Mastering Enum Classes in Embedded C++

Prepared by: MD. Jesan

December 5, 2025

Introduction

Think of **enum classes** as giving "nicknames" to numbers. Instead of remembering 0, 1, 2 for LED colors or pin numbers, you give them meaningful names. In embedded systems, this makes your code safe and easy to read.

Why use enum classes in embedded systems?

- Type-safe: Prevents mixing unrelated numbers with pins or registers.
- Scoped: Values belong to the enum only, avoiding name conflicts.
- Can define underlying types (like `uint8_t`), perfect for hardware registers.

Basic Syntax

```
enum class LED : uint8_t {
    RED = 0,
    GREEN = 1,
    BLUE = 2
};
```

Accessing Enum Values

```
LED myLed = LED::RED;

// To use as a number (for registers), use static_cast
uint8_t ledVal = static_cast<uint8_t>(myLed);
```

Embedded Example: Direct Register Access

In embedded C++, you often need to manipulate memory addresses of hardware registers. Here, we use `reinterpret_cast` to safely convert a number to a pointer.

```cpp
// Define ports as constant pointers using reinterpret_cast
constexpr volatile uint8_t* PORTB_PTR = reinterpret_cast<volatile
    uint8_t*>(0x25);
constexpr volatile uint8_t* DDRB_PTR  = reinterpret_cast<volatile
    uint8_t*>(0x24);

// Set PB5 as output
*DDRB_PTR |= (1 << 5);

// Toggle PB5
*PORTB_PTR ^= (1 << 5);
```

Explanation for kids:

- `reinterpret_cast<volatile uint8_t*>(0x25)` says: "Hey, treat this number (0x25) as the address of a register!"
- `*PORTB_PTR` accesses the actual memory at that address.
- `static_cast<uint8_t>(PIN::PB5)` converts the enum PIN number into a number we can use in bit operations.

Using Enum Classes for Pins

```cpp
enum class PIN : uint8_t {
    PB5 = 5,
    PB4 = 4
};

// Toggle LED on PB5
*PORTB_PTR ^= (1 << static_cast<uint8_t>(PIN::PB5));
```

Why static_cast is needed

Enum class values are type-safe. You can't directly use them in arithmetic or bitwise operations. So we convert them to integers using `static_cast`.

Summary

- Enum classes give readable names to numbers.
- Use `reinterpret_cast` to turn a number into a pointer to a hardware register.
- Use `static_cast` to convert enum values to integers for bit operations.
- Makes embedded code safer, clearer, and less prone to errors.

Deep Dive: Understanding the getReg() Function

This section explains the following function in complete detail:

```
volatile uint8_t & getReg(Register reg)
{
    return *reinterpret_cast<volatile uint8_t*>(static_cast<
       uint8_t>(reg));
}
```

Let's break it down step by step, from inside out.

Step 1: static_cast¡uint8_t¿(reg)

```
enum class Register : uint8_t {
    TCCR0B = 0x25  // This is stored as the number 37 (0x25 in
       hex)
};
```

Why static_cast?

- `enum class` is **strongly typed** - it won't convert to a number automatically
- We need the actual number `0x25` to use as a memory address
- `static_cast<uint8_t>(reg)` converts `Register::TCCR0B` to `0x25`

```
Register reg = Register::TCCR0B;

uint8_t address = reg;                          // ERROR! enum
   class won't convert
uint8_t address = static_cast<uint8_t>(reg);  // OK! address = 0
   x25
```

Step 2: reinterpret_cast¡volatile uint8_t*¿(...)

Now we have the number `0x25`. But it's just a number, not a pointer.

Why reinterpret_cast?

- We want to treat `0x25` as a **memory address** (pointer)
- `reinterpret_cast` says: "Trust me, this number IS a memory address"

```
uint8_t address = 0x25;  // Just a number

volatile uint8_t* ptr = reinterpret_cast<volatile uint8_t*>(
   address);
// Now ptr points to memory location 0x25
```

Visual representation:

```
Memory:
Address:  0x24   0x25   0x26   0x27
          [ ]    [ ]    [ ]    [ ]
                  ^
                 ptr points here
```

Step 3: The * at the front (Dereference)

```
return *reinterpret_cast<volatile uint8_t*>(...);
       ^
       This asterisk!
```

Why *?

