FEABHAS

EMBEDDING SOFTWARE COMPETENCE

Modern C++ White paper:

Making things do stuff

Glennan Carnie
Technical consultant, Feabhas

Introduction

C has long been the language of choice for smaller, microcontroller-based embedded systems; particularly for close-to-the-metal hardware manipulation.

C++ was originally conceived with a bias towards systems programming; performance and efficiency being key design highlights. Traditionally, many of the advancements in compiler technology, optimisation, etc., had centred around generating code for PC-like platforms (Linux, Windows, etc). In the last few years C++ compiler support for microcontroller targets has advanced dramatically, to the point where Modern C++ is an increasingly attractive language for embedded systems development.

In this whitepaper we will explore how to use Modern C++ to manipulate hardware on a typical embedded microcontroller. We'll see how you can use C++'s features to hide the actual underlying hardware of our target system and provide an abstract hardware API that developers can work to. We'll explore the performance (in terms of memory and code size) of these abstractions compared to their C counterparts.

Contents

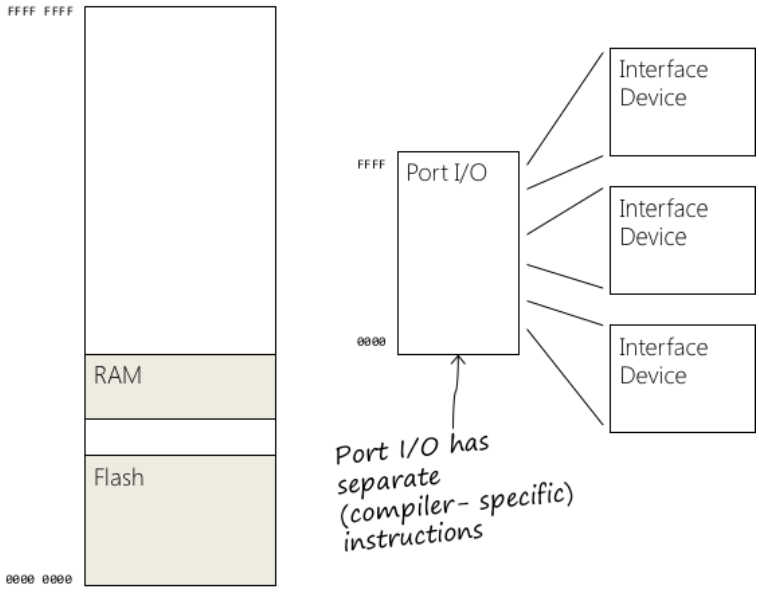
Basic concepts

We'll begin by having a look at the very basics of hardware manipulation – accessing hardware devices and bit manipulation.

Port vs memory-mapped I/O

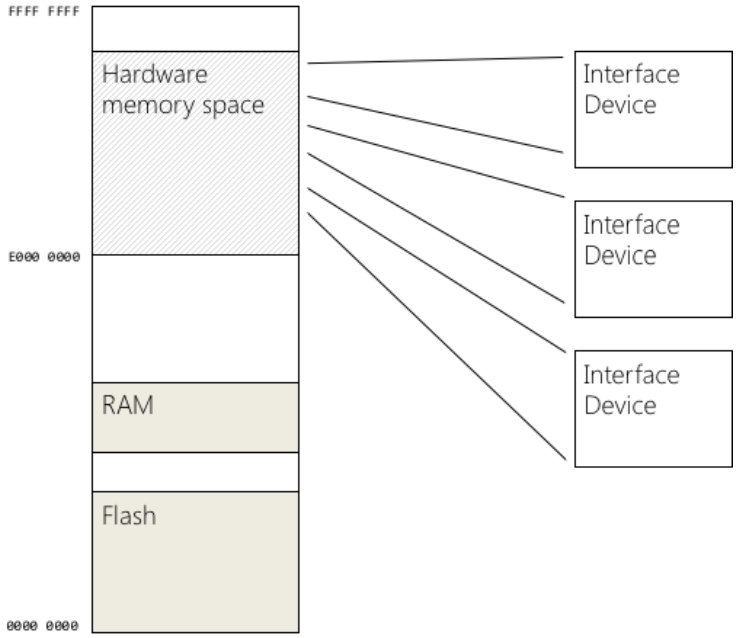
Memory-Mapped Input/Output (MMIO) and port Input/Output (also called port-mapped I/O or PMIO) are two complementary methods of performing input/output between the CPU and I/O devices in a target.

PMIO uses a special class of CPU instructions specifically for performing I/O. This is generally found on Intel microprocessors, specifically the IN and OUT instructions which can read and write a single byte to an I/O device. I/O devices have a separate address space from general memory, either accomplished by an extra "I/O" pin on the CPU's physical interface, or an entire bus dedicated to I/O.



PMIO is usually accessed via special compiler-specific intrinsic functions; that is, non-standard C++. For that reason, I'll exclude PMIO from this discussion.

MMIO uses the same bus to address both memory and I/O devices, and the CPU instructions used to read and write to memory are also used in accessing I/O devices. In order to accommodate the I/O devices, areas of CPU addressable space must be reserved for I/O rather than memory. The I/O devices monitor the CPU's address bus and respond to any CPU access of their assigned address space, mapping the address to their hardware registers.



It is important to note that the two techniques are not normally found on same architecture; although for example on older PCs video ram would be memory mapped and all other I/O devices would be port I/O.

Accessing hardware from C++

The problem from a language perspective is that the compiler can only see objects that have been declared; and by definition those objects with be in the memory areas. There is no direct way of accessing I/O addresses from C++. The way round this is to *indirectly* access the memory, using pointers.

There are two things to take into consideration with this method:

- We must select a pointer type that matches our hardware register
- We have to 'force' an address into the pointer

The type of a pointer tells the compiler how many bytes to access, how to interpret the bits and the valid behaviours that can be performed on that memory location. In configuration-type registers normally each bit has its own significance; for data-type registers the value is normally held as a 'raw' number. For that reason, hardware access registers should always be qualified as unsigned. The actual type depends on the size of the hardware register.

(Avoid assuming that (for example) a 32-bit register pointer is good enough for all 8-, 16- and 32-bit registers. You risk potentially overwriting adjacent registers; and even adjacent reads could have unintended side effects).

```
unsigned char*  reg_8  { };  // 8-bit
unsigned short* reg_16 { };  // 16-bit
unsigned long*  reg_32 { };  // 32-bit
```

Note: A 32-bit register (for example) may be accessible (depending on hardware) as either 8-bits, 16-bits or 32-bits.

However, using a pointer-to-32-bit to access an 8-bit register is undefined. You risk overwriting adjacent registers; and even reads could have unintended side-effects (for example, reading some hardware registers has the effect of clearing them)

To be more explicit it's probably good practice to use specific-width aliases:

```
#include <cstdint>

int main()
{
  std::uint8_t*  reg_8  { };
  std::uint16_t* reg_16 { };
  std::uint32_t* reg_32 { };
  ...
}
```

Our next problem is to 'force' a register address into the pointer. As an integral value cannot be assigned to a pointer, we must cast the value. The correct C++ way is using `reinterpret_cast<>` (in fact, it's one of the few uses of `reinterpret_cast<>`)

```
#include <cstdint>

using std::uint8_t;
using std::uint16_t;
using std::uint32_t;

int main()
{
  uint8_t*  reg_8  { reinterpret_cast<uint8_t*>(0x40020000)  };
  uint16_t* reg_16 { reinterpret_cast<uint16_t*>(0x40020010) };
  uint32_t* reg_32 { reinterpret_cast<uint32_t*>(0x40020020) };
  ...
}
```

Now we've set up the pointer we can access the register indirectly through the pointer.

```
#include <cstdint>

using std::uint8_t;
using std::uint16_t;
using std::uint32_t;

int main()
{
  uint8_t*  reg_8  { reinterpret_cast<uint8_t*>(0x40020000)  };
  uint16_t* reg_16 { reinterpret_cast<uint16_t*>(0x40020010) };
  uint32_t* reg_32 { reinterpret_cast<uint32_t*>(0x40020020) };

  uint8_t value { };

  value = *byte_reg;      // Read
  *byte_reg = 0;          // Write
}
```

Since our hardware registers are at fixed locations in memory (and should never change!) we can improve the optimisation capacity of the compiler by making the pointers constants.

```
#include <cstdint>

using std::uint8_t;
using std::uint16_t;
using std::uint32_t;

int main()
{
  uint8_t*  const reg_8  { reinterpret_cast<uint8_t*>(0x40020000)  };
  uint16_t* const reg_16 { reinterpret_cast<uint16_t*>(0x40020010) };
  uint32_t* const reg_32 { reinterpret_cast<uint32_t*>(0x40020020) };

  uint8_t value { };

  value = *byte_reg;      // Read
  *byte_reg = 0;          // Write; also OK.  Register is not constant
}
```

Remember, the `const` refers to the pointer, not the object being addressed (the register)

(It might seem compelling at this point to consider making the pointers a `constexpr`. After all, the pointer is fixed and can never change. However, the C++ standard explicitly prohibits the results of `reinterpret_cast` in constant-expressions. See following link for more details - http://stackoverflow.com/questions/10369606/constexpr-pointer-value)

Registers and side-effects

One of the defining features of hardware registers is their value is dictated by the (current) state of the hardware; and not necessarily by the actions of the program. This means, for example:

- A write to a hardware register, followed by a read may not yield the same value. This is often true for write-only registers
- Two sequential reads from a register may yield different results

We could therefore declare any access to a hardware register – read or write – could yield a side-effect.

However, the compiler can only reason about objects declared within the program. It will base all its optimisations on the code that is presented. For example:

```
int main()
{
  uint8_t* const ctrl { reinterpret_cast<uint8_t*>(0x40020000)  };
  uint8_t* const cfg  { reinterpret_cast<uint8_t*>(0x40020001)  };
  uint8_t* const data { reinterpret_cast<uint8_t*>(0x40020002)  };


  while(*data == 0)
  {
    // Wait for data to arrive...
  }
}
```

The code above could fail at run-time; particularly at higher optimisation levels.

From the compiler's perspective there is no code to modify the object referenced by data. Therefore, the compiler is free to optimise out the (apparently) redundant reads and simply read `*data` before the loop.

Similarly, the compiler is likely to optimise redundant writes; for example:

```
int main()
{
  uint8_t* const ctrl { reinterpret_cast<uint8_t*>(0x40020000)  };
  uint8_t* const cfg  { reinterpret_cast<uint8_t*>(0x40020001)  };
  uint8_t* const data { reinterpret_cast<uint8_t*>(0x40020002)  };

  *ctrl = 1;   // Enter configuration mode
  *cfg  = 3;   // Configure the device
  *ctrl = 0;   // Enter operational mode.
}
```

It is likely that the first write to `*ctrl` will be optimised away, since there is no read of `*ctrl` before the second write.

We have to inform the compiler that the object we are referencing via the pointer is a special case and therefore any optimisations (removing redundant reads / writes; or re-orderings) must be disabled for this object. Enter the *volatile* qualifier:

```
int main()
{
  volatile uint8_t* const ctrl { reinterpret_cast<uint8_t*>(0x40020000)  };
  volatile uint8_t* const cfg  { reinterpret_cast<uint8_t*>(0x40020001)  };
  volatile uint8_t* const data { reinterpret_cast<uint8_t*>(0x40020002)  };

  *ctrl = 1;      // Redundant writes not optimised-out.
  *cfg  = 3;
  *ctrl = 0;

  ...

  while(*data == 0) {   // Redundant reads not optimised-out
    // Wait for data...
  }
}
```

Unless you have a very good reason not to, all hardware-access pointers should be marked as volatile.

We've now established the basic idiom for accessing hardware via pointers.

An aside: decluttering code

Our pointer declarations are starting to look a bit verbose. We can use auto type-deduction to make our code cleaner:

```
int main()
{
  auto const ctrl { reinterpret_cast<volatile uint8_t*>(0x40020000) };
  auto const cfg  { reinterpret_cast<volatile uint8_t*>(0x40020001) };
  auto const data { reinterpret_cast<volatile uint8_t*>(0x40020002) };

  *ctrl = 1;
  *cfg  = 3;
  *ctrl = 0;

  while(*data == 0) {
    // Wait for data...
  }
```

```
}
```

There are a few things to note in this new code:

Since `auto` uses the type of the initialiser to determine the type of the object, we must change the type of the `reinterpret_cast<>` to a `volatile uint8_t*`.

The `const` qualifier is applied *after* type-deduction and applies to the deduced type. So in this case our pointers become

```
volatile uint8_t* const ctrl;
volatile uint8_t* const cfg;
volatile uint8_t* const data;
```

Although our idiom is explicit (and for that reason, preferred) all the pointer dereferencing can make code less-than-clean to read.
As a C programmer we might resort to the pre-processor to clean up the code:

```
// C programmer's version
//
#define CTRL (*(volatile uint8_t*) 0x4002000)
#define CFG  (*(volatile uint8_t*) 0x4002010)
#define DATA (*(volatile uint8_t*) 0x4002020)

int main(void)
{
  CTRL = 1;
  CFG  = 3;
  CTRL = 0;

  while(DATA == 0) {
    //...
  }
}
```

The macros (`CTRL`, `CFG`, `DATA`) perform an inline cast of an integer to a pointer, then immediately dereference it to get an object. So any access to `CTRL` (for example) will be an indirect access to the address `0x4002000`.

In C++ we can use references to achieve the same effect. The advantage of using references is that they will appear in the symbol table, making debugging easier.

```
// C++ programmer's version
//
int main()
{
  auto& ctrl { *reinterpret_cast<volatile uint8_t*>(0x40020000) };
  auto& cfg  { *reinterpret_cast<volatile uint8_t*>(0x40020010) };
  auto& data { *reinterpret_cast<volatile uint8_t*>(0x40020020) };

  ctrl = 1;
  cfg  = 3;
  ctrl = 0;

  while(data == 0) {
    //...
  }
}
```

Notice the pointer dereference in the initialisers - we want references to objects, not pointers. Also notice you don't have to make references `const`. References cannot be 're-seated' so are effectively always `const`.

Although we'll look at this more later it's worth mentioning that both the pointer and reference versions shown will generate the same opcodes. The choice, then, becomes which is clearer to the reader (and maintainer) of the code. For the rest of this paper, I'm going to stick to the pointer version. From an explanation point of view it is more explicit that we are indirectly accessing the IO memory.

