Chapter 7
Accessing Microcontroller Registers

Microcontroller programming requires efficient techniques for register access. Registers are used to configure the CPU and peripheral hardware devices such as flash access, clocks, I/O ports, timers, communication interfaces (UART, SPI™, CAN [1]), etc. This chapter describes C++ methods that can be used to manipulate microcontroller registers. The focus of this chapter is placed on template methods that provide for efficient, scalable and nearly portable register access.

7.1 Defining Constant Register Addresses

C programmers often define register addresses with a preprocessor `#define`. For example,

```c
// The 8-bit address of portb.
#define REG_PORTB ((uint8_t) 0x25U)
```

The preprocessor symbol `REG_PORTB` represents the 8-bit address of `portb` on our target with the 8-bit microcontroller. We first encountered this register in the LED program of Sect. 1.1. The value of `portb`’s address is `0x25`. The type of the address is `uint8_t`. In addition, the type information is tightly bound to the preprocessor definition with a C-style cast operator. All-in-all, this is a robust register definition in C.

As mentioned in association with the LED program in Sect. 1.10, `portb` can also be manipulated via direct memory access in the C language. For example, the following C code sets the value of `portb` to zero.

```c
// Set portb to 0.
*(((volatile uint8_t*) REG_PORTB) = 0U;
```
In C++ it can be convenient to define register addresses with compile-time constant static integral members of a class type (such as a structure) or using the constexpr keyword. This technique has already been used a few times in this book and is described in greater detail in Sect. 4.10. In particular,

```cpp
namespace mcal
{
    struct reg
    {
        static constexpr std::uint8_t portb = 0x25U;

        // Additional registers
        // ...
    };
}
```

Register addresses can alternatively be defined as compile-time constants with constexpr possibly in a namespace for naming uniqueness. For example,

```cpp
namespace mcal
{
    namespace reg
    {
        constexpr std::uint8_t portb = 0x25U;

        // Additional registers
        // ...
    }
}
```

The mcal::reg structure (or the mcal::reg namespace) can be used to define a variety of microcontroller register addresses. Each register address needed in the program can be included as a compile-time constant. In the mcal::reg structure above, for example, the 8-bit address of portb on our target with the 8-bit microcontroller has a compile-time constant value equal to 0x25.

Using the mcal::reg structure (or alternatively the namespace mcal::reg) it is straightforward to set portb via direct memory access in C++. For instance,

```cpp
// Set portb to 0.
*reinterpret_cast<volatile std::uint8_t*>(mcal::reg::portb) = 0U;
```

As mentioned in Sects. 1.10 and 4.10, compile-time constants are just as efficient as preprocessor #defines, but have superior type information. Compile-time
constants are well-suited for defining register addresses because they require no storage and are available for constant folding. Register addresses defined as compile-time constants can also be used as parameters in C++ templates. This can be used to create highly optimized template classes that can be mapped to the microcontroller’s peripherals resulting in efficient hardware-access code that possesses a high degree of portability. This technique will be shown in the next section and also used for a serial SPI™ driver in Sect. 9.5.

### 7.2 Using Templates for Register Access

Consider the template class below. It is a scalable template class designed for setting the value of a microcontroller register.

```cpp
template<typename addr_type,
         typename reg_type,
         const addr_type addr,
         const reg_type val>
class reg_access
{
    public:
        static void reg_set()
        {
            *reinterpret_cast<volatile reg_type*>(addr) = val;
        }
};
```

The `reg_access` class has four template parameters that specify the characteristics of the microcontroller register. The `addr_type` parameter defines the type of the register’s address. When used with `portb` on our target with the 8–bit microcontroller, for example, the type of `addr_type` is `std::uint8_t`. The `reg_type` parameter defines the physical width of the register. This is also `std::uint8_t` for `portb` on our target with the 8–bit microcontroller.¹ The last two template parameters, `addr` and `val`, define the register’s address and the value that should be written it. These two parameters must be integral compile-time constants.

The `reg_access` template has one static method called `reg_set()`. This function is designed for setting a register at a fixed address with a constant value. For example,

---

¹Note, however, that a register’s width need not necessarily have the same type as its address. One often encounters registers with 8–bit width or 16–bit width on a 32–bit machine, etc.
As in the examples in the previous section, this code also sets the value of the `portb` register to zero. This is accomplished by calling the `reg_set()` function. Notice how this code obtains the address of `portb` from the `mcal::reg` class.

There are several advantages to implementing register access functions in a templated class such as `reg_access`. In particular, `reg_access` offers scalability and portability because it can be used with different register types and microcontroller architectures.

