A Deeper Understanding of Data Types and Type Casting

In Embedded Systems, playing with bits is key. Understanding how bits are stored in variables and the effects of type casting is crucial. It's all about getting the details right in this bit-level manipulation game.

Basic Integer Type with its Modifiers:

uint8_t and int8_t (Unsigned and Signed 8-bit integer):

- Size: 1 Byte
- Range(uint8_t): 0 to 255
- Range(int8_t): -128 to 127

uint16_t and int16_t (Unsigned and Signed 16-bit Integer):

- Size: 2 bytes
- Range (uint16_t): 0 to 65,535
- Range (int16_t): -32,768 to 32,767

uint32_t and int32_t (Unsigned and Signed 32-bit Integer):

- Size: 4 bytes
- Range (uint32_t): 0 to 4,294,967,295
- Range (int32_t): -2,147,483,648 to 2,147,483,647

uint64_t and int64_t (Unsigned and Signed 64-bit Integer):

- Size: 8 bytes
- Range (uint64_t): 0 to 18,446,744,073,709,551,615
- Range (int64_t): -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Short and Long Modifiers

short int or short

- Typically a signed 16-bit integer
- Range: -32,768 to 32,767

unsigned short int

- Typically an unsigned 16-bit integer.
- Range: 0 to 65,535

long int or long

- Commonly a signed 32-bit integer, but can be 64-bit on some systems.
- Range: -2,147,483,648 to 2,147,483,647 (for 32-bit)
- Range: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (for 64-bit)

unsigned long int

- Commonly an unsigned 32-bit integer, but can be 64-bit on some systems.
- Range: 0 to 4,294,967,295 (for 32-bit)
- Range: 0 to 18,446,744,073,709,551,615 (for 64-bit)

long long int

- Typically a signed 64-bit integer.
- Range: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

unsigned long long int

- Typically an unsigned 64-bit integer.
- Range: 0 to 18,446,744,073,709,551,615

Floating Point Types

float

- Size: 4 bytes
- Precision: Typically 6-7 decimal digits

double

- Size: 8 bytes
- Precision: Typically 15-16 decimal digits

long double

- Size: Varies, often 12, 16, or 16 bytes depending on the system.
- Precision: More than **double**, but specific precision varies by system.

Char type

char

- Size: 1 byte
- Can be signed or unsigned based on the system/compiler. Used to represent characters.

wchar_t

- Size: Varies, often 2 or 4 bytes depending on the system.
- Used for wide characters, capable of representing a larger set of characters (like Unicode).

bool

- Size: Typically 1 byte
- Represents: **true** (1) or **false** (0)

Playing with the bits:

Scenario 1: Storing uint8 Return Value in uint64

Imagine a function that returns a **uint8_t** value, but you need to store this in a **uint64_t** variable and manipulate its individual bytes.

```
// Function returning uint8_t value
uint8_t getValue() {
    return 0xAB; // Example value in hexadecimal
}

// Storing and manipulating in uint64_t
uint64_t extendedValue = getValue();
// Manipulate and access individual bytes
uint8_t byte1 = (extendedValue >> 8) & 0xFF; // 2nd byte (will be 0x00)
```

Considering a Little-Endian system, **extendedValue** will have its least significant byte set to the **uint8_t** value, and the rest of the bytes will be zero.

- The return value is **0xAB**(**10101011**). Since **getValue()** returns a **uint8_t**, it represents an 8-bit value.
- **extendedValue** is a **uint64_t** variable, meaning it has 64 bits. Initially, all bits are zero.
- After assignment, **extendedValue** holds the **0xAB** value in its least significant byte (8 bits), while the rest of the bits remain zero: **00000000 00000000 00000000 00000000 00000000 00000000 00000000 10101011**.
- The right shift operator **>> 8** shifts **extendedValue** 8 bits to the right, resulting in all zeroes in the least significant byte (the original **0xAB** is shifted out): **00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000**.
- The bitwise AND with **0xFF** (which is **11111111** in binary) isolates the least significant byte. Since this byte is all zeroes, **byte1** becomes **00000000** in binary, or **0x00** in hexadecimal.

Scenario 2: Extracting Bytes from uint64_t

Consider you have a **uint64_t** value, and you need to store each byte in an array of **uint8_t**.

```
uint64_t bigValue = 0x1122334455667788;
uint8_t byteArray[8];
for (int i = 0; i < 8; i++) {
    byteArray[i] = (bigValue >> (i * 8)) & 0xFF;
}
```

Here, you're extracting each byte from **bigValue** and storing it in **byteArray**

bigValue is a 64-bit value: **00010001 00100010 00110011 01000100 01010101 01100110 01110111 10001000**. The loop iterates 8 times, extracting each byte from **bigValue**.

i = 0: **(bigValue >> (0 * 8)) & 0xFF**

- **bigValue** shifted right by 0 bits: **00010001 00100010 00110011 01000100 01010101 01100110 01110111 10001000**
- After applying **& 0xFF**: **10001000** (Last 8 bits)
- Stored in **byteArray[0]**.