Bit manipulation

One of the distinguishing aspects of hardware manipulation code is that we are often dealing with variables on a bit-by-bit basis. There are a set of idiomatic operations we'll need to do regularly:

- Set a particular bit, or set of bits
- Clearing bit(s)
- Check to see if a bit is set or not

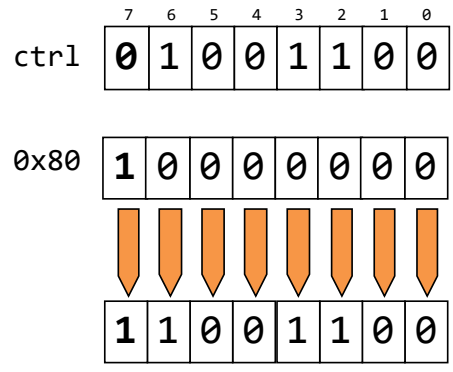
Setting bits

When setting individual bits a we have to leave all the bits we're not interested in unchanged. Therefore, a simple assignment is not adequate. We need bitwise OR:

```
int main()
{
  auto const ctrl { reinterpret_cast<volatile uint8_t*>(0x40020000) };
  auto const cfg  { reinterpret_cast<volatile uint8_t*>(0x40020001) };
  auto const data { reinterpret_cast<volatile uint8_t*>(0x40020002) };

  *ctrl = *ctrl | 0b10000000;  // Set bit 7
  ...
}
```

This code will set bit 7 of the ctrl register, leaving all others intact, since OR-ing with zero has no effect.



The above code explicitly shows the read-modify-write operation, although idiomatically programmers prefer the syntactic sugar of the OR-assignment operator.

```
*ctrl |= 0b10000000;
```

Notice here we're using C++'s binary literal to specify the bits we want to set. Hard-coding bit values is fine for simple (8-bit) values but can become tedious – and error-prone – for multiple bits on larger words (For example, what about setting bits 17 and 23 on a 32-bit word?).

We could use hexadecimal:

```
*ctrl |= 0x80
```

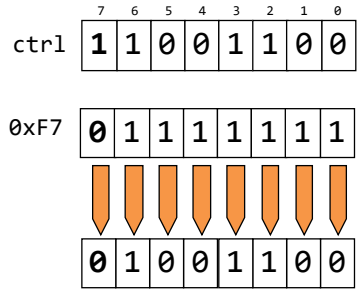
Or we can make use of the left-shift operator:

```
*ctrl |= (1 << 7);
```

That is, put a 1 in the least-significant bit position then shift left 7 times. This will put the 1 in bit 7, the rightmost bits guaranteed to be zero.

Clearing a bit

We (obviously?) can't use bitwise-OR to clear a bit since OR-ing with zero has no effect. When clearing a bit we need to set the offending bits to 0, whilst maintaining the state of all other bits. For this we use bitwise-AND




```
int main()
{
  auto const ctrl { reinterpret_cast<volatile uint8_t*>(0x40020000) };
  auto const cfg  { reinterpret_cast<volatile uint8_t*>(0x40020001) };
  auto const data { reinterpret_cast<volatile uint8_t*>(0x40020002) };

  *ctrl |= (1 << 7);  // Set bit 7
  ...

  *ctrl &= 0b01111111;      // Clear bit 7
  ...
}
```

To make the code more readable (for larger register sizes) we can again make use of the bitwise-NOT operator (~):

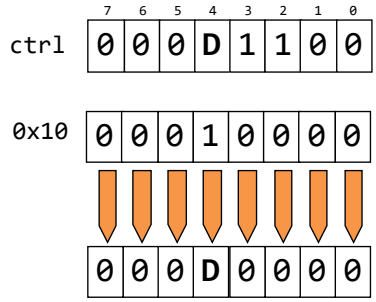
```
·ctrl &= ~0x80;
```

Or even:

```
*ctrl &= ~(1 << 7);
```

Checking a bit

To check whether a bit is set, we again use bitwise-AND. An (bit) value AND-ed with 1 will retain its original value.



Note: the result will either be 0 if our target bit is not set, or *non-zero* if it is set. Therefore, always compare the result of the bitwise-AND operation to zero

```
int main()
{
  auto const ctrl { reinterpret_cast<volatile uint8_t*>(0x40020000) };
  auto const cfg  { reinterpret_cast<volatile uint8_t*>(0x40020001) };
  auto const data { reinterpret_cast<volatile uint8_t*>(0x40020002) };

  *ctrl |= (1 << 7);             // Set bit 7

  if((*cfg & (1 << 4)) != 0)) {  // Always compare to zero.
    // ...
  }

  *ctrl &= ~(1 << 7);            // Clear bit 7
  ...
}
```

Read-only registers

As the name suggests a read-only register cannot be written to. Writing to a read-only register is undefined.

Rather than rely on programmer diligence we can get the compiler to help us by marking our read-only registers as pointers-to-`const`:

```
int main()
{
  auto const ro_reg { reintepret_cast<const volatile uint8_t*>(0x40020001) };

  auto val = *ro_reg;     // OK   - Read allowed.
  *ro_reg  = 1;           // FAIL - Write not allowed.
}
```

Hardware abstraction layers like CMSIS provide similar macros to define read-only and read-write registers

```
/* IO definitions (access restrictions to peripheral registers) */
/*CMSIS Global Defines
  IO Type Qualifiers are used:
   - to specify the access to peripheral variables.
   - for automatic generation of peripheral register debug
     information.
*/
#define __I volatile const  /* Defines 'read only'  permissions */
#define __O volatile        /* Defines 'write only' permissions */
#define __IO volatile       /* Defines 'read/write' permissions */
```

Write-only registers

A write-only register can only be written to. The value read from a write-only register is undefined. They are likely to be junk, and no reflection of the actual state of the register.

Therefore, the code idioms I've shown above should not be used with write-only registers. In fact, they could even be dangerous. Writing back a (modified) version of a junk value (from a read) could unintentionally enable bits in the register!

When accessing write-only registers only ever use the assignment operator (=)

```
int main()
{
  // Can we enforce write-only?
  //
  auto const wo_reg { reintepret_cast<volatile uint8_t*>(0x40020002) };

  auto val = *wo_reg;    // Will compile, but invalid
  *wo_reg  = 0x55;       // Never use |= to set bits
}
```

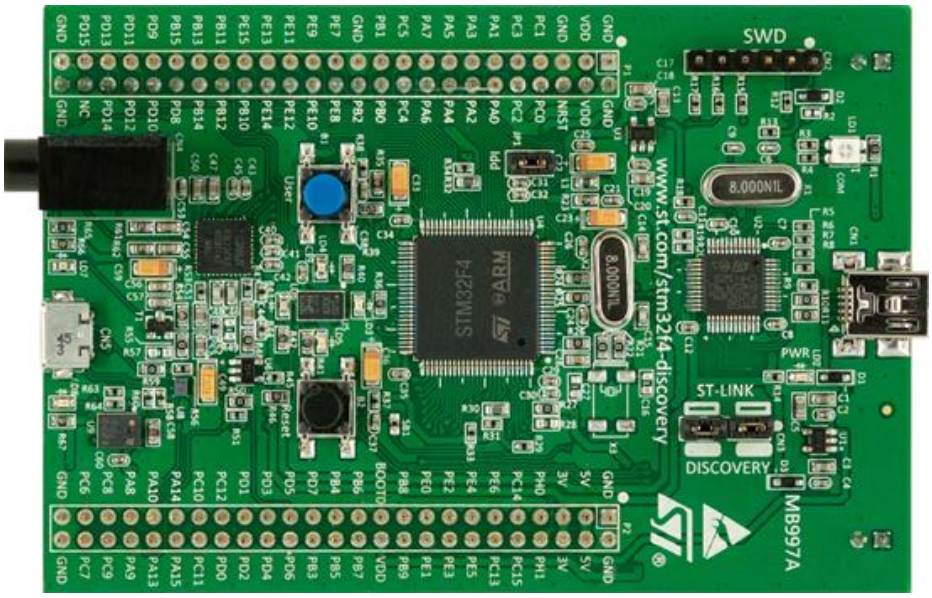
Unfortunately, unlike read-only registers there's no way with pointers of ensuring you never read the pointer (you can always read an object in C++). You are reliant on the programmer applying due diligence. (Notice in the CMSIS code above the definition for write-only is the same as for read-write!)

We will explore how C++ can help us enforce register read- and write- characteristics later in this document.

General-Purpose Input-Ouput (GPIO)

"Hello World" for embedded programmers

Although this paper is about general principles it's always nice to have a concrete example to work with. In this case I'm aiming my code at an STM32F4 Discovery board based on an ARM Cortex M4 processor.



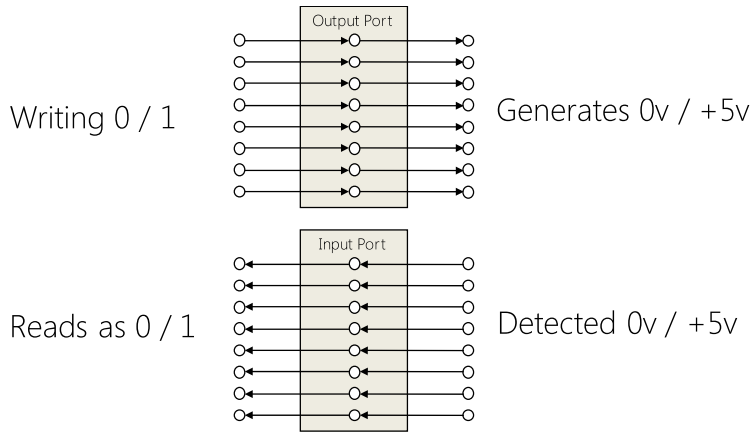
The classic "Hello World" for embedded programmers is to flash one of the LEDs on the target board.

On the STM Discovery board there are four LEDs. The LEDs are controlled via one of the General Purpose Input-Output (GPIO) ports on the microcontroller.

GPIO basics

Central to modern microcontrollers is the ability to perform general purpose I/O (GPIO). External pins can then be connected to physical hardware to sense or control voltages.

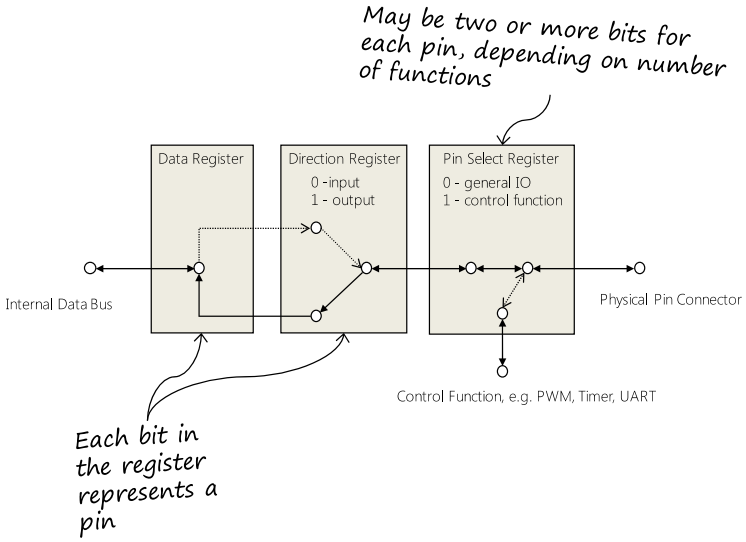
Conceptually, we could imagine each pin being represented by a bit in a hardware register. For example, we could have a output register where writing 1/0 to a bit would cause a 5V/0V output to be produced on the physical pin. Similarly, applying 5V/0V to an input pin would cause the appropriate bit in the input register to signal 1/0.



Such a mechanism would lead to a profusion of pins, and be very inflexible – there would always be a fixed number of input and output pins. In most real world microcontrollers each physical pin can be configured for a variety of uses. Usually the I/O pins are multiplexed so they can not only act as either inputs or outputs, but also perform alternative operations (e.g. transmit pin for RS232).

This means for a typical GPIO port there are multiple hardware registers involved in its use:

- A function-select register to specify whether the pin is being used for GPIO or some other function
- A direction register to specify whether the pin is an input or output
- One (or more) data registers for reading / writing data



The number of, and operation of, the GPIO registers is hardware-specific and will vary from manufacturer to manufacturer; but they typically all have the same basic principles.

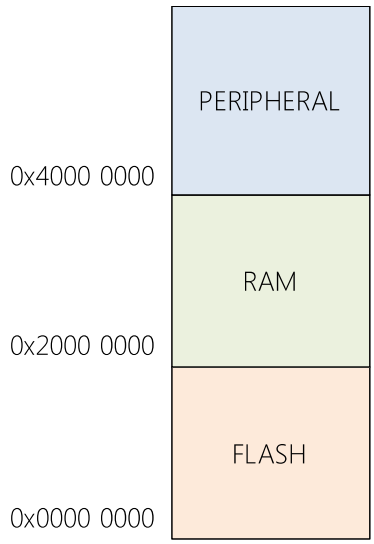
STM32F407VG GPIO

The STM32F407VG microcontroller has 9 identical GPIO ports, labelled A – I. Each GPIO port is highly configurable and each output pin can have several functions (usually digital input and output, analog; and up to 16 alternative functions)

Each peripheral on the STM32F407 is clock-gated. The clock signal does not reach the peripheral until we tell it to do so by way of setting a bit in a register (known as the Reset and Clock Control, or RCC, register). By default, clock signals never reach peripherals that are not in use, thus saving power.