In the code below, for example, a register with a 32-bit address and an 8-bit width is set with an 8-bit value.2

```c++
// Set timer0 mode register tm0ctl0 to zero.
reg_access<std::uint32_t,
           std::uint8_t,
           mcal::reg::tm0ctl0,
           0x00U>::reg_set();
```

In the following code, a register with a 32-bit address and 16-bit width is set with a 16-bit value.

```c++
// Set timer0 compare register tm0cmp0 to 32,000.
reg_access<std::uint32_t,
           std::uint16_t,
           mcal::reg::tm0cmp0,
           32000U>::reg_set();
```

The `reg_set()` function of the `reg_access` class is quite efficient because all the template parameters are compile-time entities. When compiling the sample above, for example, the compiler eliminates the `addr` and `val` template parameters via constant folding and sees in `reg_set()` the following statement.

```c++
*reinterpret_cast<volatile std::uint16_t*>(0xFFFFF694UL) = 32000U;
```

---

2This example and the following one have been taken from code that I once wrote to initialize timer0 for a well-known 32-bit microcontroller.
Since this code is entirely known at compile time, the compiler can optimize it to the best of its ability. In fact, the compiler could potentially substitute a single opcode for the operation if one is available for the CPU architecture and the compiler is capable of recognizing the opportunity to do so.

### 7.3 Generic Templates for Register Access

Based on the `reg_set()` subroutine in the previous section, we can add additional functions such as logic and bit operations to the `reg_access` class. For example, we will now add to the `reg_access` class a function for the logical `or` operator.

```cpp
template<
    typename addr_type,
    typename reg_type,
    const addr_type addr,
    const reg_type val = 0>
class reg_access
{
public:
    static void reg_set()
    {
        *reinterpret_cast<volatile reg_type*>(addr) = val;
    }

    static void reg_or()
    {
        *reinterpret_cast<volatile reg_type*>(addr) |= val;
    }
};
```

The `reg_or()` function is similar to the `reg_set()` function. The only difference is that instead of setting the value with `operator=()`, the logical `or` operator is used. This subroutine can be used for `or-ing` the value of a register at a fixed address with a constant value. In particular,

```c
// Set portb.5 to 1.
reg_access<std::uint8_t, std::uint8_t, mcal::reg::portb, 0x20U>::reg_or();
```
This code is equivalent to

```cpp
*reinterpret_cast<volatile std::uint8_t*>(0x25) |= 0x20;
```

and it performs a bitwise or of `portb` with the 8–bit value `0x20`. This sets `portb.5` on our target with the 8–bit microcontroller to high.

As a final example, we will add a dedicated bit operation to the `reg_access` class. For example,

```cpp
template<typename addr_type, typename reg_type, const addr_type addr, const reg_type val = 0>
class reg_access
{
 public:
  // ...

  static void bit_not()
  {
    *reinterpret_cast<volatile reg_type*>(addr)
      ^= (1 << val);
  }
};
```

The `bit_not()` function performs a bitwise exclusive-or (xor) of a register with a bitmask containing a single bit. Notice that the `val` parameter here is used to create the bitmask from 1 shifted left `val` times.

The `bit_not()` function has the effect of toggling a bit from low to high and vice versa. For example,

```cpp
// Toggle portb.5.
reg_access<std::uint8_t, std::uint8_t, mcal::reg::portb, 5U>::bit_not();
```

This code is equivalent to

```cpp
*reinterpret_cast<volatile std::uint8_t*>(0x25) ^= 0x20;
```
and it performs a bitwise xor of portb with 0x20. This toggles portb.5 on our target with the 8–bit microcontroller from low to high and vice versa. It is the same register manipulation that was introduced in the toggle() function of the led class in the LED program of Sect. 1.1.

So now the reg_access class includes functions for register set, logical or and bitwise xor. It is straightforward to add even more register functions. For example, the class synopsis of an extended reg_access class is shown below.

```cpp
template<typename addr_type,
         typename reg_type,
         const addr_type addr,
         const reg_type val>
class reg_access
{
public:
    static void reg_set() { /* ... */ }
    static void reg_and() { /* ... */ }
    static void reg_or () { /* ... */ }
    static reg_type reg_get() { /* ... */ }
    static void bit_set() { /* ... */ }
    static void bit_clr() { /* ... */ }
    static void bit_not() { /* ... */ }
    static bool bit_get() { /* ... */ }

    static void variable_reg_set(const reg_type)
    {
        // ...
    }
};
```

This version of the reg_access class is contained in the companion code of this book. It has functions for register set, get, various bit operations, etc. In this sense, the reg_access class is a scalable, flexible and generic template that can be used for register manipulation on any microcontroller platform, regardless of the address widths and register types.

