Notes for ECE 36200 - Microprocessor Systems And Interfacing

Zeke Ulrich

September 3, 2025

Contents

Course Description

An introduction to basic computer organization, microprocessor instruction sets, assembly language programming, and microcontroller peripherals.

GPIO

General Purpose Input/Output (GPIO)

Input/output (I/O) is the interface between our digital microcontroller (MCU) and the rest of the world. Each MCU has pins which its software uses to read input and write output.

Some pins are general purpose I/O, but some are used to power the MCU (VDD, GND) and some pins are special (oscillator, clock pins), can to analog to digital I/O or vice versa, or more complex digital I/O (SPI, I2C, UART).

Digital input reads the pin to check if an external voltage is applied. Digital output drives the voltage on the pin. More advanced circuity performs digital-to-analog conversion (DAC) or analog-to-digital (ADC).

The layout of every GPIO pin port looks like Figure **??**

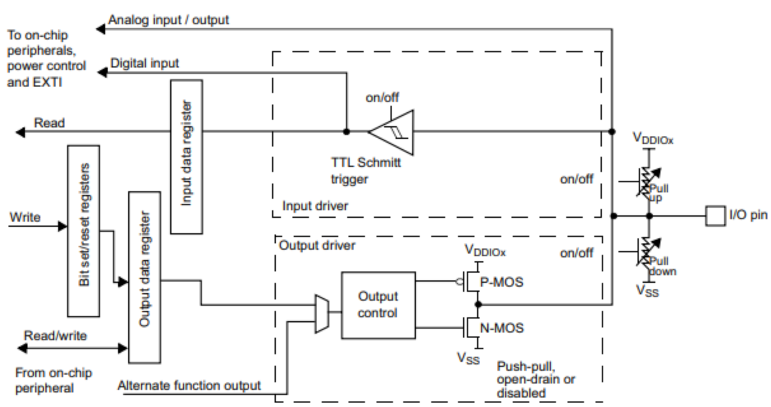

Figure 1: GPIO Layout

Each MCU has multiple GPIO ports A, B, etc. Each port has multiple GPIO bits (typically 8).

Output

The way output works is that an output bit is written into the Output Data Register (ODR). Then the output driver reads the ODR and drives the pin either high or low depending on the value of the ODR bit. The voltage then appears on the I/O pin.

The GPIO can be configured as either read or write, output mode of push-pull or open drain, output speed, and more.

In push-pull mode, we have need of a buffer to increase the output voltage of the MCU so that when the output of the software is 1 the output of the GPIO pin is VCC. This is accomplished with two NOT gates. If the software write a 1, then the controller outputs a 0 to a

NOT gate, which pulls the output pin to 1. Basically, in push-pull mode the MCU actively drives the pin low for 0 or high for 1.

In open drain mode, the MCU drives the output to low, but floats in 1. That is the GPIO pin can be 0 volts, but it cannot be VCC. Basically, it can actively drive a pin low for 0, but leaves the pin floating for 1. Open drain mode is useful when there are multiple outputs. Two output pins can be tied together, which isn't possible with push-pull outputs because one could be high and one low, causing a short circuit. Thus, all tied pins must be open drain to avoid short circuits. Any one of them can drive the shared output low, while a pull-up resistor passively pulls the wire up to high when no MCU is driving it to low.

This is how I23 works. More details to follow in the unit of I2C.

Another configurable is output speed, the speed of voltage rising and falling. A faster GPIO is good for fast communication, but higher speed increases electromagnetic interference and power consumption.

A related concept is slew rate, defined as

$$\max\left(\frac{\Delta V}{\Delta t}\right) \tag{1}$$

Input

With input, an external circuit applies a voltage to the I/O pin. The input driver converts the voltage to either 1 or 0. The input is then sampled into the Input Data Register (IDR) every clock cycle.

In real life, the input voltage is noisy and messy. We add a Schmitt trigger to smooth it, reduce noise, and increase the slew rate to make it suitable for our digital circuit. Recall that a Schmitt trigger is just a buffer that is immune to oscillating issues because it has a low and high threshold. When the input signal crosses the high threshold, the output of the Schmitt trigger is high. It stays high until the input signal crosses the low threshold, at which point the output of the Schmitt trigger goes low.

Electricity in the real world is unfortunately messy. Consider the circuit in Figure ??.

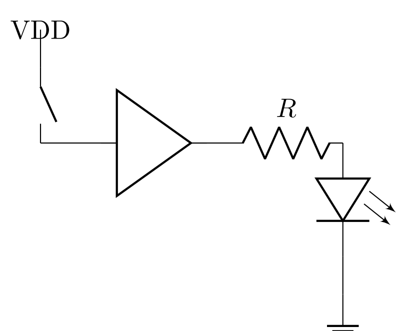

Figure 2: Faulty LED Control Circuit

You would expect that the circuit turns off when the switch is open.

However, the input to the buffer would be floating, so the output of the buffer is unpredictable.

We can remedy this by adding a pull-down resistor, as in Figure **??**.