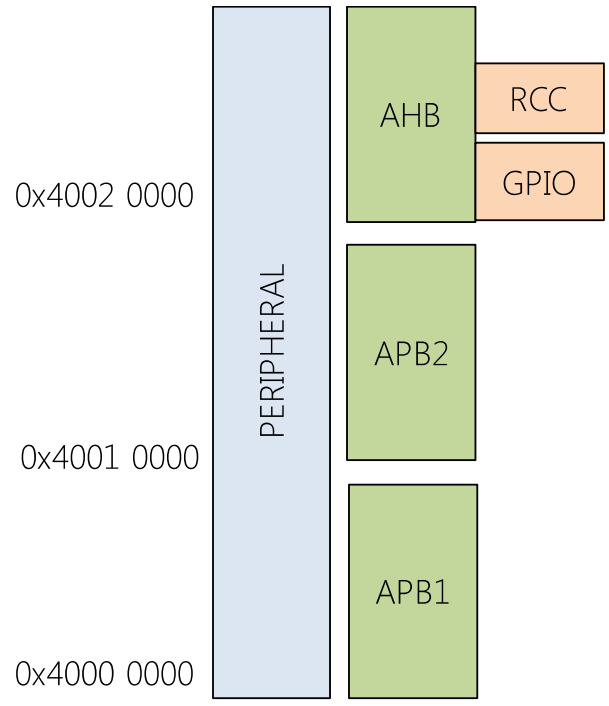
The memory map

One of the very convenient features of the ARM Cortex M architecture is that the memory map is well-defined. The basic memory map looks like this:



The STM32F407VG has three peripheral buses

- 1 Advanced High-Performance Bus (AHB)
- 2 Advanced Peripheral Bus (APB)



All the GPIO ports are on the Advanced High Performance bus. The addresses of the ports are shown in the table below

Port	Base Address
A	0x4002 0000
B	0x4002 0400
C	0x4002 0800
D	0x4002 0C00
E	0x4002 1000
F	0x4002 1400
G	0x4002 1800
H	0x4002 1C00
I	0x4002 2000

The GPIO hardware registers

There are 10 configuration / data registers for each GPIO port but for our purposes we only need to consider three –

- the Mode register, to configure
- the Input Data register
- the Output Data register.

	offset
Port mode register (GPIOx_MODER)	0x00
Port output type register (GPIOx_OTYPER)	0x04
Port output speed register (GPIOx_OSPEEDR)	0x08
Port pull-up/pull-down register (GPIOx_PUPDR)	0x0C
Port input data register (GPIOx_IDR)	0x10
Port output data register (GPIOx_ODR)	0x14
Port bit set/reset register (GPIOx_BSRR)	0x18
Port configuration lock register (GPIOx_LCKR)	0x1C
Port alternate function low register (GPIOx_AFRL)	0x20
Port alternate function high register (GPIOx_AFRH)	0x24

Notice that all registers are 32-bit (although in many cases not all 32 bits are used).

To keep the code simple we're going to do bare-minimum configuration and pretty much ignore good practices like error-checking, parameter validation, etc.

In a real-world system we would probably want to explicitly configure the output type, output speed and pull-up/pull-down settings for the port. The default settings for these registers are fine for this example, so to save space I'm going to ignore those registers.

There are three steps to getting our “Hello World” flashing LED:

- Declare our hardware register pointers
- Enable the GPIO port clock
- Configure the port for output
- Flash the LED by turning the pin on/off

Declaring the hardware registers

The LEDs on the STM32F4-Discovery are all on GPIO port D; on pins 12 – 15. For this exercise we’ll flash the blue LED (pin 15).

To add a (bare minimum) of flexibility I’ve declared the addresses of the hardware components as constant-expressions.

```
#include <cstdint>

using std::uint32_t;

constexpr auto RCC_addr  { 0x40023830 };
constexpr auto GPIO_addr { 0x40020C00 };

auto const RCC_AHB1ENR { reinterpret_cast<uint32_t*>(RCC_addr) };

auto const GPIO_MODER  { reinterpret_cast<volatile uint32_t*>(GPIO_addr + 0x00) };
auto const GPIO_IDR    { reinterpret_cast<volatile uint32_t*>(GPIO_addr + 0x10) };
auto const GPIO_ODR    { reinterpret_cast<volatile uint32_t*>(GPIO_addr + 0x14) };

int main()
{
  // ...
}
```

Enabling the GPIO port clock

Each device on the AHB1 bus is enabled using a special configuration register, the AHB1 RCC Enable Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reser-ved	OTGHS ULPIEN	OTGHSEN	ETHMACPTPEN	ETHMACRXEN	ETHMACTXEN	ETHMACEN	Reserved		DMA2EN	DMA1EN	CCMDATARAMEN	Res.	BKPSRAMEN	Reserved	
	rw	rw	rw	rw	rw	rw			rw	rw			rw		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CRCEN	Reserved			GPIOIEN	GPIOHEN	GPIOGEN	GPIOFEN	GPIOEEN	GPIODEN	GPIOCEN	GPIOBEN	GPIOAEN
			rw				rw	rw	rw	rw	rw	rw	rw	rw	rw

We could hard-code this value for our exercise (which would work just fine). As a generic solution it’s worth noting that bits [13:10] of the port’s address map onto the bit position in the RCC AHB1 Enable Register:

```
40020000 =>  0b0100’ 0000’ 0000’ 0010’ 0000’ 0000’ 0000’ 0000  => Port A
40020400 =>  0b0100’ 0000’ 0000’ 0010’ 0000’ 0100’ 0000’ 0000  => Port B
40020800 =>  0b0100’ 0000’ 0000’ 0010’ 0000’ 1000’ 0000’ 0000  => Port C
40020C00 =>  0b0100’ 0000’ 0000’ 0010’ 0000’ 1100’ 0000’ 0000  => Port D
```

Thus we can mask off those 4 bits of a port's address to work out which port number it is.

```
inline void enable_device(uint32_t address)
{
  // The 4 bits [13:10] identify the
  // port.
  //
  auto port_number = (address >> 10) & 0xOF;

  *RCC_AHB1ENR |= (1 << port_number);
}
```

Configuring the port for output

To enable a pin for output we must configure its port's mode.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Each pin has four modes of operation, thus requiring two configuration bits per pin:

00	Input
01	Output
10	Alternative function (configured via the AFRH and AFRL registers)
11	Analogue

We can generalise this into a simple function. Note that, because there are two configuration bits per pin, we must multiply the pin number by two to get the correct offset into the register.

```
inline void set_as_output(unsigned int pin_num)
{
  *GPIO_MODER |= (0b01 << (pin_num * 2));
}
```

Flashing the LED

Flashing the LED requires turning on / off the appropriate pin. A pair of functions will suffice. Notice, unlike the mode configuration, controlling a pin only requires one bit (on/off).

```
inline void turn_on_pin(unsigned int pin_num)
{
  *GPIO_ODR |= (1 << pin_num);
}

inline void turn_off_pin(unsigned int pin_num)
{
  *GPIO_ODR &= ~(1 << pin_num);
}
```

Finally, our `main()` function puts all this together:

```
int main()
{
  enable_device(GPIO_addr);

  set_as_output(15);         // Pin 15 is the blue LED

  while(true) {
    turn_on_pin(15);
    sleep(1000);             // Some generic busy-wait...

    turn_off_pin(15);
    sleep(1000);
  }
}
```

Under the hood

Let's have a quick look at the generated assembler for this code. For this example I'm using the ARM gcc compiler toolchain, generating Thumb2 instructions. I've turned the optimiser off (`00`).

I don't want to dwell on this code much here. Since this is pretty much the minimum code we could write to get our "Hello World" working it's a useful benchmark for any code we generate later.

In the code below I've removed the opcodes for the call to the generic `sleep()` function since they aren't really relevant to what we're examining.

```
; main()
;
08000d14:   ldr     r2, [pc, #36]       ; r2 = 0x40023830 <RCC_AHB1ENR>
08000d16:   ldr     r3, [r2, #0]        ; r3 = *r2
08000d18:   orr.w   r3, r3, #8          ; r3 = r3 | 0x08
08000d1c:   str     r3, [r2, #0]        ; *r2 = r3

; *GPIO_MODER |= (0b01 << (pin_num * 2));
;
08000d1e:   ldr     r2, [pc, #32]       ; r2  = 0x40020C00 <GPIO_MODER>
08000d20:   ldr     r3, [r2, #0]        ; r3  = *r2
08000d22:   orr.w   r3, r3, #1073741824 ; r3  = r3 | 0x40000000
08000d26:   str     r3, [r2, #0]        ; *r2 = r3

; while(true) {
; *GPIO_ODR |= (1 << pin_num);
;
loop:
08000d28:   ldr     r2, [pc, #24]       ; r2  = 0x40020C14 <GPIO_ODR>
08000d2a:   ldr     r3, [r2, #0]        ; r3  = *r2
08000d2c:   orr.w   r3, r3, #32768      ; r3  = r3 | 0x8000
08000d30:   str     r3, [r2, #0]        ; *r2 = r3

; *GPIO_ODR &= ~(1 << pin_num);
;
08000d32:   ldr     r3, [r2, #0]        ; r3  = *r2 <GPIO_ODR>
08000d34:   bic.w   r3, r3, #32768      ; r3  = r3 & ~0x8000
08000d38:   str     r3, [r2, #0]        ; *r2 = r3
; }
;
08000d3a:   b.n     0x8000d28           ; goto loop
                                        ; Register addresses:
08000d3c:   dcd     1073887280          ; 0x40023830
08000d40:   dcd     1073875968          ; 0x40020C00
08000d44:   dcd     1073875988          ; 0x40020C14
```

An object-based approach

Hardware devices lend themselves nicely to an object-based approach. Each hardware device has an equivalent software object for accessing it; there is a one-to-one mapping between devices and objects.

Encapsulating device access within a class gives us a number of other benefits:

- The member functions decouple the actual access from the application. We can easier separate interface (API) from implementation
- Constructors can be used to initialise the device, removing the need for the client to explicitly do so. Destructors can be used to place the device back into a 'safe' state.
- We can support multiple devices.
- We can use specialisation (inheritance) to provide 'families' of devices.

Interface design

Before we look at implementation it's worth taking some time to explore our client interface (API). Firstly, there's no such thing as a 'perfect' API. All interface designs will make assumptions and compromises; design is the act of balancing these things.

We (generally) want to provide an interface that is simpler than the underlying implementation (otherwise, what would be the point?). This means not only encapsulating implementation details, but also restricting the allowable behaviour of the device. As a rule-of-thumb, the more fine-grained control you grant clients the less portable your API is. That is, allowing fine-grain control generally requires exposing more of the underlying implementation, which is the non-portable part.

For our GPIO class we have a couple of options:

Bitmask-based member functions

In this interface design the client provides bit masks that specify which bits are to be set / cleared. This interface allows multiple bits to be set / cleared in one call.

Clients can perform bitwise operations on the GPIO device as if it were a single abstract memory location; the fact that there are multiple registers being manipulated is hidden from the client.

```
class GPIO {
public:
  void direction(std::uint32_t bitmask);

  GPIO& operator=(std::uint32_t bitmask);
  operator uint32_t();

  GPIO& operator &= (std::uint32_t bitmask);
  GPIO& operator |= (std::uint32_t bitmask);
};
```

From a client perspective using the GPIO class looks a lot like using a pointer-based technique.

```
int main()
{
  // Create a GPIO object, port_D (see later
  // for a discussion on object construction)

  port_D.direction(1 << 15);

  while(true) {
    port_D |= (1 << 15);
    sleep(1000);

    port_D &= ~(1 << 15);
    sleep(1000);
  }
}
```

Pin-based member functions

Here we are restricting clients to manipulating one GPIO pin at a time. We could perhaps restrict clients even more by using an enumeration for the pins. For example:

```
class GPIO {
public:
  enum Pin {
    Pin00, Pin01, Pin02, Pin03,
    Pin04, Pin05, Pin06, Pin07,
    Pin08, Pin09, Pin10, Pin11,
    Pin12, Pin13, Pin14, Pin15,
  };

  void set_as_output(Pin pin);
  void set_as_input (Pin pin);

  void set_pin  (Pin pin);
  void clear_pin(Pin pin);
  bool is_set   (Pin pin);
};
```

This interface design removes the need for bit calculations from the client.

```
int main()
{
  // Create the GPIO object, port_D...

  port_D.set_as_output(Pin15);

  while(true) {
    port_D.set(Pin15);
    sleep(1000);

    port_D.clear(Pin15);
    sleep(1000);
  }
}
```

The Pin class

This idea could be taken further by defining a GPIO Pin class that represents a single pin on a port.

```
class Pin {
public:
```

```
  Pin(Port port, unsigned int number);

  void set();
  void clear();
  bool is_set();

  Pin& operator=(unsigned int val);
  operator unsigned int();
};
```

This is the most 'abstract' API, treating each physical pin as a one-bit integer.

```
int main()
{
  // Create a Pin object, blue_LED...

  blue_LED.direction(Pin::OUTPUT);

  while(true) {
    blue_LED = 1;
    sleep(1000);

    blue_LED = 0;
    sleep(1000);
  }
}
```

Choosing an interface

Firstly, there's no such thing as a 'perfect' API. All interface designs will make assumptions and compromises; design is the act of balancing these things.

As a rule-of-thumb, the more fine-grained control you grant clients the less portable your API is. That is, allowing fine-grain control generally requires exposing more of the underlying implementation, which is the non-portable part.

For individual bit-manipulation a pin-based (or Pin class) interface is generally easier to use. However, if your hardware requires multiple bits to be set a bitmask-based interface is more useful. Often, you can't know in advance which mechanism will be most valuable to the client so you provide a 'combination' interface; for example adding functions for both pin-based and bitmask-based calls.

For this article I'm going to use the pin-based GPIO API since the member function implementations are very close to the code we have seen in previous articles. I'll leave it as a perennial 'exercise for the reader' to implement the other APIs.

Construction and destruction

There are two problems we need to solve with our GPIO class constructor

- Identifying a unique hardware port
- Configuring the hardware for use

GPIO objects map explicitly (and, ideally, exclusively) onto a single hardware device. We could use the hardware address for the device. This is certainly unique but requires the client correctly remember a set of arcane, and target-specific, numbers in order to use the device.

Better, perhaps, would be to use an abstract identifier – an enumeration – for the device.

```
namespace STM32F407 {

  enum device {
    GPIO_A, GPIO_B, GPIO_C,
    GPIO_D, GPIO_E, GPIO_F,
```

```
      GPIO_G, GPIO_H, GPIO_I
    };
}
```

I'm putting device-specific elements (there will be more, shortly) in a namespace to separate it.

We can now use this enumeration as a constructor parameter:

```
class GPIO {
public:
  explicit GPIO(STM32F407::device id);

  // Other API...
};
```

Note the constructor is marked `explicit`. This is to stop implicit conversion of (random!) integers to `GPIO` objects.