Register manipulation code can never be truly portable because the addresses and purposes of registers are specific to a given microcontroller. The reg_access class, however, makes no use of these kinds of microcontroller-specific details. So as long as the microcontroller-specific details are localized somewhere else (such as in something like the mcal::reg structure), the reg_access class remains portable—perhaps as portable as possible for microcontroller register access.
7.4 Bit-Mapped Structures

Microcontroller programmers often use C-style structures with bit-fields to represent bits or groups of bits in a register. This is useful for creating a bit-mapped structure that identically matches the bits in a hardware register. For example, an 8-bit port register can be represented with the C-style bit-mapped structure shown below.

```c
typedef struct struct_bit8_type
{
    std::uint8_t b0 : 1;
    std::uint8_t b1 : 1;
    std::uint8_t b2 : 1;
    std::uint8_t b3 : 1;
    std::uint8_t b4 : 1;
    std::uint8_t b5 : 1;
    std::uint8_t b6 : 1;
    std::uint8_t b7 : 1;
} bit8_type;
```

Using the `bit8_type` structure is straightforward. For example, the code below sets `portb.5` to high.

```c
reinterpret_cast<volatile bit8_type*>(mcal::reg::portb)->b5 = 1U;
```

It can also be convenient to combine a built-in integral type with a bit-mapped register structure in a C-style union. For instance,

```c
typedef union union_reg_map_c
{
    std::uint8_t value;
    bit8_type bits;
} reg_map_c;
```

In this example, we have combined the 8 bits in the `bit8_type` structure with an `std::uint8_t` in the `reg_map_c` union. This makes it possible to manipulate either the individual bits or the value of the entire register depending on the coding situation. In particular,
// Set portb to 0.
reinterpret_cast<volatile reg_map_c*>(mcal::reg::portb)->value = 0U;

// Set portb.5 to 1.
reinterpret_cast<volatile reg_map_c*>(mcal::reg::portb)->bits.b5 = 1U;

In C++, it is possible to take the concept of the reg_map_c union and create from it a generic template class for register mapping. For example,

template<typename addr_type, typename reg_type, typename bits_type, const addr_type addr>
class reg_map
{
public:
  static reg_type& value()
  {
    return *reinterpret_cast<volatile reg_type*>(addr);
  }

  static bits_type& bits()
  {
    return *reinterpret_cast<volatile bits_type*>(addr);
  }
};

The reg_map class has four template parameters similar to the ones in the reg_access structure from the previous sections of this chapter. In particular, the addr_type parameter specifies the type of the register’s address. The addr parameter provides the constant value of the register’s address. The reg_type gives the type of the register. The new bits_type template parameter is intended to be a bit-mapped structure representing the bit-mapping of the hardware register.

These template parameters are used by reg_map’s two static members functions to provide access the register as a value or a bit-map. The value() subroutine returns a non-constant (i.e., modifiable) reference to the value of the register. The bits() subroutine returns a non-constant reference to the bit-mapped value of the register.
Imagine we would like to use the `reg_map` class to access the `portb` register on our target with the 8-bit microcontroller. In particular,

```cpp
// Set portb to 0.
reg_map<std::uint8_t,
        std::uint8_t,
        bit8_type,
        mcal::reg::portb>::value() = 0U;

// Set portb.5 to 1.
reg_map<std::uint8_t,
        std::uint8_t,
        bit8_type,
        mcal::reg::portb>::bits().b5 = 1U;
```

Bit-mapped structures provide an intuitive and elegant way to identically map a software structure to a hardware register or set of registers. Using bit-mapped structures, however, can result in potentially non-portable code. This is because, according to specification, the type of bit-field members in a structure must be one of `signed` or `unsigned int`. Bit-mapped structures, however, often use other integral types in order to obtain the right structure packing for the hardware.

If bit-mapped structures are to be used, one may want to check how the compiler handles them and ensure that the desired bit-mapping is actually carried out. The code of bit-mapped structures should also be clearly marked with a comment indicating potential non-portability.

**Reference**

Real-Time C++
Efficient Object-Oriented and Template Microcontroller Programming
Kormanyos, C.M.
2013, XXIII, 357 p. 20 illus., 3 illus. in color., Hardcover
ISBN: 978-3-642-34687-3