- `reinterpret_cast<volatile uint8_t*>(...)` gives us a **pointer** (an address)
- We want to return the **actual value** at that address, not the address itself
- `*` **dereferences** the pointer - it "goes to" the address and gets the value

```
volatile uint8_t* ptr = reinterpret_cast<volatile uint8_t*>(0x25);
// ptr = 0x25 (an address)

volatile uint8_t value = *ptr;
// value = whatever is stored at address 0x25
```

Step 4: Return type volatile uint8_t &

```
volatile uint8_t & getReg(...)
                  ^
                  Reference!
```

Why return a reference &?

- We want the caller to be able to **read AND write** to the register
- A reference lets you modify the original value

```
// Without reference (just value) - can only READ
uint8_t value = getReg(Register::TCCR0B);  // Gets a copy
value = 5;  // Only changes the copy, not the register!

// With reference - can READ and WRITE
getReg(Register::TCCR0B) = 5;              // Directly changes the register!
getReg(Register::TCCR0B) |= (1 << 2);      // Modifies the register!
```

Complete Flow Example

```
getReg(Register::TCCR0B) |= (1 << 2);
```

1. `Register::TCCR0B` - enum value
2. `static_cast<uint8_t>(...)` - converts to `0x25` (number)

3. `reinterpret_cast<volatile uint8_t*>(...)` - pointer to address `0x25`
4. `*` - dereference to get the value at that address
5. `&` - return as reference so caller can modify it
6. `|= (1 << 2)` - modify the actual hardware register!

Visual Summary

```
Register::TCCR0B
       |
       v
static_cast<uint8_t>  -->  0x25 (just a number)
       |
       v
reinterpret_cast<volatile uint8_t*>  -->  0x25 (now it's a pointer)
       |
       v
* (dereference)  -->  actual value at memory 0x25
       |
       v
& (reference)  -->  caller can read/write directly
```

Pointer vs Reference: Simple Analogy

Pointer (*): Your friend gives you the address of their house. You have to go there first to access it.

Reference (&): Your friend hands you the keys to their house directly. You can use it right away.

```
// Pointer version - need to dereference
volatile uint8_t * getReg(Register reg) {
    return reinterpret_cast<volatile uint8_t*>(static_cast<
       uint8_t>(reg));
}
*getReg(Register::TCCR0B) |= (1 << 2);  // Need * to access

// Reference version - direct access
volatile uint8_t & getReg(Register reg) {
    return *reinterpret_cast<volatile uint8_t*>(static_cast<
       uint8_t>(reg));
}
getReg(Register::TCCR0B) |= (1 << 2);   // No * needed!
```

Best Practice: Use **reference** because it's simpler and safer!

Creating a Delay Function Using Timer0

In this section, we will create a precise 1ms delay function using Timer0 on the ATmega328P.

The Math Behind the Delay

To create a delay, we need to understand how the timer counts:

1. **CPU Frequency:** The ATmega328P runs at 16 MHz (16,000,000 Hz)
2. **Prescaler:** We divide the clock to slow down counting
3. **Timer Register:** TCNT0 is an 8-bit counter (0-255)

Formula:

$$\text{Timer Frequency} = \frac{F\_CPU}{\text{Prescaler}}$$

$$\text{Time per tick} = \frac{1}{\text{Timer Frequency}}$$

$$\text{Ticks needed} = \frac{\text{Desired Time}}{\text{Time per tick}}$$

Calculation for 1ms Delay

Let's use a prescaler of 64:

$$\text{Timer Frequency} = \frac{16,000,000}{64} = 250,000 \text{ Hz}$$

$$\text{Time per tick} = \frac{1}{250,000} = 4\mu s$$

$$\text{Ticks for 1ms} = \frac{1000\mu s}{4\mu s} = 250 \text{ ticks}$$

So we need to count 250 ticks to get exactly 1ms!

Why We Reset TCNT0

```
getReg(Register::TCNT0) = 0;  // Reset counter to 0
```

Why reset TCNT0?

- TCNT0 is the Timer Counter register - it counts up automatically
- If we don't reset it, it might start from any value (0-255)
- Resetting ensures we always count from 0 to 250 for accurate timing
- Without reset: unpredictable delay times!