Inside the `GPIO` constructor, we need to enable the clock to the particular GPIO hardware device. A simple function will suffice for this.

```
namespace STM32F407 {

  enum device {
    GPIO_A, GPIO_B, GPIO_C,
    GPIO_D, GPIO_E, GPIO_F,
    GPIO_G, GPIO_H, GPIO_I
  };

  constexpr std::uint32_t periph_base { 0x40020000 };


  inline void enable_device(device dev)
  {
    auto const enable_reg { reinterpret_cast<volatile std::uint32_t*>(periph_base + 0x3830) };

    *enable_reg |= (1 << dev);
  }
}
```

The call to this function can be made within the `GPIO` constructor.

```
GPIO::GPIO(STM32F407::device dev)
{
  STM32F407::enable_device(dev);
}
```

The destructor on the `GPIO` could do nothing (the default); or it place the device back into a safe / quiescent state. In our system one option would be to disable the clock to the hardware device. A complementary function to the `enable_device()` function above could be simply written. Note, however, the `GPIO` class has to store its device ID as a member in order to call the function.

```
class GPIO {
public:
  explicit GPIO(STM32F407::device dev);
  ~GPIO();

  // Other API...

private:
  STM32F407::device ID;
};


GPIO::GPIO(STM32F407::device dev) :
 ID { dev }
{
  STM32F407::enable_device(ID);
}
```

```
GPIO::~GPIO()
{
  STM32F407::disable_device(ID);
}
```

Copy and move policy

Since there is a one-to-one mapping between objects and the physical hardware we have to think carefully about copying (and, by extension, moving) objects.

What does it mean to 'copy' a hardware device? It could mean:

- Two GPIO objects both refer to the same hardware device (the default copy behaviour)
- All the configuration of one hardware port is copied to another
- All the configuration settings and current output values are copied

None of these options is particularly desirable (or safe), so our 'best' option is to disable copying; and moving, for similar reasons.

```
class GPIO {
public:
  explicit GPIO(STM32F407::device dev);
  ~GPIO();

  // Copy and move policy
  //
  GPIO(const GPIO&)            = delete;
  GPIO(GPIO&&)                 = delete;
  GPIO& operator=(const GPIO&) = delete;
  GPIO& operator=(GPIO&&)      = delete;
};
```

When things go wrong

A crucial part of a class's interface design is what happens when things go wrong. That is, what is the mechanism by which you report success / failure on an object's behaviour. Within an operation there are generally two places you will apply error checking

- Pre-condition validation – The input parameters to the operation are not acceptable
- Post-condition validation – The resulting behaviour of the operation is outside some acceptable bounds

When an error condition occurs you have four options:

- Do nothing
- Return an error
- Throw an exception
- Terminate

Each of these options has a different level of consequence.

Do nothing

Silently ignoring failures when manipulating hardware could be potentially catastrophic to your system. However, if it is known that the exception does not affect system operation it can be safely ignored without consequence. In other cases the error condition may be transient (for example, during a system mode change) and it is understood that the error condition will not persist.

Return an error

Returning an error code gives a mechanism for our hardware manipulation code to report back its status.

The design of a function's interface can affect the explicitness of error handling. For example, consider the two operations on our GPIO class

```
class GPIO {
public:
  error_code set_pin(Pin pin);

  void clear_pin(Pin pin, error_code& err);
};
```

For `set_pin()` the error code, being the return value from the function, can be implicitly ignored. This gives the potential to miss error conditions, which could lead to failures, hazardous system conditions and all sorts of undesirable consequences.

C++17 gives us a mechanism to prevent the ignoring of error return codes. Marking our function with the `[[nodiscard]]` attribute encourages the compiler to emit a warning if the return value is not used.

```
class GPIO {
public:
  [[nodiscard]] error_code set_pin(Pin pin);
  ...
};


int main()
{
  GPIO port_D { STM32F407::GPIO_D };

  port_D.set_pin(GPIO::Pin15);      // WARNING: error_code ignored
}
```

For `clear_pin()` the error condition, passed as an input-output parameter, must be supplied by the client (caller) and the compiler will give a warning if the error code is not subsequently read; unless it is explicitly 'ignored' by the caller.

Up to C++17 to ignore the unused `error_code` we would explicitly cast it to void to prevent a compiler warning. From C++17 we can hint to the compiler that we may not be checking the `error_code`.

```
class GPIO {
public:
  void clear_pin(Pin pin, error_code& err);
};


int main()
{
  [[maybe_unused]] error_code err { };  // We have to supply an
                                        // error_code object, but
                                        // we aren't interested
                                        // in it.

  GPIO port_D { STM32F407::GPIO_D };
  port_D.clear_pin(GPIO::Pin15, err);
  ...
}
```

Notice, in the case of operator overloads there is no way of returning error codes (without breaking the semantics of the operator overload).

Global error objects

An alternative to error codes, then, is to use some global error condition object that can be updated and checked by clients.
Use of global error objects (and global objects in general) is frowned upon in modern programming, so we'll say no more of it here.

Throw an exception

Throwing an exception provides a couple of benefits over error codes. We can exploit the exception handling mechanism of the language to route error conditions to where they can be most appropriately handled. Client code becomes more explicit, separating behaviour code from failure code. Also, the behavioural API does not need to be cluttered with error handling

However, the cost of this is an increase in program size; and exception handling is non-deterministic and slow (when exceptions are thrown; there should be no little, or no, run-time cost if exceptions are not thrown). For this reason many embedded systems explicitly disable exception handling.

More subtly, but arguably far more important, exception handling strategies must be built into your code from the ground up. Simply adding ad-hoc exceptions to pre-existing code is likely to wreak havoc on the performance, maintainability and extensibility of your system.

Terminate

This is, of course, the extreme option. Use assert-like code to terminate code on error conditions. In application code, terminate-on-error will likely lead to code that is frustratingly cumbersome for clients to use. However, hardware is far more pernickety. There is little tolerance or error – the hardware is either working or it isn't. Terminating code is a 'reasonable' option with hardware manipulation; particularly with respect to pre-condition validation.

Implementation

Implementation options

There are three mainstream approaches to hardware access available to us:

- Nested pointers / references
- Pointer offsets
- Structure overlay

Nested pointers are the simplest, and probably most common, approach but can have some memory overhead costs. Pointer offsets and structure overlay are more memory-efficient but can have some shortcomings if not implemented carefully.

Nested pointers / references

Nested pointers, as the name suggests, involves storing a pointer to each hardware register as a private member within the class.

```
class GPIO {
public:
  explicit GPIO(STM32F407::device dev);
  ~GPIO();

  // Copy and move policy...

  // Behavioural API...

private:
  STM32F407::device ID;

  volatile std::uint32_t* const mode;
  volatile std::uint32_t* const type;
  volatile std::uint32_t* const speed;
  volatile std::uint32_t* const pull_up_down;
  volatile std::uint32_t* const input_data;
  volatile std::uint32_t* const output_data;
  volatile std::uint32_t* const set_reset;
  volatile std::uint32_t* const lock;
  volatile std::uint32_t* const alt_fn_low;
  volatile std::uint32_t* const alt_fn_high;
};
```

Note, the pointers are declared as `const`. This means the default copy constructor and assignment operator are not available; but since we declared these operations as deleted this does not affect the design of this class.

Since the pointers are constants they must be initialised in the `GPIO` constructor. At the moment, though, we don't have an address to 'force' into the pointers; only an enumeration identifying the device.

Luckily, in our case there is a direct mapping between the device's ID and its address in memory. We can construct a simple conversion function to do the mapping.

```
namespace STM32F407 {

  inline constexpr uint32_t device_address(device dev)
  {
    return peripheral_base + (dev << 10);
```

```
  }
}
```

We can use this function when constructing the class:

```
using std::uint32_t;
using STM32F407::device_address;

GPIO::GPIO(STM32F407::device dev) :
  ID           { dev },
  //                                          NOTE: This is NOT pointer
  //                                                arithmetic!
  //                                                    |
  mode         { reinterpret_cast<uint32_t*>(device_address(dev) + 0x00) },
  type         { reinterpret_cast<uint32_t*>(device_address(dev) + 0x04) },
  speed        { reinterpret_cast<uint32_t*>(device_address(dev) + 0x08) },
  pull_up_down { reinterpret_cast<uint32_t*>(device_address(dev) + 0x0C) },
  input_data   { reinterpret_cast<uint32_t*>(device_address(dev) + 0x10) },
  output_data  { reinterpret_cast<uint32_t*>(device_address(dev) + 0x14) },
  set_reset    { reinterpret_cast<uint32_t*>(device_address(dev) + 0x18) },
  lock         { reinterpret_cast<uint32_t*>(device_address(dev) + 0x1C) },
  alt_fn_low   { reinterpret_cast<uint32_t*>(device_address(dev) + 0x20) },
  alt_fn_high  { reinterpret_cast<uint32_t*>(device_address(dev) + 0x24) }
{
  STM32F407::enable_device(ID);
}
```

A small point to note here: Programmers unused to the above notation often make the mistake of thinking the addition in the pointer initialisers is actually pointer arithmetic. The addition is done as (unsigned) integers, *then* cast to a pointer type.

The behavioural member functions of the class can now be implemented in much the same way as we have done previously; for example:

```
void GPIO::set_pin(GPIO::Pin pin)
{
  *output_data |= (1 << pin);
}


void GPIO::clear_pin(GPIO::Pin pin)
{
  *output_data &= ~(1 << pin);
}
```

The client code is very clean.

```
int main()
{
  GPIO port_d { STM32F407::GPIO_D };

  port_d.set_as_output(GPIO::Pin15);

  while(true) {
    port_d.set_pin(GPIO::Pin15);
    sleep(1000);

    port_d.clear_pin(GPIO::Pin15);
    sleep(1000);
  }
}
```

From a performance perspective the code looks very similar to our earlier example.

```
; main() {
;
08000d78:   push    {lr}
08000d7a:   sub     sp, #52             ; Allocate memory for GPIO

; GPIO port_d { STM32F407::GPIO_D };
;
08000d7c:   add     r0, sp, #4          ; r0 = &port_d
08000d7e:   movs    r1, #3              ; r1 = STM32F407::GPIO_D
08000d80:   bl      0x8000cf8           ; GPIO::GPIO()

; port_d.set_as_output(GPIO::Pin15);
;
08000d84:   ldr     r2, [sp, #8]        ; r2  = mode
08000d86:   ldr     r3, [r2, #0]        ; r3  = *r2;
08000d88:   orr.w   r3, r3, #1073741824 ; r3  = r3 | 0x40000000
08000d8c:   str     r3, [r2, #0]        ; *r2 = r3

; while (true) {
; port_d.set_pin(GPIO::Pin15);
;
loop:
08000d8e:   ldr     r2, [sp, #28]       ; r2  = output_data
08000d90:   ldr     r3, [r2, #0]        ; r3  = *r2
08000d92:   orr.w   r3, r3, #32768      ; r3  = r3 | 0x8000
08000d96:   str     r3, [r2, #0]        ; *r2 = r3

; port_d.clear_pin(GPIO::Pin15);
;
08000d98:   ldr     r2, [sp, #28]       ; r2  = output_data
08000d9a:   ldr     r3, [r2, #0]        ; r3  = *r2
08000d9c:   bic.w   r3, r3, #32768      ; r3  = r3 & ~0x8000
08000da0:   str     r3, [r2, #0]        ; *r2 = r3

08000da2:   b.n     0x8000d8e           ; goto loop
; }
```

One of the drawbacks of the nested pointer approach is the amount of memory required for each GPIO object. In the case of our example each 32-bit hardware register has a software analogue in the form of a 32-bit pointer. This seems reasonable. However, if our hardware consisted of 8-bit registers our software object would be four times the size the hardware device it accessed.

Of course, we could omit some of these pointers if we are not exposing the behaviour they allow (for example, only using the default output type and speed).

Pointer offsets

An alternative, and more memory-efficient, implementation can exploit the fact that the hardware registers are always at fixed offsets from some base address. We can store a single pointer and use pointer arithmetic to access individual registers. The code can be made more readable by using an enumeration.

```
class GPIO {
public:
  explicit GPIO(STM32F407::device dev);
  ~GPIO();

  // Copy and move policy...

  // Behavioural API...

private:
  STM32F407::device ID;

  volatile uint32_t* const registers;  // Base address

  enum Offset {                        // Offsets
    mode,
    type,
    speed,
    pull_up_down,
    input_data,
    output_data,
    set_reset,
    lock,
    alt_fn_low,
    alt_fn_high
  };
};
```

The constructor is commensurately simpler, also.

```
GPIO::GPIO(STM32F407::device dev) :
  ID        { dev },
  registers { reinterpret_cast<uint32_t*>(device_address(dev)) }

{
  STM32F407::enable_device(ID);
}
```

In order to access individual registers we now have to perform pointer arithmetic on the base address.

```
void GPIO::set_pin(GPIO::Pin pin)
{
  *(registers + output_data) |= (1 << pin);
}


void GPIO::clear_pin(GPIO::Pin pin)
{
  *(registers + output_data) &= ~(1 << pin);
}
```

The readability of the code is starting to suffer now (to say the least). We can make an improvement by exploiting the relationship between pointer arithmetic and the index operator

```
void GPIO::set_pin(GPIO::Pin pin)
{
  registers[output_data] |= (1 << pin);
}


void GPIO::clear_pin(GPIO::Pin pin)
{
```

```
    registers[output_data] &= ~(1 << pin);
}
```

We've made a potentially significant improvement in our memory-efficiency now: irrespective of the number of registers the size of the object remains the same – a single pointer (plus any additional management data).

Run-time performance is not affected, either:

```
; main() {
;
08000d44:   push    {lr}
08000d46:   sub     sp, #12             ; Allocate memory for port_d

; GPIO port_d { STM32F407::GPIO_D };
;
08000d48:   mov     r0, sp              ; r0 = &port_d
08000d4a:   movs    r1, #3              ; r1 = STM32F407::GPIO_D
08000d4c:   bl      0x8000cf8           ; GPIO::GPIO()

; port_d.set_as_output(GPIO::Pin15);
;
08000d50:   ldr     r2, [sp, #4]        ; r2 = registers
08000d52:   ldr     r3, [r2, #0]        ; r3 = registers->mode
08000d54:   orr.w   r3, r3, #1073741824 ; r3 = r3 | 0x40000000
08000d58:   str     r3, [r2, #0]        ; registers->mode = r3

; while(true) {
; port_d.set_pin(GPIO::Pin15);
;
loop:
08000d5a:   ldr     r3, [sp, #4]        ; r3 = registers
08000d5c:   ldr     r2, [r3, #20]       ; r2 = registers->output_data
08000d5e:   orr.w   r2, r2, #32768      ; r2 = r2 | 0x8000
08000d62:   str     r2, [r3, #20]       ; registers->output_data = r2

; port_d.clear_pin(GPIO::Pin15);
;
08000d64:   ldr     r2, [r3, #20]       ; r2 = registers->output_data
08000d66:   bic.w   r2, r2, #32768      ; r2 = r2 & ~0x8000
08000d6a:   str     r2, [r3, #20]       ; registers->output_data = r2
08000d6c:   b.n     0x8000d5a           ; goto loop
08000d6e:   nop
; }
```

Structure overlay

The main limitation of the pointer offset implementation is that all registers must be the same size, and/or any offsets between registers (if they are not contiguous) must be the some multiple of the register size. This is because the pointer offset implementation basically treats the hardware memory as an array.

