ECE 271 – Microcomputer Architecture and Applications Lecture 2

Vince Weaver

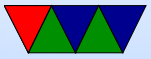
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

20 January 2022

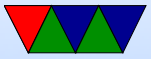
Announcements

- Apologize for course webserver being unavailable to outside world (still accessible from UMaine network). Hopefully this gets fixed soon
- If you need files and cannot use the UMaine VPN let me know, I will make sure you can access the files you need.



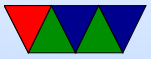
Preparing for Lab #1

- Be sure to bring a laptop
- Be sure to do the Pre-Lab in advance (will be posted to website)
- We will hand out the boards in lab



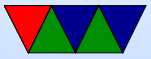
Bare-Metal vs Operating System

- Modern computers often run an Operating System (such as Linux or Windows) that abstracts away low-level hardware accesses
- This makes programming easier, among other things
- In embedded work on small systems like the STM boards it's not really practical to have an operating system, so we will program directly to the hardware, "bare metal"



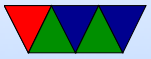
General Purpose Input/Output (GPIO)

- Configurable pins that can be used to send data into and out of the embedded board
- Read textbook Chapter 14



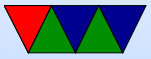
GPIOs on STM32L4

- GPIO pins can be configured 4 ways
  - Digital input (0 or 1 based on some threshold)
    What threshold?
  - Digital output (0 or Vdd)
    How much current can you drive? What happens if you short it?
  - Analog functions
    DAC (digital-analog-conversion), ADC
  - Alternate Functions:



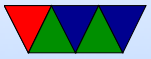
PWM, LCD driver, timer, USART, SPI, I2C, USB

- Why alternate functions? "only" 100 pins on chip, can have more functions than that so you can map them to the pins (but it does have limitations, might not be possible to have all combinations of supported I/O at once)



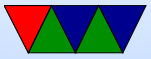
GPIO Inputs

- Read a value, 0 or 1 (ground or 3.3V)
- When GPIO configured input, appears as high-impedance (floating) to outside
- When left floating, what value is there if you read it with CPU?
- Parameters to set
  - Pull-up/Pull-down (see next slide)
  - Schmitt-trigger – to give cleaner signal on input
    See plot at Textbook Figure 14-4

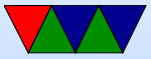


Pull-up/Pull-down resistors

- Will pull to high or low value.
- When external voltage applied, it can overpower these.
- With a pull-up/down resistor, will have current flow. This wastes power, so often they have a relatively high value.
- These are "weak" pull-ups that can be overpowered
  Don't want to use them in high-speed cases. Why? RC time constant?

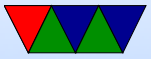


- Can use “strong” external resistors if really need pullups.



GPIO Outputs

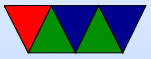
- Output 0 or 1 (ground or 3.3V)
- Various parameters you can set
  - Push-pull vs open-drain.
    Draw diagram of Open Drain (wired-and/wired-or)
  - Pull-up or Pull Down
  - Slew (how fast the clock signal rises). In an ideal world would want as fast as possible, but low-slew can be bad. Fourier series, lots of harmonics?



Memory-mapped I/O

- Some CPUs have specific I/O instructions that have an I/O port range and a special instruction to output to it
- Most CPUs these days do I/O via memory-mapped I/O
- A range of address space in the memory range is interpreted sort of as registers, and these are used to program the I/O

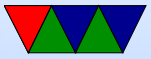
  Use standard load/store instructions to access it



Memory-mapped I/O on Cortex-M

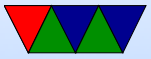
- The ARM Cortex-M4 board has mmio registers
- Peripheral MMIO Starts at 0x40000000
- Mention physical vs virtual memory

0xe000.0000	(3.75GB)	ARM internal peripherals
0x4000.0000	(1GB)	peripherals
0x2000.0000	(512MB) 96k	primary SRAM
0x1000.0000	(256MB) 32k	part of SRAM
0x0800.0000	(128MB) 1MB	Flash (stack ptr and vectors at bottom)



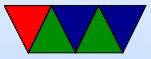
Memory-mapped I/O in C

- If you want to read or write to address `0x4800.0414` can you do this?
  Could you do this in Java or Python? why not
  Could you do this in assembly language?
  Great power comes great responsibility. This is fun part of computing.
- C has pointers. Powerful. Mysterious. Easy to get wrong.
- ```
  uint32_t value;
  uint32_t *pointer;
  pointer= &value;    // pointer now has address of value
  ```


  ```

```
*pointer=42;          // dereference this address and store to it
```

- Can we just set `pointer=0x48000414; *pointer=whatev`
- Yes... but modern C compilers make this difficult. If C compiler sees you write to memory but not use the result it might optimize it away. *p=1; *p=2; *p=3; Might see this and say that's pointless, let's just set p to 3, gives right final result. But if points to I/O space it might be important that all 3 get set.
- Big hack is the `volatile` keyword, tells C compiler that what is being pointed to can chance so you should always read or write from it even if it looks like it doesn't



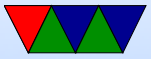
matter.