Visual:

```
Without reset:          With reset:
TCNT0 = 100             TCNT0 = 0
  |                       |
  v                       v
Count to 250            Count to 250
  |                       |
  v                       v
Only 150 ticks!         Exactly 250 ticks!
(0.6ms - WRONG!)        (1ms - CORRECT!)
```

Why Compare with ¡ 250, not ¡= 250?

```
while(getReg(Register::TCNT0) < 250)    // Correct: 0 to 249 = 250 ticks
while(getReg(Register::TCNT0) <= 250)   // Wrong: 0 to 250 = 251 ticks
```

Explanation:

- Counter starts at 0
- `< 250`: Loop runs while counter is 0, 1, 2, ... 249 (250 values = 250 ticks)
- `<= 250`: Loop runs while counter is 0, 1, 2, ... 250 (251 values = 251 ticks)

Think of it like counting on your fingers:

- Fingers 0-249 = 250 fingers (correct!)
- Fingers 0-250 = 251 fingers (one too many!)

Complete delayms() Function

```
void delayms(uint16_t ms)
{
    // Prescaler 64: CS01=1, CS00=1
    getReg(Register::TCCR0B) = (1 << Bits::CS00) | (1 << Bits::CS01);

    while(ms--)
    {
        // Reset timer counter to 0
        getReg(Register::TCNT0) = 0;

        // 16MHz / 64 = 250,000 Hz
        // 1 tick = 4us
        // 250 ticks = 1ms
        while(getReg(Register::TCNT0) < 250)
        {
            // Wait
        }
    }
}
```

Prescaler Selection Table

CS02	CS01	CS00	Prescaler	Timer Freq (16MHz)
0	0	0	No clock	Timer stopped
0	0	1	1	16 MHz
0	1	0	8	2 MHz
0	1	1	64	250 kHz
1	0	0	256	62.5 kHz
1	0	1	1024	15.625 kHz

Step-by-Step Flow

1. Set prescaler to 64 (TCCR0B = CS01 — CS00)
2. For each millisecond:
   (a) Reset TCNT0 to 0
   (b) Wait until TCNT0 reaches 250
   (c) Repeat for next millisecond

Visual Flow:

```
delayms(3) called:

ms=3: TCNT0: 0 -> 1 -> 2 -> ... -> 249 (1ms) done!
ms=2: TCNT0: 0 -> 1 -> 2 -> ... -> 249 (1ms) done!
ms=1: TCNT0: 0 -> 1 -> 2 -> ... -> 249 (1ms) done!

Total: 3ms delay complete!
```

What is a Namespace?

Imagine you have a huge toy box full of toys. Sometimes two toys might have the same name, like two "robots" from different sets. To avoid confusion, you put toys from Set A in one small box and toys from Set B in another. Now you know exactly which "robot" you are playing with.

In C++ programming, a **namespace** is like that small box. It helps us avoid confusion when we have functions, variables, or classes with the same name.

- Namespaces are a way to group code into a named "box."
- They prevent name conflicts between different parts of your program.
- Especially in embedded programming, where you might deal with registers, peripherals, and different libraries, namespaces help keep your code organized.

Basic Example

```
// Two sets of LEDs in different namespaces
namespace BoardA {
    void turnOnLED() {
        // Code to turn on LED in Board A
    }
}

namespace BoardB {
    void turnOnLED() {
        // Code to turn on LED in Board B
    }
}

int main() {
    BoardA::turnOnLED(); // Calls the function from Board A
    BoardB::turnOnLED(); // Calls the function from Board B
}
```

Explanation: By putting functions in different namespaces, we avoid the compiler getting confused about which `turnOnLED()` function we mean.

Namespaces in Embedded Programming

In embedded systems, you often control hardware directly:

- Different modules (like ADC, PWM, GPIO) may have functions with the same names.
- Using namespaces lets you group functions for each module separately.
- Example: `GPIO::init()`, `PWM::init()`, `ADC::init()`.

Advanced Usage: Nested Namespaces

You can even have namespaces inside namespaces. It's like putting a smaller box inside a bigger one.

```
namespace BoardA {
    namespace LED {
        void turnOn() { /* turn on LED */ }
    }
}

int main() {
    BoardA::LED::turnOn(); // Nested access
}
```

Why Namespaces are Useful in Embedded Programming

1. **Avoid Conflicts:** Multiple libraries may define the same function name.
2. **Organize Code:** Group functions by hardware modules or features.
3. **Maintainability:** Easy to find and modify code.
4. **Scalability:** Helpful when projects grow larger.

Kid-Friendly Summary

Think of namespaces like toy boxes. Each box has its own toys. You never mix up your Lego robot with your Transformers robot because they are in separate boxes. Similarly, namespaces keep your code clean, organized, and safe from name conflicts.