In our example the pointer offset implementation is a viable option; but that is not always the case. It is possible (although less likely these days) that you have different-sized registers, or registers at odd offsets. For these situations a structure overlay is a good option.

Structure overlay make uses of the fact that the type of a pointer defines not only how much memory to read but also how to interpret it. Until now we have been using pointers to scalar types – unsigned integers. There is nothing to stop us declaring a pointer to a user-defined type, with multiple members (in other words, a class or structure)

If we can define a structure that matches our hardware register layout we can 'overlay' this structure on memory by declaring a pointer to the struct type and 'forcing' an address into the pointer.

A big word of warning here:

By default, the compiler is free to insert padding into a structure's layout to word-align the members for more efficient access. This can mean that actual structure has a different size and member offsets to the structure you declared. This will be invisible from the code; and a pain to debug.

When using structures for hardware overlay ALWAYS pack the structures (that is, force the compiler to remove any padding). Unless you can guarantee that your structure will never be padded.

Unfortunately, structure packing is not part of the C++ standard so the packing instruction is always compiler-specific.

In our example our registers are all 32-bit and contiguous in memory; therefore we can be comfortable that we will have no padding issues.

Here's the class declaration

```
class GPIO {
public:
  explicit GPIO(STM32F407::device dev);
  ~GPIO();

  // Copy and move policy...

  // Behavioural API...

private:
  STM32F407::device ID;

  // Overlay structure
  //
  struct Registers {
    std::uint32_t mode;
    std::uint32_t type;
    std::uint32_t speed;
    std::uint32_t pull_up_down;
    std::uint32_t input_data;
    std::uint32_t output_data;
    std::uint32_t set_reset;
    std::uint32_t lock;
    std::uint32_t alt_fn_low;
    std::uint32_t alt_fn_high;
  };

  volatile Registers* const registers;
};
```

Notice the struct declaration is a private declaration within the GPIO class. Since this is a declaration it does not add to the size of a GPIO object.

The constructor has to change. Note we're now casting our address to a pointer-to-structure.

```
GPIO::GPIO(STM32F407::device dev) :
  ID        { dev },
  registers { reinterpret_cast<Registers*>(device_address(dev)) }

{
  STM32F407::enable_device(ID);
}
```

In the member functions we can make use of the pointer-to-member operator. The compiler will automatically calculate the member offsets from the base address

```
void GPIO::set_pin(GPIO::Pin pin)
{
  registers->output_data |= (1 << pin);
}


void GPIO::clear_pin(GPIO::Pin pin)
{
  registers->output_data &= ~(1 << pin);
}
```

From a memory perspective, the structure overlay implementation has the same footprint as the pointer-offset implementation.

From a code-performance perspective?

```
; main() {
;
08000d44:   push    {lr}
08000d46:   sub     sp, #12             ; Allocate memory for port_d

; GPIO port_d { STM32F407::GPIO_D };
;
08000d48:   mov     r0, sp              ; r0 = &port_d
08000d4a:   movs    r1, #3              ; r1 = STM32F407::GPIO_D
08000d4c:   bl      0x8000cf8           ; GPIO::GPIO()

; port_d.set_as_output(GPIO::Pin15);
;
08000d50:   ldr     r2, [sp, #4]        ; r2 = registers
08000d52:   ldr     r3, [r2, #0]        ; r3 = registers->mode
08000d54:   orr.w   r3, r3, #1073741824 ; r3 = r3 | 0x40000000
08000d58:   str     r3, [r2, #0]        ; registers->mode = r3

while(true) {
; port_d.set_pin(GPIO::Pin15);
;
loop:
08000d5a:   ldr     r3, [sp, #4]        ; r3 = registers
08000d5c:   ldr     r2, [r3, #20]       ; r2 = registers->output_data
08000d5e:   orr.w   r2, r2, #32768      ; r2 = r2 | 0x8000
08000d62:   str     r2, [r3, #20]       ; registers->output_data = r2

; port_d.clear_pin(GPIO::Pin15);
;
08000d64:   ldr     r2, [r3, #20]       ; r2 = registers->output_data
08000d66:   bic.w   r2, r2, #32768      ; r2 = r2 & ~0x8000
08000d6a:   str     r2, [r3, #20]       ; registers->output_data = r2
08000d6c:   b.n     0x8000d5a           ; goto loop
08000d6e:   nop
;  }
```

Looks pretty familiar, doesn't it?

One final option with the structure overlay implementation: we can hide our implementation using the Pointer-to-Implementation (pImpl) idiom. Since we are simply declaring a pointer to a structure inside the class declaration the compiler is happy for that to be a pointer to an incomplete (that is, not-yet-defined) structure. The structure itself can be defined inside the implementation file.

```
class GPIO {
public:
  explicit GPIO(STM32F407::device dev);
  ~GPIO();

  // Copy and move policy...

  // Behavioural API...

private:
  STM32F407::device ID;

  // Pointer to incomplete type;
  // struct Registers must be defined in
  // the .cpp file
  //
  volatile struct Registers* const registers;
};
```

There is a price to pay for this, though. Our member functions are currently inlined for speed, and hence in the header file. Since the `Registers` structure is not defined in the header file any more we cannot refer to any of its members. Therefore, we cannot inline our member functions.

Placement new

Structure overlay

If you're accessing a hardware I/O device with multiple registers it's very convenient (and efficient) to do so with a structure overlay. A structure is defined matching the layout of the hardware registers and this is 'overlaid' onto I/O memory using a pointer:

```
struct UART_Registers {
  std::uint8_t tx;
  std::uint8_t ctrl;
  std::uint8_t rx;
  std::uint8_t status;
};
```

(For the purposes of this example I'm going to assume 8-bit registers, consecutively in memory. This eliminates some of the potential pitfalls of structure overlay.

Of course, with a `struct` clients are free to manipulate any of the registers in ways that might not be appropriate or safe

```
int main()
{
  auto const uart_ptr { reinterpret_cast<volatile UART_Registers*>(0x40021000) };

  uart_ptr->ctrl = 0b00000001;  // Is this valid?

  uint8_t data {  };
  ...
  uart_ptr->rx   = data;        // What does this do?!
}
```

Class overlay

Uncontrolled access to underlying hardware is unlikely to be a robust solution. Using a class allows us to provide encapsulated access to the hardware via member functions (the actual implementation of our UART is unimportant for this article).

```
class UART {
public:
  UART();
  ~UART();

  uint8_t read();
  void    write(uint8_t data);

private:
  std::uint8_t tx;
  std::uint8_t ctrl;
  std::uint8_t rx;
  std::uint8_t status;
};
```

```
UART::UART()
{
  // Initialise and configure
  // hardware for use...
}


uint8_t UART::read()
{
  // Wait for data...
  //
  while((status & (1 << 3)) == 0){ }

  return rx;
}
```

We can once again perform structure overlay. Now, we have protection for our registers and an abstract API to program to.

```
int main()
{
  auto const uart { reinterpret_cast<volatile UART*>(0x40021000) };

  uart->ctrl = 0b00000001;  // FAIL! => ctrl is private

  uint8_t data {  };
  ...
  data = uart->read();      // FAIL!  Eh?
}
```

It should be no surprise that the first use of `uart_ptr` fails – we are attempting to access a private member of the `UART` class.

The second failure is more perplexing. The `read()` function is marked as public, so this isn't the issue. The error message is also confusing:

```
no known conversion for implicit 'this' parameter from 'volatile UART*' to 'UART*'
```

cv-qualifers means const AND volatile

If you're familiar with `const`-correctness in your code this error should look familiar (If you're not familiar with `const`-correctness, here's a good overview - https://isocpp.org/wiki/faq/const-correctness#overview-const)

When we build `const`-correct classes we mark member functions as `const` if the only inspect the owning object; not modify it. Technically, marking a member function as `const` changes the type of the 'this' pointer to a `const`-pointer-to-`const`-type.

The thing is, this also holds true for `volatile` objects (that's why they're known as the *cv-qualifiers* and not the *c-qualifer* and *v-qualifer*!)

When we did our structure (class) overlay we declared the this pointer as a `volatile UART*`. This is perfectly correct – since we are overlaying on hardware we don't want the compiler to optimise away any read/write operations.

Just as you cannot call a non-`const` member function on a `const` object, you cannot call a non-`volatile` member function on a `volatile` object!

Bizarre but true!

To fix this we would have to modify our `UART` class.

```
class UART {
public:
  UART();
  ~UART();

  uint8_t read() volatile;
  void    write(uint8_t data) volatile;

private:
  std::uint8_t tx;
  std::uint8_t ctrl;
  std::uint8_t rx;
  std::uint8_t status;
};


UART::UART()
{
  // Initialise and configure
  // hardware for use...
}


uint8_t UART::read() volatile
{
  // Wait for data...
  //
  while((status & (1 << 3)) == 0) { }

  return rx;
}
```

Now we get the behaviour we were expecting.

```
int main()
{
  auto const uart { reinterpret_cast<volatile UART*>(0x40021000) };

  uart->ctrl = 0b00000001;  // FAIL! As expected

  uint8_t data {  };
  ...
  data = uart->read();      // OK
}
```

When we declare an object as `volatile` (either in its declaration or via a pointer) we are saying that all attributes of the class must be treated as `volatile` objects. Perhaps a cleaner way to specify this is to explicitly `volatile`-qualify our class members instead.

```
class UART {
public:
  UART();
  ~UART();

  uint8_t read();
```

```
    void    write(uint8_t data);

  private:
    volatile std::uint8_t tx;
    volatile std::uint8_t ctrl;
    volatile std::uint8_t rx;
    volatile std::uint8_t status;
  };
```

```
int main()
{
  auto const uart { reinterpret_cast<UART*>(0x40021000) };

  uart->ctrl = 0b00000001;     // FAIL! As expected

  uint8_t data {  };
  ...
  data = uart->read();         // OK
}
```

Placement new

Our code now compiles (Quick! Ship it!) but it probably still won't work properly. The reason? The underlying hardware is not being initialised.

The initialisation code for our UART is (quite reasonably) in its constructor code. However, the constructor for the `UART` class is never being called!

Let's review this line of code:

```
auto const uart { reinterpret_cast<volatile UART*>(0x40021000) };
```

This code effectively says "treat the address 0x40021000 *as if there was a `UART` object at that location*". At no point is that `UART` object ever constructed (and hence initialised).

Enter *placement new*.

The default operator new allocates memory for an object, then calls its constructor. Placement new is an overload of `operator new` that does not allocate memory, but just constructs an object at a – provided – memory location.

The signature of placement new is:

```
void* operator new(std::size_t, void*)
```

Note the placement new takes a second parameter – the address to construct the object at. It can be called like this

```
class ADT {
  // ...
};

static uint8_t pool[64];


int main()
{
  ADT* const adt { new(reinterpret_cast<void*>(pool)) ADT { } };
  ...
}
```

(Note: To be more correct, I should have used `static_cast` in the above code. I've stuck with `reinterpret_cast` to avoid confusion with the rest of the paper)

We can make use of placement new to perform structure overlay on our hardware, and guarantee the constructor is called.

```
int main()
{
  auto const uart { new (reinterpret_cast<void*>(0x40021000)) UART { } };

  uart->read();
  ...
}
```

Notice we can't use `constexpr` here, since the `UART` is (as far as the language is concerned) a dynamic, not a compile-time, object (and we can't use `reinterpret_cast<>` in constant-expressions).

What about placement delete?

If we are using `new`, the usual idiom is to pair it with a call to `delete`. However, there is no *placement delete* (You can read a discussion on the topic here - http://stackoverflow.com/questions/5857240/why-there-is-no-placement-delete-expression-in-c)

Don't be tempted to call operator delete on your placement `new`'d object, either. The behaviour of delete is to call the destructor on the object, then free the memory associated with the (provided) pointer. In our case, that memory is somewhere in the I/O space! The chances of that ending well are slim-to-none!

If you want to ensure your class's destructor is called you must call it explicitly:

```
int main()
{
  auto const uart { new (reinterpret_cast<void*>(0x40021000)) UART { } };

  uart->read();
  ...

  uart->~UART();  // Explicitly call destructor for cleanup.
}
```

Other limitations of class overlay

This is all looking very neat but there are a couple of important limitations on using classes for hardware memory overlay:

- You can't have virtual functions on the class
- The class can't have any additional attributes, beyond the registers

Think about where our overlay object is being constructed: I/O memory. This is not normal memory. It does not consist of consecutive bytes of data memory. Attempting to store additional attributes will cause them to be read from / written to undefined memory:

- Possibly other hardware registers, where they could randomly change, according to the whims of the underlying hardware device;
- They could corrupt the operation of hardware devices
- Writing to undefined memory locations could (silently)fail or produce junk values
- Putting attributes in the wrong place in the class declaration could 'slew' the hardware overlay layout

Virtual functions require the use of a virtual table pointer as part of the object. This pointer is implicitly added to the object by the compiler; normally at the object's base address. This pointer would (probably) be overlaid on a hardware register (which might be changing!) or in undefined memory. Neither of these is likely to end well for virtual function calls made on the object!

Generic register types

As code designers we tend to eschew specific 'stove-pipe' code in favour of reusable code elements. Up until now we've been coding some very specific examples so it's probably worth looking at some more generic solutions.

Now we'll look at building generic register manipulation classes (or 'register proxy' classes if you prefer). We're really exploring code design rather than coding 'mechanics'. I'm using this to explore some factors like the balance between efficiency, performance and flexibility.

