EE 14: Writing registers with C

Steven Bell

January 2025

Tufts UNIVERSITY

By the end of class today, you should be able to:

- Use C code to read / write a memory-mapped hardware register
- Explain what the volatile keyword means

Problem

You have a processor, and you want to connect a widget to it. How?

Problem

You have a processor, and you want to connect a widget to it. How?

1) Add special instructions to the processor

Then you need a custom compiler for every processor?!

This is possible (e.g., Tensilica Xtensa core), but hard to scale

Problem

You have a processor, and you want to connect a widget to it. How?

1) Add special instructions to the processor

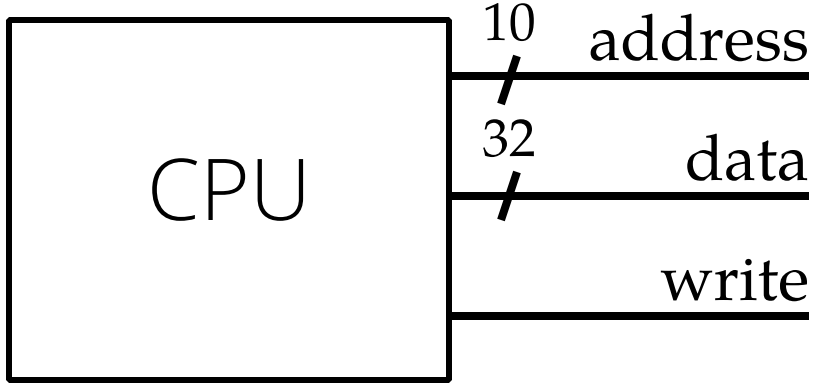
Then you need a custom compiler for every processor?!

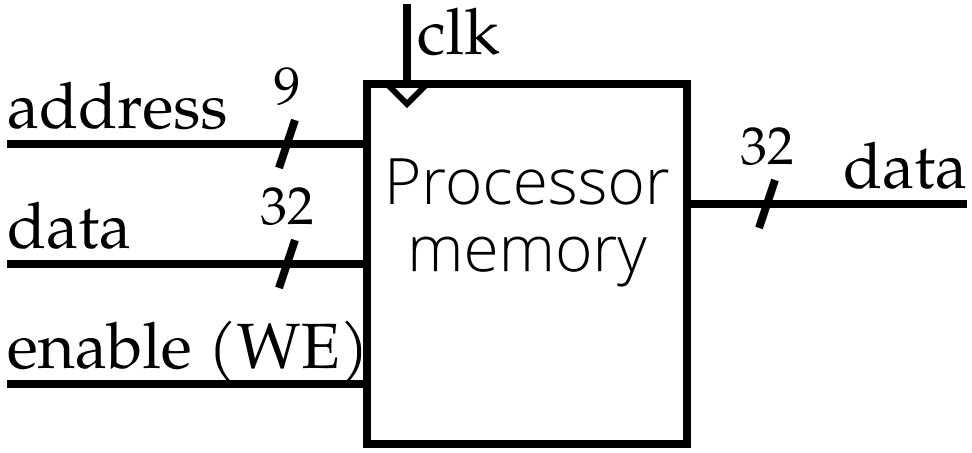
This is possible (e.g., Tensilica Xtensa core), but hard to scale

