Methods of Hardware Access in C++

Duane Strong

duanes@strongenging.com

12/29/2004

Introduction

This paper evaluates different methods for manipulating memory mapped hardware in a C++ environment. Alternatives are discussed and the resulting assembly code is examined for efficiency. GCC Intel x86 disassembly is shown in the examples.

Requirements

In an embedded system hardware register access is usually provided by direct addressing in the processors memory map. These systems require efficient access to these locations and typically involve individual bit manipulations in each register. In many cases blocks of registers are semantically bound together and an interface that reflects this grouping is desirable.

Using the #define for everything method

In this method the addresses of hardware registers are represented by preprocessor symbols containing a typecast of a constant integer address to the proper type for the register.

```
#define MBAR_ADDRESS (0x10000000)
#define UART1_BASE_ADDRESS    (MBAR_ADDRESS + 0x100) /* UART 1 base */
#define UART_UMR_PTR        ((volatile unsigned char *)(UART1_BASE_ADDRESS+0x00))
#define UART_USR_UCSR_PTR   ((volatile unsigned char *)(UART1_BASE_ADDRESS+0x04))
#define UART_UCR_PTR        ((volatile unsigned char *)(UART1_BASE_ADDRESS+0x08))
```

Individual bits in registers are represented by more preprocessor symbols. These symbols are combined by hand using bitwise logical operators.

```
/* UART Mode Register bits */
#define UMR1_OP             (0x04)    /* odd parity */
#define UMR1_BPC(n)         (n-5)     /* number of bits per char 5,6,7,8 */

/*    UART Command  Register bits */
#define UCR_RESET_RX        (0x20)    /* reset receiver */

/*    UART Status Register bits */
#define UART_STS_OVERRUN_ERR        (0x10)
```

Using these preprocessor symbols as pointers affords the register access.

```
*UART_UMR_PTR = UMR1_BPC(8) | UMR1_OP;
*UART_UCR_PTR = UCR_RESET_RX;
if( *UART_USR_UCSR_PTR & UART_STS_OVERRUN_ERR ) {
    do something about overrun error;
}
```

This approach generates very efficient code.

```
        movb    $7, 268435712
        movb    $32, 268435720
        movzbl  268435716, %eax
        testb   $16, %al
        je      L2
         # basic block 1
        movb    $1, %bl
L2:
```

This approach suffers from a few weaknesses. First the use of the preprocessor is undesirable for the following reasons:

- The symbols have global scope in the module and therefore necessitates the practice of pre-pending all symbols with some kind of group name.
- The symbols are not a proper part of the C++ language and therefore have no type and are not conveyed to the debugger.
- Numerous known problems with preprocessor syntax such as unintended concatenation requiring copious parenthesis.

Secondly there is no semantic grouping of related elements. Groups of registers are still distinct entities, and bit definitions are not bound to particular registers allowing their use on the wrong register.

Using structures or classes and enumerations

As an evolution of the previous method, structures may be used to group semantically related registers. This reduces dependence on the preprocessor, and imparts type information to each register symbol. It also establishes a name scope in which the register names are bound.

```
struct UartRegs
{
    volatile unsigned char ucUmr;       // mode registers 1/2 (rw - flips to other reg)
    unsigned char skip[3];
    volatile unsigned char ucUsrUcsr;   // status register(r) + clock select register (w)
    unsigned char skip1[3];
    volatile unsigned char ucUcr;       // command register (wo)
    unsigned char  skip2[3];
        .
        .
        .
}
```

This technique comes with an important caveat. Compilers are allowed to insert extra space between structure elements that are invisible to the programmer to allow for more efficient alignment of the elements in the data structure. These extra elements would cause the structure to incorrectly map to the hardware and must not be allowed. Compiler "pack" options or pragmas can be used to eliminate them. Usually the hardware designer will create the hardware addressing of registers on native alignment boundaries and the subsequent structure mapping will end up natively aligned without extra space inserted. Nevertheless it is important to check this in all cases.

While not technically necessary, changing the struct to a class yields a more familiar context to add public and protected sections, enumerations scoped inside the class, and inline macros to replace preprocessor macros.