A first-pass design

Our previous designs have focussed on building abstractions of I/O devices – GPIO ports, UARTs, etc. This time we will focus on a class that represents a single hadware register. Here's a first-pass design. This is not a complete interface by any stretch. I'm deliberately ignoring most of the API for clarity. We've discussed interface design in detail earlier.

```
class Register {
public:
  explicit Register(std::uint32_t address);

  Register& operator=(std::uint32_t bit_mask);
  operator uint32_t();

  inline Register& operator|=(std::uint32_t bit_mask);
  inline Register& operator&=(std::uint32_t bit_mask);
  inline Register& operator^=(std::uint32_t bit_mask);

  // etc...

private:
  volatile std::uint32_t* raw_ptr;
};
```

The constructor maps the internal `raw_ptr` onto the supplied address.

```
Register::Register(std::uint32_t address) :
  raw_ptr { reinterpret_cast<std::uint32_t*>(address) }
{
}
```

Operations on the `Register` class are mapped directly onto the hardware

```
Register& Register::operator|=(std::uint32_t bit_mask)
{
  *raw_ptr |= bit_mask;
  return *this;
}
```

Clients can now use `Register` objects as proxies for hardware registers

```
int main()
{
  Register mode   { 0x40020C00 };
  Register output { 0x40020C14 };
```

```
  mode   |= (1 << (15 * 2));
  output |= (1 << 15);
  output &= ~(1 << 15);
  ...
}
```

There's a limitation with this class at the moment: it only handles 32-bit registers. For flexibility we'd like to be able to support 8-bit, 16-bit and 32-bit registers. We could provide multiple classes, for example:

```
class Register_32 {
  // As above.
};


class Register_16 {
  // As above, but for uint16_t.
};


class Register_8 {
  // You get the idea...
};
```

There's a huge amount of code repetition here. The design is crying out for a generic solution.

Second attempt: template-based solution

Let's make our `Register` class a template. But what's the template parameter? Let's start by making it the underlying register type

```
template <typename T>
class Register {
public:
  explicit Register(std::uint32_t address);

  Register& operator=(T bit_mask);
  operator T();

  inline Register& operator|=(T bit_mask);
  inline Register& operator&=(T bit_mask);
  inline Register& operator^=(T bit_mask);
  // etc...

private:
  volatile T* raw_ptr;
};
```

Our client code changes, now

```
int main()
{
  Register<std::uint32_t> mode   { 0x40020C00 };
  Register<std::uint32_t> output { 0x40020C14 };

  mode   |= (1 << (15 * 2));
  output |= (1 << 15);
  output &= ~(1 << 15);
  ...
}
```

This is fine; but it doesn't prevent awkward client code like this:

```
int main()
{
  Register<int> mode   { 0x40020C00 };
  Register<int> output { 0x40020C14 };

  // What happens when you perform bitwise
  // operations on signed numbers?
  //
  mode   |=  (1 << (15 * 2));
  output |=  (1 << 15);
  output &= ~(1 << 15);
  ...
}
```

Third attempt: Using a template trait class

An alternative approach is to encapsulate the type of the pointer and just allow the client to specify the number of bits in the register.

```
#include <cstddef>

template <std::size_t sz>
class Register {
  // We' ll come back to this...
};


int main()
{
  Register<32> mode   { 0x40020C00 };
  Register<16> output { 0x40020C14 };

  mode   |= (1 << (15 * 2));
  output |= (1 << 16);       // Should this be allowed on
                             // on a 16-bit register?
  ...
}
```

This gives us an implementation problem: what's the type of the underlying raw pointer?

```
template <std::size_t sz>
class Register {
public:
  explicit Register(std::uint32_t address);

  Register& operator=(??? bit_mask);
  operator ???();

  inline Register& operator|=(??? bit_mask);
  inline Register& operator&=(??? bit_mask);
  inline Register& operator^=(??? bit_mask);

  // etc...

private:
  volatile ???* raw_ptr;   // What should the type of this pointer be?!
};
```

For an 8-bit register the pointer-type should be (something like) `std::uint8_t`; for a 16-bit register it should be `std::uint16_t`; and so on. How can we deduce the type just from the number of bits?

The solution is to use a template trait class. A trait class acts as a compile-time lookup for type-specific (or, in our case, value-specific) characteristics.

```
template <unsigned int sz>
struct Register_traits { };

template <>
struct Register_traits<8>  { using internal_type = std::uint8_t; };


template <>
struct Register_traits<16> { using internal_type = std::uint16_t };


template <>
struct Register_traits<32> { using internal_type = std::uint32_t; };


template <>
struct Register_traits<64> { using internal_type = std::uint64_t; };


template <std::size_t sz>
class Register {
public:
  // Type alias for convenience
  //
  using reg_type = typename Register_traits<sz>::internal_type;

  explicit Register(std::uint32_t address);

  Register& operator=(reg_type bit_mask);
  operator reg_type();

  inline Register& operator|=(reg_type bit_mask);
  inline Register& operator&=(reg_type bit_mask);
  inline Register& operator^=(reg_type bit_mask);

  // etc...

private:
  volatile reg_type* const raw_ptr;   // <= Register_traits<sz>::internal_type
};
```

When a `Register` template class is instantiated the `sz` template parameter is used to select an appropriate trait class specialisation. The trait class's `internal_type` alias is used to provide the `reg_type` alias for the `Register` class.

```
int main()
{
  Register<32> mode   { 0x40020C00 }; // <= std::uint32_t
  Register<16> output { 0x40020C14 }; // <= std::uint16_t
  Register<17> odd    { 0x40021000 }; // FAIL - No trait for 17-bit
}
```

Thus, a `Register<32>` will have its `reg_type` set as `std::uint32_t`; a `Register<16>` will have its `reg_type` declared as `std::uint16_t`.

If an arbitrary number is selected, for example `Register<17>`, the code will fail to compile as there is no appropriate trait class.

Under the hood

What's the cost of this template complexity in terms of run-time performance? We'll use a variation on code we've used before.

```
#include “Register.h”

namespace STM32F407 {

  enum device {
    GPIO_A, GPIO_B, GPIO_C,
    GPIO_D, GPIO_E, GPIO_F,
    GPIO_G, GPIO_H, GPIO_I
  };


  inline void enable_device(device dev)
  {
    Register<32> rcc_enable { 0x40023830 };

    rcc_enable |= (1 << dev);
  }
}


int main()
{
  Register<32> mode   { 0x40020C00 };
  Register<32> output { 0x40020C14 };

  STM32F407::enable_device(STM32F407::GPIO_D);

  // Set the GPIO mode to output
  // The blue LED is on pin 15
  //
  mode |= (0b01 << (15 * 2));

  while(true) {
    output |= (1 << 15);
    sleep(1000);

    output &= ~(1 << 15);
    sleep(1000);
  }
}
```

Here's the assembler output:

```
; main() {
;
08000d44:   ldr     r1, [pc, #36]       ; r1 = 0x40020C00 <mode>
08000d46:   ldr     r3, [pc, #40]       ; r3 = 0x40020C14 <output>

; STM32F407::enable_device(STM32F407::GPIO_D);
;
08000d48:   ldr     r0, [pc, #40]       ; r0  = 0x40023830 <rcc_enable>
08000d4a:   ldr     r2, [r0, #0]        ; r2  = *r0
08000d4c:   orr.w   r2, r2, #8          ; r2  = r2 | 0x08
08000d50:   str     r2, [r0, #0]        ; *r0 = r2

; mode |= (0b01 << (15 * 2));
;
08000d52:   ldr     r2, [r1, #0]        ; r2  = *r1 <mode>
08000d54:   orr.w   r2, r2, #1073741824 ; r2  = r2 | 0x40000000
08000d58:   str     r2, [r1, #0]        ; *r1 = r2

; while(true) {
; output |= (1 << 15);
loop:
08000d5a:   ldr     r2, [r3, #0]        ; r2  = *r3 <output>
08000d5c:   orr.w   r2, r2, #32768      ; r2  = r2 | 0x8000
08000d60:   str     r2, [r3, #0]        ; *r3 = r2
```

```
; output &= ~(1 << 15);
;
08000d62:   ldr     r2, [r3, #0]        ; r2  = *r3 <output>
08000d64:   bic.w   r2, r2, #32768      ; r2  = r2 & ~0x8000
08000d68:   str     r2, [r3, #0]        ; *r3 = r2
; }
;
08000d6a:   b.n     0x8000d5a           ; goto loop
```

A quick comparison with our earlier implementations reveals almost identical code. This is not a huge surprise as most of the template 'magic' is being done at compile-time, not run-time.

Dealing with read-only and write-only registers

At the moment, our generic `Register` class gives us little extra functionality beyond cleaning up the syntax (although, in its favour, it also doesn't cost any performance at run-time).

Now we're going to extend our design to consider special hardware register types – notably read-only and write-only registers – and see how we can add extra functionality to our `Register` type.

Although there is more than one way to solve this problem we're going to use a simple function-overload selection mechanism called *tag dispatch*.

Read- and Write-only registers

As we discussed in the first chapter a read-only register is a register than cannot be written to; a write-only register is one that may be written to, but if it is read its contents are undefined.

In C it is simple to deal with read-only and read-write registers: we can do so with the `const` qualifier

```
#define READ_WRITE volatile
#define READ_ONLY  volatile const

int main(void)
{
  READ_WRITE uint32_t * const rw_reg = (uint32_t*)0x40020000;
  READ_ONLY  uint32_t * const ro_reg = (uint32_t*)0x40020004;

  uint32_t val_a = *rw_reg;  // OK   - read
  *rw_reg = 0x01;            // OK   - write

  uint32_t val_b = *ro_reg;  // OK   - read
  *ro_reg = 0x01;            // FAIL - write
}
```

But what about write-only registers? There's no way in C of qualifying an object as write-only. Most implementations simply use the same qualification as read-write and rely on the programmer (or code reviews) to find errors

```
#define READ_WRITE volatile
#define WRITE_ONLY volatile
#define READ_ONLY  volatile const

int main(void)
{
  READ_WRITE uint32_t * const rw_reg = (uint32_t*)0x40020000;
  READ_ONLY  uint32_t * const ro_reg = (uint32_t*)0x40020004;
  WRITE_ONLY uint32_t * const wo_reg = (uint32_t*)0x40020008;
  uint32_t val_a = *rw_reg;  // OK   - read
  *rw_reg = 0x01;            // OK   - write

  uint32_t val_b = *ro_reg;  // OK   - read
  *ro_reg = 0x01;            // FAIL - write

  uint32_t val_c = *wo_reg;  // Works, but is not valid
  *wo_reg = 0x01;            // OK - write
}
```

To solve this problem in C++ we're going to use tag dispatch.

Tag dispatch

Tag dispatch is a function overloading and dispatch mechanism. The aim is to categorise classes with particular 'characteristics' and dispatch function overloads based on the particular characteristics on an object.

Class characteristics

First we need to establish what characteristics our classes should have. This is used to define a class hierarchy. The idea is that classes 'accumulate' characteristics through inheritance. These characteristics are defined through a set of 'tag' classes

Let's establish our tag classes:

```
class read_only  { };
class write_only { };
class read_write : public read_only, public write_only { };
```

These are just empty classes. Their only purpose is as a function dispatch selector.

Notice the use of multiple inheritance in our tags. A `read_write` tag is a type-of `read_only` *and* a type-of `write_only`. That is, it supports both reading and writing. More accurately, a `read_write` tag object could be substituted for both a `read_only` object *and* a `write_only` object. We will exploit these features later.

We now need to extend our `Register` class with a new template parameter

```
template <std::size_t sz, typename Register_Ty = read_write>
class Register {

  // As previously...
};
```

The `Register_Ty` template parameter defines the class's read-write characteristics. I've defaulted it to `read_write` (the most general)

We can use this now in `Register` object construction

```
int main()
{
  Register<32>             rw_reg { 0x4002000 }; // Implicitly read-write
  Register<32, read_only>  ro_reg { 0x4002004 }; // Explicitly read-only
  Register<32, write_only> wo_reg { 0x4002008 }; // Explicitly write-only
}
```

Tag dispatch overloads

The next step is to define behaviour (functions) appropriate to each object's characteristics. For our `Register` class this means we want to:

- Enable 'read' functions for read-only `Register` types
- Enable 'write' functions for write-only `Register` types
- Enable 'read' and 'write' for read-write `Register` types

To do this we mark each function based on the `Register` type that can call it. This is done by adding another parameter to the function – the tag class.

In our design this means we hit a problem – for an operator overload (for example, |=) there's no way we can supply an extra parameter to the call. And, in reality, we don't want to have to do that anyway.

The solution is to use nested calls: the operator overload function calls on a (protected) implementation that implements the tag overload

```
template <std::size_t sz, typename Register_Ty = read_write>
class Register {
public:
  using reg_type = typename Register_traits<sz>::internal_type;

  explicit Register(std::uint32_t address) :
    raw_ptr { reinterpret_cast<std::uint32_t*>(address) }
  {
  }

  // Operator overloads.  The public API
  //
  void operator=(reg_type bit_mask)
  {
    write(bit_mask, Register_Ty { });
  }


  operator reg_type() const
  {
    return read(Register_Ty { });
  }


  void operator|=(std::uint32_t bit_mask)
  {
    or_assign(bit_mask, Register_Ty { });
  }


  void operator&=(std::uint32_t bit_mask)
  {
    and_assign(bit_mask, Register_Ty { });
  }

protected:

  // Tag-dispatched implementation
  // functions
  //
  void write(reg_type bit_mask, write_only)
  {
    *raw_ptr = bit_mask;
  }


  reg_type read(read_only) const
  {
    return *raw_ptr;
  }


  void or_assign(std::uint32_t bit_mask, read_write)
  {
    *raw_ptr |= bit_mask;
  }


  void and_assign(std::uint32_t bit_mask, read_write)
  {
    *raw_ptr &= bit_mask;
```

```
  }


private:
  volatile reg_type* raw_ptr;
};
```

Let's look at the public API functions (the operator overloads). These functions simply call the implementation function, passing on any parameters. In addition, they pass an object of type `Register_Ty`. *This* is the tag dispatch. The function that will be called will be the function with the *best-match* signature. If no signature matches, there will be a compiler error.

For example, If the `Register` object is tagged as `read_only`, a call to `operator reg_type()` will call `read(read_only { })`:

```
// For a read_only-tagged Register...
//
operator reg_type() const
{
  return read(Register_Ty { }); // => read(read_only { })
}
```

That is an exact match to the implementation function `read()`.