Similarly,

- **byteArray[0]**: **0x88** (**10001000**)
- **byteArray[1]**: **0x77** (**01110111**)
- **byteArray[2]**: **0x66** (**01100110**)
- **byteArray[3]**: **0x55** (**01010101**)
- **byteArray[4]**: **0x44** (**01000100**)
- **byteArray[5]**: **0x33** (**00110011**)
- **byteArray[6]**: **0x22** (**00100010**)
- **byteArray[7]**: **0x11** (**00010001**)

Scenario 3: Type Casting for Memory Efficiency

Suppose you have an array of **uint64_t**, but you know that the values will always be within the **uint8_t** range. To save memory, you cast them to **uint8_t** when processing.

```
uint64_t largeNumbers[5] = {1, 2, 3, 4, 5};
uint8_t smallNumbers[5];
for (int i = 0; i < 5; i++) {
    smallNumbers[i] = static_cast<uint8_t>(largeNumbers[i]);
}
```

- **largeNumbers** is an array of **uint64_t** values. Each **uint64_t** is a 64-bit (8-byte) value.
- The array contains simple numbers (1, 2, 3, 4, 5) which are quite small but stored in a large 64-bit format.

This loop iterates through **largeNumbers**, casting each 64-bit value to an 8-bit (**uint8_t**) value.

For Each Element:

- Let's take **largeNumbers[0] = 1** as an example. In 64-bit , **00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001**.
- When this is cast to **uint8_t**, only the least significant 8 bits are kept, resulting in **00000001**.

After casting:

- **smallNumbers[0]**: **1** (**00000001**)
- **smallNumbers[1]**: **2** (**00000010**)
- **smallNumbers[2]**: **3** (**00000011**)
- **smallNumbers[3]**: **4** (**00000100**)
- **smallNumbers[4]**: **5** (**00000101**)

Scenario 4: Bit Manipulations for Control Operations

Setting or clearing specific bits,

```
uint8_t controlRegister = 0b00001111; // Initial state
// Set the 6th bit
controlRegister |= (1 << 5);
// Clear the 3rd bit
controlRegister &= ~(1 << 2);
```

controlRegister is an 8-bit value initialized to **00001111**.

- **(1 << 5)** shifts the binary **1** five places to the left, resulting in **00100000**.
- The bitwise OR (**|=**) operation between **controlRegister** (**00001111**) and **00100000** sets the 6th bit (zero-indexed) to **1** while leaving other bits unchanged. **controlRegister** becomes **00101111**.
- **(1 << 2)** shifts the binary **1** two places to the left, giving **00000100**.
- **~** negates this value, resulting in **11111011**.
- The bitwise AND (**&=**) operation between **controlRegister** (**00101111**) and **11111011** clears the 3rd bit.
- **controlRegister** becomes **00101011**.
- **After Setting the 6th Bit**: **controlRegister** = **00101111**
- **After Clearing the 3rd Bit**: **controlRegister** = **00101011**

Scenario 5: Combining Multiple uint8_t Values into a uint32_t

Imagine a situation where you need to combine four **uint8_t** sensor readings into a single **uint32_t** value for efficient transmission.

```
uint8_t sensor1 = 0x12; // Example sensor values
uint8_t sensor2 = 0x34;
uint8_t sensor3 = 0x56;
uint8_t sensor4 = 0x78;

uint32_t combinedSensors = (static_cast<uint32_t>(sensor1) << 24) |
                           (static_cast<uint32_t>(sensor2) << 16) |
                           (static_cast<uint32_t>(sensor3) << 8)  |
                            static_cast<uint32_t>(sensor4);
```

This example shows how to shift and combine multiple **uint8_t** values into a single **uint32_t**.

In binary: **sensor1 = 00010010**, **sensor2 = 00110100**, **sensor3 = 01010110**, **sensor4 = 01111000**.

Shifting and Combining:

- **sensor1**: Shifted left by 24 bits: **00010010 00000000 00000000 00000000**.
- **sensor2**: Shifted left by 16 bits: **00000000 00110100 00000000 00000000**.
- **sensor3**: Shifted left by 8 bits: **00000000 00000000 01010110 00000000**.
- **sensor4**: No shift: **00000000 00000000 00000000 01111000**.

Bitwise OR Operations:

- Combining these shifted values using bitwise OR (**|**) results in the **combinedSensors** value.
- The final **combinedSensors** value: **00010010 00110100 01010110 01111000**.

combinedSensors: **0x12345678** in hexadecimal, combining **sensor1**, **sensor2**, **sensor3**, and **sensor4** in that order.

Scenario 6: Signed to Unsigned Casting and Vice-Versa

Consider a scenario where you need to cast between signed and unsigned types, such as **int32_t** to **uint32_t**.

```
int32_t signedValue = -12345;
uint32_t unsignedValue = static_cast<uint32_t>(signedValue);
int32_t signedValueAgain = static_cast<int32_t>(unsignedValue);
```

signedValue is first cast to an **uint32_t**, and then back to an **int32_t**, demonstrating how values can change with casting between signed and unsigned types.