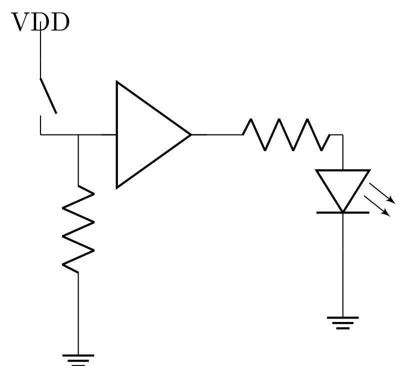

Figure 3: LED Control Circuit

Pins face a similar issue. A floating pin's voltage is unknown. We need to add a pull-up or pull-down resistor so that when the voltage isn't actively being driven low or high then the pin is at a known voltage. The resistors have to be weaker than however the pins are being driven, or the pin would always be high/low, but if it's too weak then the pin will be slow and unresponsive.

Many MCUs have on-chip PU/PD resistors, which are software configurable. Off-chip PU/PD can also be on the PCB instead, in which case it is not configurable.

We have two options or using software to set up a pin.

- Port-mapped I/O, which uses special instructions in the processor where every device is assigned a unique port number (used in x86).
- Memory-mapped I/O, which has special addresses which can be read from or written to in order to perform I/O.

This is true for all on-chip modules, not just GPIO.

Memory-mapped I/O

If we have a 32-bit CPU, each address is 1 byte and there are up to 4 GBs of addressable memory in a 32 bit system. MCUs have limited read/write memory (less than a few MB). Memory is *byte-addressable*. Each byte has a unique address. Addresses go from 0x00000000 to OxFFFFFFFF. The bottom sections have on-chip flash memory, for code and data. The next sections have SRAM, on-chip RAM, for the heap, stack, and some code. At the top is system memory dedicated to external devices like NVIC, system timer, SCB, other vendor-specific memory. Also in high memory (but not the highest) is memory for external devices like SD cards. In the middle is external RAM, off-chip memory for data. See Figure **??**.

Most modern computers are 64-bit, but MCUs are still mainly 32-bit.

When the CPU needs to talk to RAM (read/write variables) it's connected via the address bus and data bus. It reads from ROM (read-only memory). When the CPU wants to write to the RAM, it gives an address to the address bus and data to the data bus. The RAM looks at the address bus and if it sees an address that belongs in RAM space, the RAM will take the data from the data bus (if's it's a write signal) and write it at the specified address. When the CPU wants to read it puts an address on the address bus and signals read. The RAM sees the read signal and puts data at the specified address on the data bus.

While RAM is read and write, ROM is read-only. The reason for this separation is that ROM is cheaper. If you turn off your computer, anything in RAM is lost. However, anything is ROM is non-volatile and anything flashed there will remain even if power is lost.

GPIO modules are on the same address and data buses as the ROM and RAM. Each periperphal has a set of control registers, including direction and data registers. Each register has a unique memory address. If you want to write something to the data register, get its address by looking at the data sheet, and write there. Reading is similar. By writing a one or zero into the direction register, you can configure the GPIO as either input or output.

To turn on an LED, for instance, we need to know what port and pin it's connected to. We then consult the data sheet to get the addresses of the registers for that pin and port, and then we just need to write some value to the registers.

Although the GPIO pins are not really memory, the CPU treats them like memory. That's why this method is called memory-mapped I/O.

Memory is accessed at a minimum granularity of one byte, but typically 4.

```
char* controlRegisterPtr = 0x50000000;
*controlRegisterPtr = 1;
```

Interrupts

A key concept in embedded systems is the need to process external stimuli, like a button press, or when a sensor detects a change in its environment. However, the microcontroller may already be busy executing another long running task - maybe it's waiting on a second sensor, or its busy updating a large display. The most efficient way to handle this issue is with *interrupts*.

An interrupt is a signal that is generated by the hardware or software when an event occurs that needs immediate attention. Once it's fired from an interrupt source, say a rising edge on a GPIO pin, the interrupt signal arrives at the CPU, which - if the conditions are right and the correct bits are set - saves what it's doing, and executes a special function called an interrupt service routine (ISR), also called an interrupt handler.

An interrupt is a special type of exception. Others relevant to MCUs are faults, traps, and resets. Peripherals can raise interrupts. The CPU can be interrupted by more than one event, each of which has its own ISR stored in the interrupt vector table. Interrupts can be enabled or disabled. Interrupts may have different priorities, and higher priorities can preempt lower priorities. This means that when the CPU is servicing one interrupt, an interrupt with a higher priority can interrupt and make the CPU service it instead.

The flow of an interrupt is thus:

1. A peripheral raises an interrupt.
2. The CPU checks if N is enabled.
3. If N is enabled, the CPU marks N as pending.
4. The CPU checks the priority of N versus the current priority level, as it might be serving a higher priority interrupt.
5. If the priority of N is greater than the current priority level, then the CPU updates the current priority level to the level of the new interrupt and pushes the CPU state to the stack. It then puts the Nth entry in the vector table into the program counter, a special register that keeps track of where the CPU is in the code. The ISR is now running.