However, if a call is made to `operator=`

```
// For a read_only-tagged Register...
//
void operator=(reg_type bit_mask)
{
  write(bit_mask, Register_Ty { });  // => write(uint32_t, read_only{ })
}
```

There is no overload for `write()` that matches this signature; and the call fails at compile-time. (Unfortunately, the failure will be flagged inside the `Register` class, not at the point-of-call. Such is the nature of templates)

What about objects marked as `read_write`? For functions like `operator&=` there is a direct match. (Note: since `&=` is a read-modify-write operation the `Register` must support both read and write to work)

For functions like `operator=` this is where tag dispatch gets useful. Since a `read_write` tag class inherits from `read_only` and `write_only` it can be substituted for either of them (note: but not the other way round!).

So a call to `write()` will work, even though the function signature is not an exact match.

```
// For a read_write-tagged Register...
//
void operator=(reg_type bit_mask)
{
  write(bit_mask, Register_Ty { });  // => write(uint32_t, read_write{ })
                                     //    read_write object will be sliced
                                     //    into write_only object.
}
```

The result of this tag dispatch is that we can check for – at compile-time – both read-only and write-only registers, without having to explicitly overload for all combinations of tag value.

```
int main()
{
  Register<32>             rw_reg { 0x4002000 };
  Register<32, read_only>  ro_reg { 0x4002004 };
  Register<32, write_only> wo_reg { 0x4002008 };

  rw_reg         = 0x01;      // OK   - write supported
  uint32_t val_a = rw_reg;    // OK   - read supported

  ro_reg         = 0x01;      // FAIL - write not supported
  uint32_t val_b = rw_reg;    // OK   - read supported

  wo_reg         = 0x01;      // OK   - write supported
  uint32_t val_c = rw_reg;    // FAIL - read not supported
}
```

Under the hood

Given all the extra functions we've written it feels as if there will be a run-time price to pay for all this code.
However - the tag classes are only used for function dispatch. Since the tag-dispatched functions are inlined, and the tag classes are empty and never used, the compiler will optimise them away.

To confirm this here's our code from earlier:

```
int main()
{
  Register<32> rcc_enable { 0x40023830 };
  Register<32> mode       { 0x40020C00 };
  Register<32> output     { 0x40020C14 };

  // Enable GPIO D clock
  //
  rcc_enable |= (1 << 3);

  // Set the GPIO mode to output
  // The blue LED is on pin 15
  //
  mode |= (0b01 << (15 * 2));

  // Flash the LED
  //
  while(true) {
    output |= (1 << 15);
    sleep(1000);

    output &= ~(1 << 15);
    sleep(1000);
  }
}
```

And here's the assembly output:

```
; main() {
;
08000d44:   ldr     r2, [pc, #36]       ; r2  = 0x40023830 <rcc_enable>
08000d46:   ldr     r3, [r2, #0]        ; r3  = *r2
08000d48:   orr.w   r3, r3, #8          ; r3  = r3 | 0x08
08000d4c:   str     r3, [r2, #0]        ; *r2 = r3

; mode |= (0b01 << (15 * 2);
;
08000d4e:   ldr     r2, [pc, #32]       ; r2  = 0x40020C00 <mode>
08000d50:   ldr     r3, [r2, #0]        ; r3  = *r2
08000d52:   orr.w   r3, r3, #1073741824 ; r3  = r3 | 0x40000000
08000d56:   str     r3, [r2, #0]         ‘ *r2 = r3

; while(true) {
; output |= (1 << 15);
;
loop:
08000d58:   ldr     r3, [pc, #24]       ; r3  = 0x40020C14 <output>
08000d5a:   ldr     r2, [r3, #0]        ; r2  = *r3
08000d5c:   orr.w   r2, r2, #32768      ; r2  = r2 | 0x8000
08000d60:   str     r2, [r3, #0]        ; *r3 = r2

; output &= ~(1 << 15);
;
08000d62:   ldr     r2, [r3, #0]        ; r2  = *r3 <output>
08000d64:   bic.w   r2, r2, #32768      ; r2  = r2 & ~0x8000
08000d68:   str     r2, [r3, #0]        ; *r2 = r2

; }
;
08000d6a:   b.n     0x8000d58           ; goto loop

                                        ; Register addresses:
08000d6c:   dcd     1073887280          ; 0x40023830
08000d70:   dcd     1073875968          ; 0x40020C00
08000d74:   dcd     1073875988          ; 0x40020C14
```

You can see from the assembly that we're paying no price at run-time for these checks.

Bit proxies

In this chapter we're going to add some syntactic sugar to our `Register` class, to allow the developer to access individual bits in the register using array-index (`[]`) notation. This will allow us to explore the concept of *proxy objects* with a concrete and practical use case.

Accesssing individual bits

Whilst it's useful to be able to access registers as whole words in many instances it is much more convenient to access individual bits.

One (very common) model for doing this is to conceptually treat a `Register` object as an array of bits. We can now use the index operator to access individual bits

```
int main()
{
  Register<32> mode   { 0x40020C00 };
  Register<32> output { 0x40020C14 };

  mode |= ( 1 << (15 * 2));  // Word-based API

  output[15] = 1;            // Access individual bit
}
```

This is elegant for the client but presents us with some implementation problems. Consider: when you use the index operator on an array it returns a reference to an array element

```
int main()
{
  int arr[10];

  int i = arr[0];  // arr[0] returns a reference to
                   // the array element, NOT a copy
}
```

In the case of the `Register` class we can't return a reference to an individual bit – there is no 'bit' type in C++.

The solution is to create a type that *represents* a bit in the hardware – a *proxy* for a bit. This proxy type should be created as a nested type within the `Register` class.

```
// Tag classes, as previously...

// Traits classes, as previously...

template <std::size_t sz, typename Register_Ty = read_write>
class Register {
private:
  using reg_type = typename Register_traits<sz>::Underlying_Ty;

  // bit_proxy represents one of the
  // individual bits in the hardware
  // register.
  //
  class bit_proxy {
  public:
    // To be defined

  protected:
    bit_proxy(Register<sz, Register_Ty>* reg, unsigned int num) :
      owner   { reg },
      bit_num { num }
    {
    }

  private:
    friend class Register<sz, Register_Ty>;
    Register<sz, Register_Ty>* owner;
    unsigned int  bit_num;
  };

public:

  // API as before...

  bit_proxy operator[](unsigned int index)
  {
    // Check that index is valid...

    return bit_proxy { this, index };
  }
};
```

The `bit_proxy` stores information about the particular bit it represents; but it must also store the identity of the actual `Register` object it is manipulating.

We've made the constructor of the `bit_proxy` protected: we don't want clients to create proxy objects. Constructing a `bit_proxy` is delegated to the index operator. The index operator does no actual bit manipulation – that is all handed off to the proxy.

The client receives a `bit_proxy` object that can be used to indirectly manipulate the original `Register` object.

```
int main()
{
  Register<32> mode   { 0x40020C00 };
  Register<32> output { 0x40020C14 };

  uint32_t value = output[15];     // Read via proxy
  output[15] = 1;                 // Write via proxy
}
```

To support the code above our `bit_proxy` requires two methods:

- Read - Convert to an (unsigned) integer (specifically, whatever `reg_type` is declared as in the owning class).
- Write - Set a particular bit, using an (unsigned) integer value

These functions must indirectly manipulate the owning `Register`. Furthermore:

- A `bit_proxy` for a read-only `Register` must allow reading, but not writing
- A `bit_proxy` for a write-only `Register` must allow writing, but not reading
- A `bit_proxy` for a read-write `Register` must allow reading and writing

To continue our previous pattern we will use tag-dispatch. As the `bit_proxy` is a nested class of the `Register` we can use the `Register_Ty` template parameter (as before).

```
class bit_proxy {
public:
  // Read
  //
  operator Register::reg_type() const
  {
    return read(Register_Ty { });
  }

  // Write
  //
  void operator=(Register::reg_type val)
  {
    write(val, Register_Ty { });
  }

protected:
  bit_proxy(Register<sz, Register_Ty>* reg, unsigned int num) :
    owner   { reg },
    bit_num { num }
  {
  }

  // Tag-dispatched read and write implementations
  //
  Register::reg_type read(read_only) const
  {
    return (*(owner->raw_ptr) & (1 << bit_num))? 1 : 0;
  }

  void write(Register::reg_type val, write_only)
  {
    if(val == 0) *(owner->raw_ptr) &= ~(1 << bit_num);
    else         *(owner->raw_ptr) |=  (1 << bit_num);
  }

private:
  friend class Register<sz, Register_Ty>;
  Register<sz, Register_Ty>* owner;
  unsigned int  bit_num;
};
```

What about the following code?

```
int main()
{
  Register<32> mode   { 0x40020C00 };
  Register<32> output { 0x40020C14 };

  output[15] = 1;               // Call 1
  output[14] = output[15];      // Call 2
}
```

The first call is invoking our assignment operator as specified above. However, the second call is the copy-assignment operator; that is

```
bit_proxy bit_proxy::operator=(const bit_proxy&);
```

We haven't written this function, the compiler supplied it; but it's not going to do what we want. The default behaviour of the copy-assignment operator is to set all the attributes of the left-hand `bit_proxy` to those of the right-hand. This could lead to a number of (disastrous) effects:

- The `bit_proxy` could change its owner to another `Register` object
- The bit number could be changed to represent another bit

In other words, our proxy object would no longer represent its original bit, but some completely other bit!

Remember, a proxy is just a stand-in for a real bit, somewhere in (hardware) memory. What we actually want to do is transfer the *state* represented by the right-hand proxy object onto the state represented by the left-hand proxy object.

Thus there are two operations we need to perform:

- Read the state of the right-hand object
- Change the state of the left-hand object to match.

Luckily, we already have two functions to allow us to do this (although the syntax for calling them in this context is far from pretty)

```
bit_proxy& bit_proxy::operator=(const bit_proxy& rhs)
{
  operator=(rhs.operator Register::reg_type());
  return *this;
}
```

Perhaps a cleaner way to do this is to get the compiler to deduce the calls to make:

```
bit_proxy& bit_proxy::operator=(const bit_proxy& rhs)
{
  *this = static_cast<Register::reg_type>(rhs);
  return *this;
}
```

What about const Registers?

A `const Register` object is effectively a read-only object. As you might expect only read-only operations should compile – even if the `Register` is declared as a read-write type.

(And yes, you can make a write-only `Register const`*; although it makes no sense to do so – you can never read from, or write to, it. Not so useful, then)*

```
int main()
{
  // input is a read-write Register type
  // that cannot be written to(!)
  //
  const Register<32> input { 0x40020C10 };

  input[15] = 1;         // <= Write should fail
```

```
  if(input[15] == 1) {   // <= Read is OK
    ...
  }
}
```

We want reads of a `const Register` object to succeed, but writes should fail.

Our first-pass is to be make a read-only version of the index operator that returns a `const bit_proxy`.

```
const bit_proxy operator[](unsigned int index) const
{
  // Check that index is valid...

  return bit_proxy { const_cast<Register*>(this), index };
}
```

Note the `const_cast<>`. This is required because the `bit_proxy` class stores a pointer to a non-`const` `Register` object internally and that cannot be initialised with a `const`-object (remember, a `const` member function changes the type of the 'this' pointer).

This code works (ugly `const_cast` notwithstanding) but is really not necessary. Since we cannot modify the `Register` object we don't need a proxy object. It is simply enough to return the value of the selected bit. To ensure we don't allow reads of write-only objects we must also called our (tag-dispatched) `read()` function.

```
template <std::size_t sz, typename Register_Ty>
unsigned int Register<sz, Register_Ty>::operator[](unsigned int index) const
{
  return ((read(Register_Ty { }) & (1 << index)) != 0 ? 1 : 0);
}
```

Here's our perennial example – flashing the blue LED – now updated for our bit-access operator

```
int main()
{
  Register<32> rcc_enable { 0x40023830 };
  Register<32> mode       { 0x40020C00 };
  Register<32> output     { 0x40020C14 };

  rcc_enable[3] = 1;

  // Set the GPIO mode to output
  // The blue LED is on pin 15
  //
  mode[30] = 1;
  mode[31] = 0;

  while(true) {
    output[15] = 1;
    sleep(1000);

    output[15] = 0;
    sleep(1000);
  }
}
```

Under the hood (again)

Of course, nothing ever comes for free; there must be a cost to our syntactic sugar. We've created a lot more template code now.

And, as usual, let's examine the output opcodes

```
;main() {
;
08000d44:   ldr     r2, [pc, #44]       ; r2 = 0x40020C00 <mode>
08000d46:   ldr     r3, [pc, #48]       ; r3 = 0x40020C14 <output>

; rcc_enable[3] = 1;
;
08000d48:   ldr     r0, [pc, #48]       ; r0  = 0x40023830 <rcc_enable>
08000d4a:   ldr     r1, [r0, #0]        ; r1  = *r0
08000d4c:   orr.w   r1, r1, #8          ; r1  = r1 | 0x08
08000d50:   str     r1, [r0, #0]        ; *r0 = r1

; mode[30] = 1;
;
08000d52:   ldr     r1, [r2, #0]        ; r1  = *r2 <mode>
08000d54:   orr.w   r1, r1, #1073741824 ; r1  = r1 | 0x40000000
08000d58:   str     r1, [r2, #0]        ; *r2 = r1

; mode[31] = 0;
;
08000d5a:   ldr     r1, [r2, #0]        ; r1  = *r2 <mode>
08000d5c:   bic.w   r1, r1, #2147483648 ; r1  = r1 & ~0x80000000
08000d60:   str     r1, [r2, #0]        ; *r2 = r1

; while(true) {
; output[15] = 1;
;
loop:
08000d62:   ldr     r2, [r3, #0]        ; r2  = *r3 <output>
08000d64:   orr.w   r2, r2, #32768      ; r2  = r2 | 0x8000
08000d68:   str     r2, [r3, #0]        ; *r3 = r2

; output[15] = 0;
;
08000d6a:   ldr     r2, [r3, #0]        ; r2  = *r3 <output>
08000d6c:   bic.w   r2, r2, #32768      ; r2  = r2 & ~0x8000
08000d70:   str     r2, [r3, #0]        ; *r3 = r2

; }
08000d72:   b.n     0x8000d62           ; goto loop

                                        ; Register addresses:
08000d74:   dcd     1073875968          ; 0x40020C00
08000d78:   dcd     1073875988          ; 0x40020C14
08000d7c:   dcd     1073887280          ; 0x40023830
```

This code will be – to all practical intents and purposes – the same performance as the original code we started with in chapter 2

Revisiting read-only and write-only register types

Using tag dispatch is not the only way to solve the read- or write-only Register problem. For completeness let's explore two other alternatives – SFINAE and *constexpr if*.