- **signedValue** is a signed 32-bit integer (**int32_t**) with the value **-12345**.
- In binary (two's complement representation for negative numbers): **signedValue** = **11111111 11111111 11000011 10001001**.

- The signed value **-12345** is cast to an unsigned 32-bit integer (**uint32_t**).
- The binary representation remains the same (**11111111 11111111 11000011 10001001**), but its interpretation changes due to the unsigned nature.
- As an unsigned value, this binary now represents a large positive number.
- Casting the **unsignedValue** back to **int32_t** doesn't change its binary representation.
- The binary **11111111 11111111 11000011 10001001** is still the same, and in the context of **int32_t**, it's interpreted as **-12345**.
- **signedValue**: -12345 (**11111111 11111111 11000011 10001001**)
- **unsignedValue**: A large positive number (remains as is, **11111111 11111111 11000011 10001001**, but interpreted as unsigned)
- **signedValueAgain**: -12345 (**11111111 11111111 11000011 10001001**)

Scenario 7: Bitwise Operations for Flag Handling

Using bitwise operations to handle multiple flags within a single byte is a common practice:

```
uint8_t flags = 0b00000000;
// Set flag at position 2
flags |= (1 << 2);
// Check if flag at position 4 is set
bool isSet = flags & (1 << 4);
// Clear flag at position 2
flags &= ~(1 << 2);
```

setting, checking, and clearing flags using bitwise operations.

flags is an 8-bit value initialized to **00000000** in binary, representing a byte with all flags cleared.

- **(1 << 2)** shifts the binary **1** two places to the left, resulting in **00000100**.
- The bitwise OR (**|=**) operation between **flags** (**00000000**) and **00000100** sets the 3rd bit (zero-indexed) to **1**. **Result**: **flags** becomes **00000100**.
- **(1 << 4)** shifts the binary **1** four places to the left, giving **00010000**.
- The bitwise AND (**&**) operation between **flags** (**00000100**) and **00010000** checks the 5th bit. **Result**: **isSet** is **false** since the 5th bit in **flags** is not set.
- **(1 << 2)** shifts the binary **1** two places to the left, resulting in **00000100**.
- **~** negates this value, resulting in **11111011**.
- The bitwise AND (**&=**) operation between **flags** (**00000100**) and **11111011** clears the 3rd bit. **Result**: **flags** becomes **00000000** again.
- **After Setting Flag at Position 2**: **flags** = **00000100** (binary)
- **After Checking Flag at Position 4**: **isSet** = **false**
- **After Clearing Flag at Position 2**: **flags** = **00000000** (binary)

To summarize the understanding:

Type Casting uint64_t to uint8_t: Gives you the **least significant byte** of the uint64_t value.

```
uint64_t largeValue = 0x1122334455667788;
uint8_t smallValue = static_cast<uint8_t>(largeValue); // smallValue will be 0x88
```

Type Casting uint32_t to uint16_t: Give you the lower 16 bits of the **uint32_t** value.

```
uint32_t mediumValue = 0x12345678;
uint16_t shortValue = static_cast<uint16_t>(mediumValue); // shortValue will be 0x5678
```

Type Casting float to int32_t: Gives you a floating-point number to an integer, truncating any decimal places.

```
float floatValue = 123.456f;
int32_t intValue = static_cast<int32_t>(floatValue); // intValue will be 123
```

Type Casting uint8_t to int8_t: The bit pattern remains the same, but the interpretation of the value changes.

```
uint8_t uValue = 0xFF; // 255 in unsigned
int8_t sValue = static_cast<int8_t>(uValue); // sValue will be -1
```

Type Casting int64_t to uint64_t: The bit pattern from a signed to an unsigned 64-bit integer.

```
int64_t signedValue = -1;
uint64_t unsignedValue = static_cast<uint64_t>(signedValue); // unsignedValue will be
0xFFFFFFFFFFFFFFFF
```

Type Casting double to float: Might result in a loss of precision as double has a higher precision than float.

```
double highPrecisionValue = 123.456789012345;
float lessPrecisionValue = static_cast<float>(highPrecisionValue); // lessPrecisionValue will
be approximately 123.45679
```

In general, there's a colloquial saying that goes, 'fuck around and find out.' This might seem a bit unorthodox, but in the world of Embedded Systems, especially when it comes to understanding bit-level operations, this saying holds a significant truth. During the learning phase, there's no substitute for actually 'fucking around' with the code and hardware. Experimenting with different types of casting, manipulating bits and bytes, and observing the outcomes firsthand is more than just educational—it's a necessity. It's through this process of trial, error, and discovery that one truly begins to grasp what's happening under the hood. So, while theoretical knowledge provides a solid foundation, it's the practical tinkering, the direct engagement with bits and bytes, that turns theory into real-world expertise.

Happy Learning!

~~~~~~

An Article Written by: Yashwanth Naidu Tikkisetty

~~~~~~