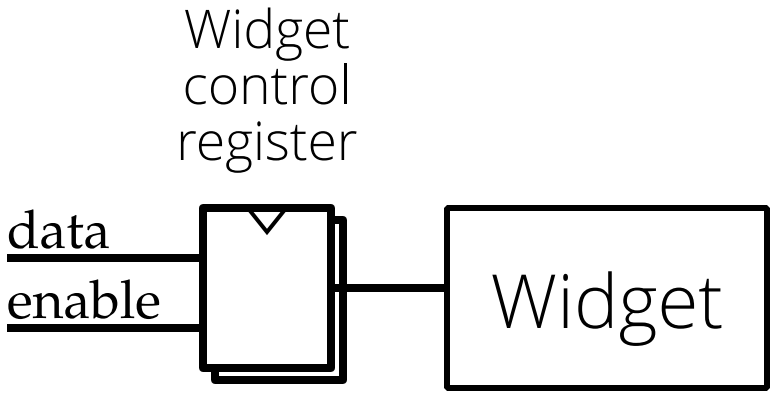
2) Make the widget act like memory

Memory-mapped I/O

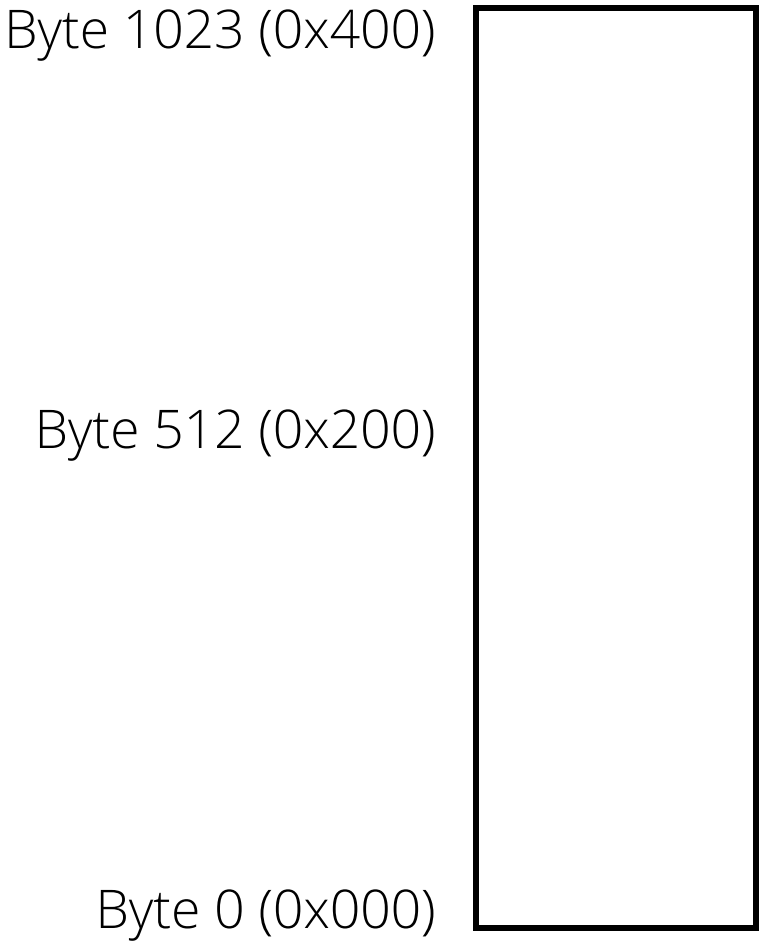
(write-only in this diagram, but extends to read/write)







Drawing a memory map



Byte-addressable (every byte has its own address)

10-bit address space = 1024 bytes

2^9 = 512 bytes of RAM

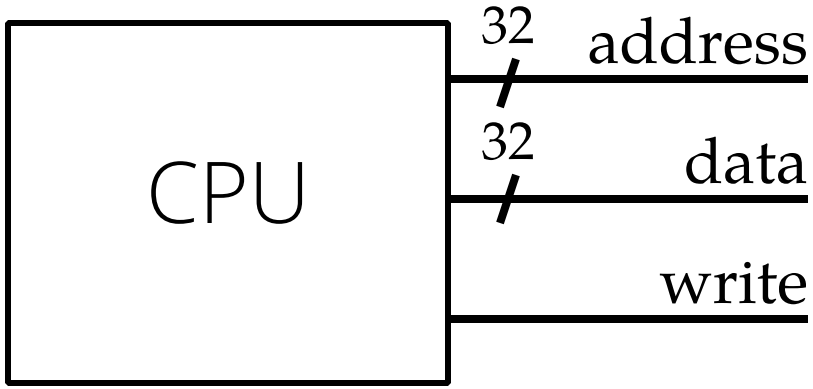
Widget register is at

Why memory-mapping is awesome

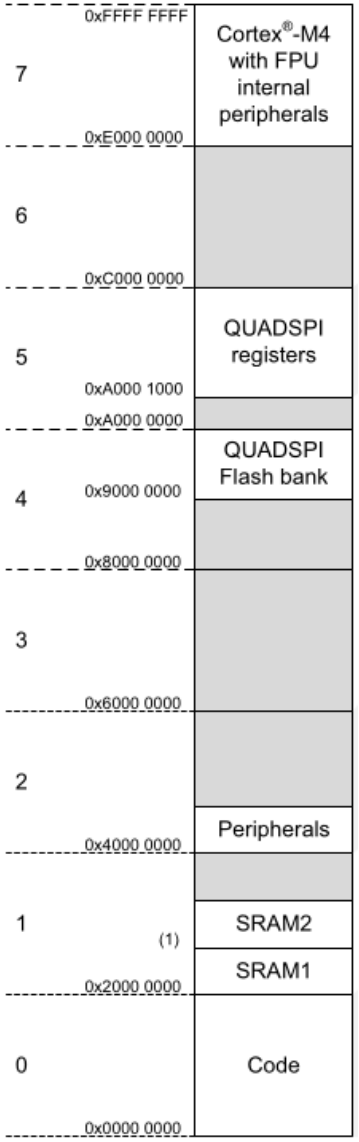
If I have a memory-mapped peripheral,
I can interact with it just by using load/store instructions
just as if it were memory!

Everything is memory-mapped

Even with multiple memory segments and lots of peripherals,
we can map it all into one address space!