The use of enumerations for integral constants is preferred over `const unsigned` because enumerations provide a one step declaration and assignment, and there are cases where the compiler can not eliminate creating storage for the const since the optimizer can not always tell if no code is going to take the address of the constant, whereas enumerations are always only used as immediate operands because a reference to an enum is not allowed.

```
class UartRegs
{
public:
    volatile unsigned char ucUmr;       // mode registers 1/2 (rw - flips to other reg)
    unsigned char skip[3];
    volatile unsigned char ucUsrUcsr;   // status register(r) + clock select register (w)
    unsigned char skip1[3];
    volatile unsigned char ucUcr;       // command register (wo)
    unsigned char  skip2[3];
        .
        .
        .

    enum tUmrBits {
        // UART Mode Register bits
        OP      = 0x04,     // odd parity
    };
    static inline unsigned char BPC( int n ) { return n-5; }

    enum tUsrUcsrBits {
        // UART Status Register bits
        ERR_MASK        = 0xF0,     // mask for error bits
        RCVD_BREAK      = 0x80,
        FRAMING_ERR     = 0x40,
        PARITY_ERR      = 0x20,
        OVERRUN_ERR     = 0x10,
    };

    enum UcrBits {
// UART Command  Register bits
        RESET_RX        = 0x20,     // reset receiver
        RESET_TX        = 0x30,     // reset transmitter
        RESET_ERR       = 0x40,     // reset error status
    };

};


#define UART_PTR_1        ((volatile UartRegs *)( MBAR_ADDRESS + 0x100))
```

Using these structure fields off the base address macro affords the register access.

```
// class and enum method
UART_PTR_1->ucUmr = UartRegs::BPC(8) | UartRegs::OP;
UART_PTR_1->ucUcr = UartRegs::RESET_RX;

if( UART_PTR_1->ucUsrUcsr & UartRegs::OVERRUN_ERR ) {
    do something about overrun error;

}
```

This approach generates very efficient code, the same as before as long as compiler optimizations (-O) are enabled.

```
         # basic block 2
        movb    $7, 268435712
        movb    $32, 268435720
        movzbl  268435716, %eax
        testb   $16, %al
        je      L4
         # basic block 3
        movb    $1, %bl
L4:
```

To eliminate the last remnant of the preprocessor, and bind the symbol for the address of the register block into the scope of the class the following constant is added to the class:

```
    static UartRegs * const PTR_1; /* UART 1 base address */

};

UartRegs * const UartRegs::PTR_1 = (UartRegs *) (MBAR_ADDRESS + 0x100);
```

Using these structure fields off the base address class constant affords the register access.

```
// const pointer method
UartRegs::PTR_1->ucUmr = UartRegs::BPC(8) | UartRegs::OP;
UartRegs::PTR_1->ucUcr = UartRegs::RESET_RX;

if( UartRegs::PTR_1->ucUsrUcsr & UartRegs::OVERRUN_ERR ) {
    do something about overrun error;
}
```

This approach generates the same as before , with the addition of an unused constant in the text section. This is due to the explanation above requiring constants to allow for the possibility of code taking the address of the constant.

```
.globl __ZN8UartRegs5PTR_1E
        .text
        .align 4
__ZN8UartRegs5PTR_1E:
        .long   268435712
.
.
.
         # basic block 4
        movb    $7, 268435712
        movb    $32, 268435720
        movzbl  268435716, %eax
        testb   $16, %al
        je      L6
```

```
         # basic block 5
        movb    $1, %bl
L6:
```

Additionally this requires the definition of the constant to only be present in a single translation unit, or a multiply defined global error will result. Alternatively the constant can be left out of the class and declared static such that it only has module scope. This no longer binds the symbol to the class name

```
static UartRegs * const CONST_UART_PTR_1 = (UartRegs *) (MBAR_ADDRESS + 0x100);
```

Using these structure fields off the base address module constant affords the register access.

```
// const static pointer method
CONST_UART_PTR_1->ucUmr = UartRegs::BPC(8) | UartRegs::OP;
CONST_UART_PTR_1->ucUcr = UartRegs::RESET_RX;

if(CONST_UART_PTR_1->ucUsrUcsr & UartRegs::OVERRUN_ERR ) {
    do something about overrun error;
}
```

This approach generates the same code as before , with the same unused constant in the text section. However this time it is at least not global and can be re-defined in each module.

```
_ZZ4mainE16CONST_UART_PTR_1:
        .long   268435712
.
.
.
         # basic block 6
        movb    $7, 268435712
        movb    $32, 268435720
        movzbl  268435716, %eax
        testb   $16, %al
        je      L8
         # basic block 7
        movb    $1, %bl
```