For these examples I'm going to use a simplified version of our `Register` class. I'm ignoring the bit proxy class and using a reduced API. Once understood, the techniques below can be applied to these elements in the same way we applied tag dispatch.

SFINAE

Substitution Failure Is Not An Error (SFINAE) is a characteristic of template compilation. When performing template deduction on a template function, if the substitution creates an invalid function declaration the compiler will remove that declaration from the set of viable candidates. It does so silently, hence the acronym. Programmers can exploit this mechanism to deliberately generate invalid function signatures for functions they don't want used.

In the case of our `Register` class we want to disable write functions for read-only types, read functions for write-only types; and only enable read-write functions for read-write types.

The standard idiom for doing this is to use the `std::enable_if` type trait.

We'll keep the characteristic classes from the tag dispatch example and make use of the type trait `std::is_base_of<>`. `std::is_base_of<>` will have a `value` of `true` if the first type is a base class of the second type. For example:

```
#include <iostream>
#include <type_traits>

using std::cout;
using std::endl;
using std::boolalpha;
using std::is_base_of;

class A { };

class B : public A { };

class C { };

int main()
{
    cout << boolalpha;
    cout << "A is base of B: " << is_base_of<A, B>::value << endl;
    cout << "B is base of A: " << is_base_of<B, A>::value << endl;
    cout << "C is base of B: " << is_base_of<C, B>::value << endl;
    cout << "C is same as C: " << is_base_of<C, C>::value << endl;
}
```

Note the last example (`std::is_base_of<C, C>`) will return `true`, even though a class can never be its own base.

Here's our modified `Register` class

```
template <std::size_t sz, typename Register_Ty = read_write>
class Register {
public:
  using reg_type = typename Register_traits<sz>::internal_type;


  explicit Register(std::uint32_t address) :
    raw_ptr { reinterpret_cast<reg_type*>(address) }
  {
  }

  template <typename T = Register_Ty,
            typename std::enable_if<is_base_of<write_only, T>::value, bool>::type = true>
  void operator=(reg_type bit_mask)
  {
    *raw_ptr = bit_mask;
  }


  template <typename T = Register_Ty,
            typename std::enable_if<is_base_of<read_only, T>::value, bool>::type = true>
  operator reg_type() const
  {
    return *raw_ptr;
  }


  template <typename T = Register_Ty,
            typename std::enable_if<is_base_of<read_write, T>::value, bool>::type = true>
  void operator|=( reg_type bit_mask)
  {
    *raw_ptr |= bit_mask;
  }


  template <typename T = Register_Ty,
            typename std::enable_if<is_base_of<read_write, T>::value, bool>::type = true>
  void operator&=(reg_type bit_mask)
  {
    *raw_ptr &= bit_mask;
  }

private:
  volatile reg_type* raw_ptr;
};
```

If you're not used to the SFINAE idiom this is pretty obnoxious code. Let's do a quick decomposition of our template functions to see what's going on.

SFINAE only works on template deduction so our member functions must be template functions. The first template parameter (`T`) is the function's template parameter. Normally, this would be deduced from the function call. However, in this case we aren't using `T` as a function parameter, so its type cannot be deduced. Therefore, we are defaulting `T` to the `Register_Ty` type of the `Register` class.

`std::enable_if` requires two template parameters: A Boolean value and a type.

- If the Boolean template parameter is `true`, `std::enable_if` defines an alias, `type`, that is set to the second template parameter.
- If the Boolean value is `false`, no alias is defined.

Thus, the following (pseudo) code

```
typename std::enable_if<true, bool>::type = true
```

Would collapse to

```
typename bool = true
```

This defines an unnamed template parameter (which is fine, we never refer to it again) of type `bool`, with the default value of `true`. This therefore forms a valid function declaration; in other words, the function would be enabled.

However, the code

```
typename std::enable_if<false, bool>::type = true
```

Is invalid, since the alias, `type`, is never declared. Thus it gives an invalid function declaration; and the function is disabled.

As we saw earlier, `std::is_base_of<read_only, T>` returns a Boolean value, depending on whether `T` (which, remember, is always the same as `Register_Ty`) is a derived class of (or actually is) `read_only`.

We could make this code a little more palatable with judicious use of a macro:

```
#define ENABLE_FOR(trait)                                                ¥
 template <typename T = Register_Ty,                                     ¥
 typename std::enable_if<is_base_of<trait, T>::value, bool>::type = true>
```

So our code becomes

```
template <std::size_t sz, typename Register_Ty = read_write>
class Register {
public:
  using reg_type = typename Register_traits<sz>::internal_type;


  explicit Register(std::uint32_t address) :
    raw_ptr { reinterpret_cast<reg_type*>(address) }
  {
  }

  ENABLE_FOR(write_only)
  void operator=(reg_type bit_mask)
  {
    *raw_ptr = bit_mask;
  }

  ENABLE_FOR(read_only)
  operator reg_type() const
  {
    return *raw_ptr;
  }

  ENABLE_FOR(read_write)
  void operator|=(reg_type bit_mask)
  {
    *raw_ptr |= bit_mask;
  }
```

```
  ENABLE_FOR(read_write)
  void operator&=(reg_type bit_mask)
  {
    *raw_ptr &= bit_mask;
  }

private:
  volatile reg_type* raw_ptr;
};
```

Our client code works as expected.

```
int main()
{
  Register<32, read_only>  ro_reg { 0x40020100 };
  Register<32, write_only> wo_reg { 0x40020104 };
  Register<32, read_write> rw_reg { 0x40020108 };

  std::uint32_t val  { };
  std::uint32_t data { 0x01 };

  val = ro_reg;       // OK
  ro_reg = data;      // FAIL
  ro_reg |= data;     // FAIL

  val = wo_reg;       // FAIL
  wo_reg = data;      // OK
  wo_reg |= data;     // FAIL

  val = rw_reg;       // OK
  rw_reg = data;      // OK
  rw_reg |= data;     // OK
}
```

Error messages are, by template standards, not too cryptic. For example:

```
error: no viable overloaded '='

  ro_reg = data;   // FAIL
  ~~~~~~ ^ ~~~~
note: candidate function (the implicit copy assignment operator)
not viable: no known conversion from 'std::uint32_t'
(aka 'unsigned int') to 'const Register<32, read_only>'
for 1st argument

class Register
      ^
note: candidate function (the implicit move assignment operator)
not viable: no known conversion from 'std::uint32_t'
(aka 'unsigned int') to 'Register<32, read_only>'
for 1st argument

class Register
      ^
note: candidate template ignored: disabled by 'enable_if'
[with T = read_only]

  ENABLE_FOR(write_only)
  ^
note: expanded from macro 'ENABLE_FOR'
  typename std::enable_if<is_base_of<trait, T>::value,
  bool>::type = true>
```

Notice that, because the operator= overload has been disabled (by SFINAE) the compiler is attempting to use the built-in assignment operator (which fails because there is no way to convert a Register<>::reg_type into a Register<> – the constructor is marked explicit)

Constexpr if

Constexpr if is a new feature in C++17. It provides compile-time conditionals. Given a compile-time Boolean expression a constexpr if statement can be used to enable or disable blocks of code.

As an example, constexpr if can be used as a pre-processor replacement for conditional compilation - #if/ #elif/ #else/ #endif.

Below, we have a function UART_enable() that may have different implementations (or even partially-different implementations) for different target systems. We use an enum class to define the set of systems, and a global constexpr object that defines the current target.

If the constexpr if statement is true the code block is enabled; if false, the code block is discarded (that is, not compiled)

```
enum class uController { STM32F103, STM32F407 };

constexpr uController target { uController::STM32F407 };


void UART_enable()
{
  if constexpr(target == uController::STM32F103) {

    // STM32F103 implementation...
  }

  if constexpr(target == uController::STM32F407) {

    // STM32F407 implementation...
  }

  // Common code...
}


int main()
{
  UART_enable();  // Implementation for STM32F407
 ...
}
```

For our Register class we don't want to enable code but disable it, based on the type of register we have. Once again, we can exploit the std::is_base_of<> trait.

Here's a first-pass at our class. I'm only showing one method for clarity; the others all follow the same pattern.

```
template <std::size_t sz, typename Register_Ty = read_write>
class Register {
public:
  using reg_type = typename Register_traits<sz>::internal_type;


  explicit Register(std::uint32_t address);
```

```
  void operator=(reg_type bit_mask)
  {
    if constexpr(!is_base_of<read_only, Register_Ty>()) {

      // Compile-time error for read-only objects...
    }

    *raw_ptr = bit_mask;
  }

  operator reg_type() const;
  void operator|=(reg_type bit_mask);
  void operator&=(reg_type bit_mask);

private:
  volatile reg_type* raw_ptr;
};
```

We want to stop compilation if `operator=` is called on a read-only object. The obvious way to do that is with `static_assert`

```
void operator=(reg_type bit_mask)
{
  if constexpr(!std::is_base_of<read_only, Register_Ty>()) {

    static_assert(false, "Writing to read-only object");
  }

  *raw_ptr = bit_mask;
}
```

This looks good – it will give us a nice, human-understandable error message at compile-time if we attempt to write to a read-only `Register` instance.

Sadly, it won't compile.

A `static_assert()` expression can be placed anywhere in code (not just inside a block). Our `operator=` is a member function of a template class. The `static_assert`, however, is completely independent of the template parameter, therefore the compiler is free to evaluate the `static_assert` before even considering the template code.

We could try (in desperation) the following

```
void operator=(reg_type bit_mask)
{
  if constexpr(!std::is_base_of<read_only, Register_Ty>()) {

    static_assert(std::false_type(), "Writing to read-only object");
  }

  *raw_ptr = bit_mask;
}
```

`std::false_type` is, again, an invariant not based on our template parameter; so it fails.

The solution is to 'trick' the compiler into instantiating the template before it evaluates the constexpr if statement. To do this we build a type-dependent variant of `std::false_type`:

```
template<typename T> struct dependent_false : std::false_type { };
```

It doesn't matter what the template parameter is, we're never using it.

This new 'false type' can be put into our `static_assert()`:

```
void operator=(reg_type bit_mask)
{
  if constexpr(!std::is_base_of<read_only, Register_Ty>()) {

    static_assert(dependent_false<Register_Ty>(),
    " Writing to read-only object" );
  }

  *raw_ptr = bit_mask;
}
```

And now our code compiles.

We could, if we desire, sprinkle some 'macro sugar' onto our code; for example

```
#define ASSERT_IF_NOT(trait, error_string)                    ¥
if constexpr(!is_base_of<trait, Register_Ty>()) {             ¥
  static_assert(dependent_false<Register_Ty>(), error_string); ¥
}
```

Our (sugar-coated) `Register` class becomes

```
template <std::size_t sz, typename Register_Ty = read_write>
class Register {
public:
  using reg_type = typename Register_traits<sz>::internal_type;


  explicit Register(std::uint32_t address) :
    raw_ptr { reinterpret_cast<reg_type*>(address) }
  {
  }

  void operator=(reg_type bit_mask)
  {
    ASSERT_IF_NOT(write_only, "Writing to read-only object");

    *raw_ptr = bit_mask;
  }

  operator reg_type() const
  {
    ASSERT_IF_NOT(read_only, "Reading from write-only object");

    return *raw_ptr;
  }

  void operator|=(reg_type bit_mask)
  {
    ASSERT_IF_NOT(read_write, "Object must be read-write");

    *raw_ptr |= bit_mask;
  }

  void operator&=(reg_type bit_mask)
  {
    ASSERT_IF_NOT(read_write, "Object must be read-write");

    *raw_ptr &= bit_mask;
  }

private:
  volatile reg_type* raw_ptr;
};
```

Error messages in client code are now explicit

```
int main()
{
  Register<32, read_only>  ro_reg { 0x40020100 };
  Register<32, write_only> wo_reg { 0x40020104 };
  Register<32, read_write> rw_reg { 0x40020108 };

  std::uint32_t val  { };
  std::uint32_t data { 0x01 };

  val = ro_reg;       // OK
  ro_reg = data;      // FAIL
```

```
}
```

```
error: static_assert failed "Writing to read-only object"

  ASSERT_IF_NOT(write_only, "Writing to read-only object");
  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

note: expanded from macro 'ASSERT_IF_NOT'
  static_assert(dependent_false<Register_Ty>(), error_string); ¥
  ^             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

note: in instantiation of member function
'Register<32, read_only>::operator=' requested here

ro_reg = data;   // FAIL
```

Performance considerations

In this chapter we've focussed on code structure and error output, rather than run-time performance.

Both the SFINAE and constexpr if variants shown are compile-time checks, therefore they have no impact on run-time performance. In fact, their performance will be identical to the tag dispatch variant shown earlier – which is (more or less) identical to 'raw' pointer manipulation.

Summary

C++, as a superset of C, provides developers with the same hardware manipulation idioms. The object-oriented facilities of C++ give programmers extra facilities that cannot be (conveniently) replicated in C

Wrapping access to a hardware device in a class gives us a number of benefits:

- The constructor can be used to initialise the device
- The member functions decouple the actual access from the application
- The destructor can be used to place the device back in a "safe" state
- We can easier separate interface (API) from implementation
- We can support multiple devices

Notice there are at least four design aspects we have to consider for any new class:

- The client – behavioural – API
- The construction /initialisation and destruction interface
- The copy and move policy
- Error condition reporting

It's very easy to get absorbed with the functional API of a class and forget that the other three interfaces can also have far-reaching consequences for your system.

For any class-based approach there are three basic implementations:

- Nested pointers are simple but not especially memory-efficient
- Pointer offsets are efficient but limited to same-size registers and (effectively) contiguous register alignment
- Structure overlay allows a more flexible arrangement than pointer offsets.

There is a fear that more complex C++ constructs – like templates – should be avoided in embedded code for fear of significant memory and performance issues. Used carefully and judiciously, though, these constructs can produce very flexible abstractions at very low (almost zero) additional cost.

Contact Us

Feabhas Limited

15-17 Lotmead Business Park,
Wanborough,
Swindon,
SN4 0UY
UK

Email: info@feabhas.com
Phone: +44 (0) 1793 792909
Website: www.feabhas.com