Actual memory map for STM32L4xxx



Bus	Boundary address	Size (bytes)	Peripheral	Peripheral register map
AHB2	0x5006 0800 - 0x5006 0BFF	1 KB	RNG	*Section 24.7.4: RNG register map*
	0x5006 0400 - 0x5006 07FF	1 KB	Reserved	-
	0x5006 0000 - 0x5006 03FF	1 KB	AES(1)	*Section 25.7.18: AES register map*
	0x5004 0400 - 0x5005 FFFF	127 KB	Reserved	-
	0x5004 0000 - 0x5004 03FF	1 KB	ADC	*Section 16.9: ADC register map on page 488*
	0x5000 0000 - 0x5003 FFFF	16 KB	Reserved	-
	0x4800 2000 - 0x4FFF FFFF	~127 MB	Reserved	-
	0x4800 1C00 - 0x4800 1FFF	1 KB	GPIOH	*Section 8.5.12: GPIO register map*
	0x4800 1400 - 0x4800 1BFF	2 KB	Reserved	-
	0x4800 1000 - 0x4800 13FF	1 KB	GPIOE(2) (3)	*Section 8.5.12: GPIO register map*
	0x4800 0C00 - 0x4800 0FFF	1 KB	GPIOD(2)	*Section 8.5.12: GPIO register map*
	0x4800 0800 - 0x4800 0BFF	1 KB	GPIOC	*Section 8.5.12: GPIO register map*
	0x4800 0400 - 0x4800 07FF	1 KB	GPIOB	*Section 8.5.12: GPIO register map*
	0x4800 0000 - 0x4800 03FF	1 KB	GPIOA	*Section 8.5.12: GPIO register map*
	0x4002 4400 - 0x47FF FFFF	~127 MB	Reserved	-
AHB1	0x4002 4000 - 0x4002 43FF	1 KB	TSC	*Section 23.6.11: TSC register map*
	0x4002 3400 - 0x4002 3FFF	1 KB	Reserved	-
	0x4002 3000 - 0x4002 33FF	1 KB	CRC	*Section 14.4.6: CRC register map*
	0x4002 2400 - 0x4002 2FFF	3 KB	Reserved	-
	0x4002 2000 - 0x4002 23FF	1 KB	FLASH registers	*Section 3.7.13: FLASH register map*
	0x4002 1400 - 0x4002 1FFF	3 KB	Reserved	-

Ok, how do we do this in C?

A pointer is just the address of specific memory location

So we can write:

```
unsigned int* p = (unsigned int*)0x40003000;
*p = 0x90; // Set some bits

unsigned int y = *p; // Read the peripheral register
```

Ok, how do we do this in C?

A pointer is just the address of specific memory location

> Side note: "big" CPUs running an OS use "virtual memory", so the memory addresses your code sees are not hardware addresses.
>
> But on a microcontroller, what you see is the physical address!

So we can write:

```
unsigned int* p = (unsigned int*)0x40003000;
*p = 0x90; // Set some bits

unsigned int y = *p; // Read the peripheral register
```

Let's look at a GPIO register

Lots of functions are packed together in each register!

8.5.1 GPIO port mode register (GPIOx_MODER) (x = A to E, H)

Address offset:0x00

Reset value: 0xABFF FFFF (for port A)

Reset value: 0xFFFF FEBF (for port B)

Reset value: 0xFFFF FFFF for ports C..E

Reset value: 0x0000 000F (for port H)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODE15[1:0]		MODE14[1:0]		MODE13[1:0]		MODE12[1:0]		MODE11[1:0]		MODE10[1:0]		MODE9[1:0]		MODE8[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
MODE7[1:0]		MODE6[1:0]		MODE5[1:0]		MODE4[1:0]		MODE3[1:0]		MODE2[1:0]		MODE1[1:0]		MODE0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0 **MODE[15:0][1:0]:** Port x configuration I/O pin y (y = 15 to 0)
These bits are written by software to configure the I/O mode.
00: Input mode
01: General purpose output mode
10: Alternate function mode
11: Analog mode (reset state)

Manipulating bits (bit masking)

```
unsigned int* p = (unsigned int*)0x40003000;
```

We could write the whole register:

```
*p = 0x90; // Set all the bits, either 1 or 0
```

We can set bits to 1 using bitwise OR with 1

```
*p = *p | 0x90; // 10010000
```

We can clear bits to 0 using bitwise AND with 0

```
*p = *p & 0x90; // 10010000
```

Easier ways to write this stuff

If I want to set bit 13 of a register, I could write

```
*p = *p | 0b0010000000000000; // how many zeros was that?
```

Or I could use hex:

```
*p = *p | 0x2000;
```

But it's easier to let the compiler do the math:

```
*p = *p | (1 << 13); // << is left shift
```

And C has a nice shorthand:

```
*p |= (1 << 13); // also works for &=, +=, -=
```

There's just one problem...

Suppose I have the following code:

```
*p = 0x01; // Turn LED on
*p = 0x00; // Turn LED off
```

Or this:

```
while(*p == 0) {} // Wait for switch to be pushed
```

When the compiler optimizes this code, it will remove it!

The volatile keyword

The solution is to declare the pointer volatile:

```
volatile unsigned int* p = (unsigned int*)0x40030200;
*p = 0x01; // Turn LED on
*p = 0x00; // Turn LED off
```

`volatile` tells the compiler
1) Don't optimize out writes, because they "do something"
2) Don't optimize out reads, because the value can change outside the code