a **control_status** object is not the hardware itself, but an object in memory that points to and monitors a register at a specified bus address. The code looks like:

```
union csr_type
    {
    short int word;
    byte_pair byte;
    };

class control_status
    {
public:
    typedef size_t address_type;
    control_status(address_type);
    void enable();
    ...
private:
    ...
    csr_type image;
    csr_type volatile *const actual_csr;
    };
```

Here, member **image** is a copy of the register pointed to by member **actual_csr**. The **control_status** constructor:

```
control_status::control_status(address_type at):
    actual_csr(reinterpret_cast<csr_type *>(at))
    {
    image.word = 0;
    }
```

sets **actual_csr** to point to the actual hardware register and clears the **image**. The corresponding **enable** function might look something like:

```
void control_status::enable()
    {
    image |= ENABLE;
    *actual_csr = image;
    }
```

Notice that the **control_status** member functions no longer need to be **volatile**. **control_status** objects have a member that's a pointer to a **volatile csr_type**, but the **control_status** object itself is not **volatile**.

Separate Device Addressing

When using separate device addressing, you define the data structures for device registers pretty much as above; however, you cannot move data to and from the device registers by simple assignment. Since the C and C++ standards do not support separate addressing, you must rely on language extensions or assembly to move data to and from the registers. Here are some examples of what your compiler might provide (from Plauger [1993]):

- special functions:

```
char c = in(0x40);
```

- inline assembly code:

```
char c;
_asm in, 0x40
_asm mov y, al
```

- unique address spaces and addressing modes:

```
@port char InBuf = 0x40;
char c = InBuf;
```

Again, your best bet is to hide messy details in as little code as possible, preferably in member functions of a C++ class.

In Summary

- You can control memory-mapped devices using only standardized language features (but they will still have platform-specific behavior).
- Separate device addressing is outside the standard. You can do it only by using language extensions or assembly code.
- Whatever you do, isolate language and hardware dependencies inside abstract types, such as C++ classes.

Acknowledgments

Some the material in this paper comes from articles that appeared in the *Programming Pointers* column of *Embedded Systems Programming* magazine, written and copyright © 1998 by Dan Saks. Also, thanks to P.J. Plauger for sharing his notes on this material.

References

- Plauger [1988]. P.J. Plauger. "Touching Memory: Standard C Makes The Act More Precise", *The C Users Journal*, Vol. 6, No. 4, May, 1988.
- Plauger [1993]. P.J. Plauger. "*Manipulating Hardware in C and C++*", Proceedings of the Embedded Systems Conference East, April, 1993. Miller Freeman.