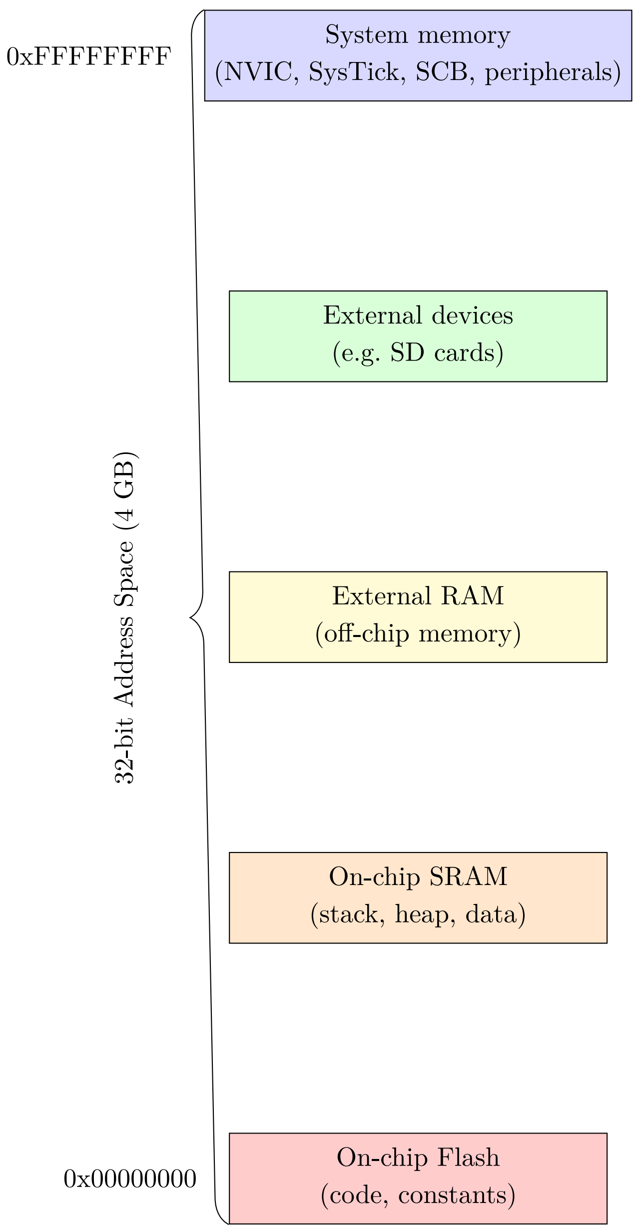

Figure 4: MCU Memory Layout

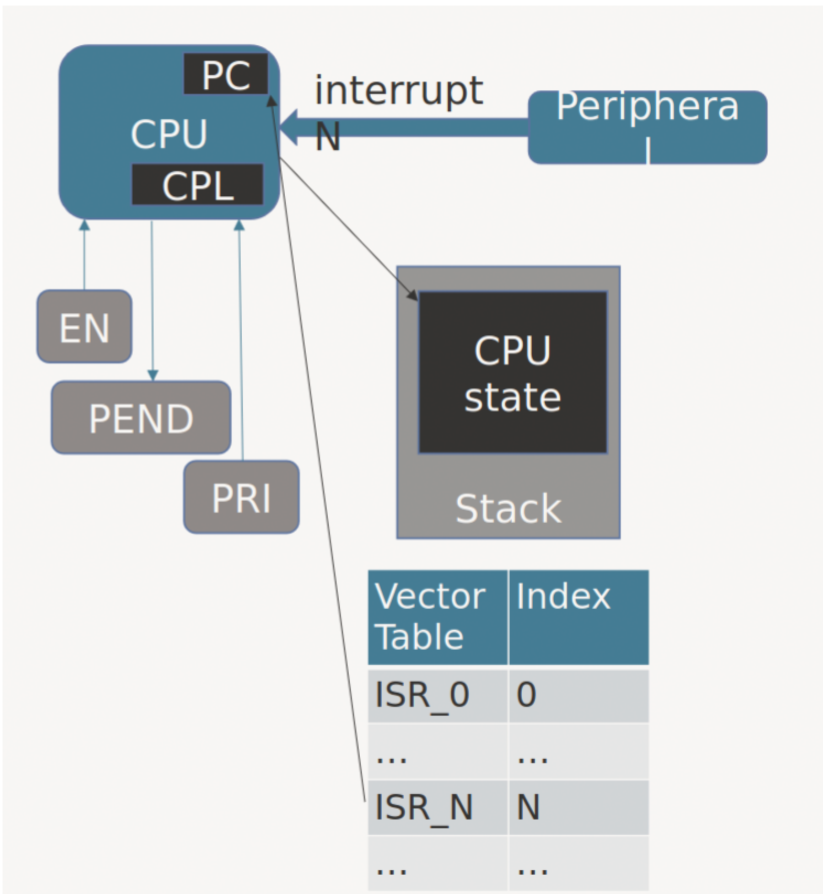

Figure 5: Interrupt Flow