```
#define GPIO_BASE   0x48000000
volatile uint32_t *gpio;
gpio = (uint32_t *)GPIO_BASE;

gpio[2]=0xdeadbeef;
```

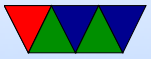
- Textbook recommends making a volatile struct and using that. Not sure I like that. Forcing "packed structs" depends on C version and can not work in some case due to padding. Also read-modify-write issues with 16-bit subfields?
- I like inline assembly

```
static inline void mmio_write(uint32_t address, uint32_t data) {
        uint32_t *ptr = (uint32_t *)address;
        asm volatile("str␣%[data],␣[%[address]]" :
                : [address]"r"(ptr), [data]"r"(data));
```



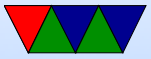
```
}

static inline uint32_t mmio_read(uint32_t address) {
        uint32_t *ptr = (uint32_t *)address;
        uint32_t data;
        asm volatile("ldr␣%[data],␣[%[address]]" :
                [data]"=r"(data) : [address]"r"(ptr));
        return data;
}
```



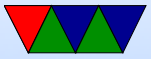
Setting bits in STM32L4 Registers

- To enable GPIO we will have to write some registers
- We will need to set or clear some bits in these registers
- What registers do we set?
  - See documentation (1800 pages!)
  - Can you always trust documentation?



Bit-manipulation in C

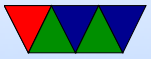
- It's useful if you can set/clear individual bits in an integer without affecting the existing bits
- C has ways of modifying the bits in an integer
- Note that LOGICAL bit manipulation is different from BITWISE big manipulation, and it can be easy to get these confused
- For example && is logical AND (like in a if comparison), but & is bitwise AND (on all bits of an integer)



Setting a bit: OR

- Remember your digital logic
- You can force a bit on by ORing with 1, ORing with 0 leaves it alone

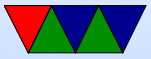
```
/* Setting bit 2 of gpio[2] */
x=gpio[2];
x=x|(1<<2);
gpio[2]=x;
```



Clearing a bit: AND with INVERSE

- You can force a bit off by ANDing with 0, ANDing with 1 leaves it alone

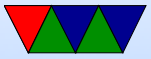
```
/* Clearing bit 2 of gpio[2] */
x=gpio[2];
x=x&(~(1<<2));
gpio[2]=x;
```



Toggling a bit: XOR

- You can toggle a bit by XORing with 1, XORing with 0 leaves it alone

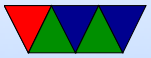
```
/* Flip bit 2 of gpio[2] */
x=gpio[2];
x=x^(1<<2);
gpio[2]=x;
```



Setting multiple bits

- To do this you need to clear first with a MASK then OR in.
- If manually making mask, be sure the 1s go the whole way to bit 31

```
/* Set bits 4 and 5 to 01 */
/* First clear bits 4 and 5 to zero */
x=gpio[2];
mask=~(0x3<<4);  // mask = 0xffffffcf
x=x&mask;
/* set to the value we want */
x=x|(0x1<<4);
gpio[2]=x;
```



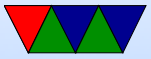
Setting register values

- Be careful if read-modify write, if value can change after you read but before you write (interrupts or for other reason) can lead to unexpected results
- In C:

```
a=memory[x];
a=a|0x1;
memory[x]=a;
```

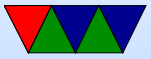
or

```
memory[x]|=0x1;
```



Use of magic constants

- Be sure to comment your code
- Having "magic constants" such as 0x3 or such can be hard to follow, often it's considered good practice to define these with names such as `GPIOA_ENABLE1` or such so it's clear what exactly you're trying to do with your bit manipulations



Converting Binary to Hex

- Some C compilers let you use binary constants like `0b00010` but this is nonstandard
- Often it's easiest to convert these to hexadecimal (base-16)
- The easiest way to do this is group your binary values to nibbles (4-bit chunks) and know those 4 bits convert directly to a hex digit
- `0000 1000 1010 0101 = 0x08A5`