Using bitfields in classes

As an additional aid to the programmer, individual bits in registers can be broken out symbolically into bitfields.

```
class UartBits
{
public:
    // mode register
    volatile unsigned char ucBits    : 2;
    volatile unsigned char ucOddP    : 1;
    volatile unsigned char ucForceP  : 1;
    volatile unsigned char ucNp      : 1;
    volatile unsigned char ucBlkErr  : 1;
    volatile unsigned char ucIntFf   : 1;
    volatile unsigned char ucAutoRts : 1;
```

```
    unsigned char skip[3];
    // status register
    volatile unsigned char ucBreak   : 1;
    volatile unsigned char ucFrameEr : 1;
    volatile unsigned char ucParErr  : 1;
    volatile unsigned char ucOverRun : 1;
    volatile unsigned char ucTxEmpty : 1;
    volatile unsigned char ucTxReady : 1;
    volatile unsigned char ucFFull   : 1;
    volatile unsigned char ucRxReady : 1;
    unsigned char skip1[3];
    // command register
    volatile unsigned char ucAutoBd  : 1;
    volatile unsigned char ucCommand : 3;
    volatile unsigned char ucTxDisab : 1;
    volatile unsigned char ucTxEnab  : 1;
    volatile unsigned char ucRxDisab : 1;
    volatile unsigned char ucRxEnab  : 1;
    unsigned char  skip2[3];

    static inline unsigned char BPC( int n ) { return n-5; }

   enum UcCommandBits {
// UART Command  Register bits
        RESET_RX        = 0x2,     // reset receiver
        RESET_TX        = 0x3,     // reset transmitter
        RESET_ERR       = 0x4,     // reset error status
    };


};

// Base addresses as a macro
#define UARTBITS_PTR_1        ((volatile UartBits *)( MBAR_ADDRESS + 0x100))
```

Using these bitfields off the base address macro affords the register access.

```
// (partial) bitfield method
UARTBITS_PTR_1->ucBits = UartBits::BPC(8);
UARTBITS_PTR_1->ucOddP = 1;
UARTBITS_PTR_1->ucCommand = UartBits::RESET_RX;

if( UARTBITS_PTR_1->ucOverRun ) {
       do something about overrun error;
}
```

This approach generates less efficient code than the other methods.

```
         # basic block 8
        movzbl  268435712, %eax
        orb     $3, %al
        movb    %al, 268435712
        movzbl  268435712, %eax
        orb     $4, %al
        movb    %al, 268435712
        movzbl  268435720, %eax
        andb    $-113, %al
        orb     $32, %al
        movb    %al, 268435720
        movzbl  268435716, %eax
        andb    $16, %al
        testb   %al, %al
        je      L10
         # basic block 9
        movb    $1, %bl
```

The bitfield approach has a number of problems. First bitfields are not portable. It is left to each compiler to decide how the bits in the bitfields are to be laid out. The first bit defined could be the most significant or the least. In this particular case they are least to most significant. Furthermore alignment issues as with structures exist but more so. Compilers can allocate bit groups into bytes as they see fit for optimization.

Secondly the code generated is not efficient. In the example setting all bits in a register to a known value was left out for clarity and would involve many bit oriented operations where one register access would suffice. The use of unions to make whole register access more efficient exacerbates the portability problem as the mapping of union elements onto one another is also left to the compiler.

Third the level of abstraction does not map to the reality of the hardware. In this simple example there are problems caused by the way this hardware works. The Uart mode register has interesting behavior in that each successive access to the register flips the meaning of the register from mode 1 to mode 2. Individual read-modify-write operations of the bitfield access causes the mode register to flip back and forth. In other cases the programmer may want to check if any of a number of bits are active in a register, to check if any error bits are on for example. In the bitfield case this must be done with successive accesses and bit operations, where as in the whole register approach it can be done in one test.

In summation the bitfield abstraction removes a level of control from the programmer in favor of an abstraction that is not portable and does not map to the hardware reality with good fidelity.

Recommendations

Using the class with enumerations offers benefits without compromising code efficiency. Bitfields offer questionable benefit to the programmer with a cost in code efficiency. Using a class constant pointer to hold the base address of the register bank binds the pointer to the class name, but causes a problem with multiply defined constants and wastes storage (although a very small cost). Using a module static constant as the base pointer does not bind the name to any class and uses up additional space. The #define macro of the base address seems to yield the same benefits as the module static without the wasted space. The recommendation then is to use classes with enumerations, and a #define for the base address.

References

1. Saks, Dan, “Mapping Memory,” Embedded Systems Programming, September 2004, p 49. http://www.embedded.com/shared/printableArticle.jhtml?articleID=26807176
2. Saks, Dan, “Mapping Memory Efficiently,” Embedded Systems Programming, November 2004, p 47. http://www.embedded.com/shared/printableArticle.jhtml?articleID=50900224

3. Saks, Dan, “More Ways to Map Memory,” Embedded Systems Programming, December 2004. http://www.embedded.com/shared/printableArticle.jhtml?articleID=55301821
4. Saks, Dan, “Symbolic Constants,” Embedded Systems Programming, November 2001 p 55. http://www.embedded.com/shared/printableArticle.jhtml?articleID=9900352
5. Saks, Dan, “As Precise as Possible,” Embedded Systems Programming, April 2002, p 43. http://www.embedded.com/shared/printableArticle.jhtml?articleID=9900563
6. Saks, Dan, “Representing and Manipulating Hardware in Standard C and C++,” Embedded Systems Seminar Silicon Valley, August 3-4 2004, Westin Santa Clara, CA.