Notes for ECE 36200 - Microprocessor Systems And Interfacing

Zeke Ulrich

September 26, 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 1

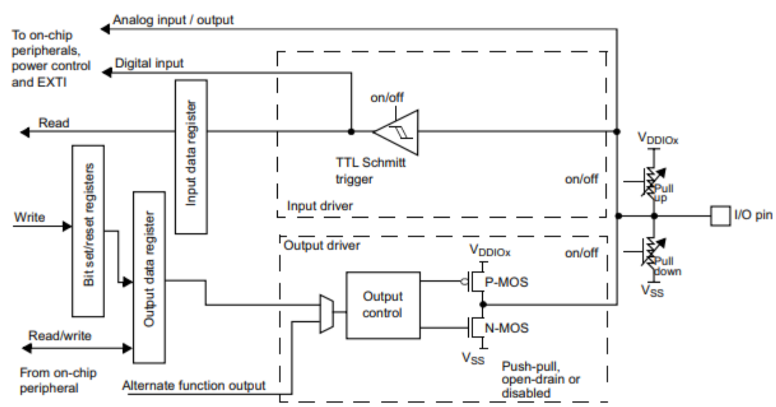

Figure 1: GPIO Layout

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 2.

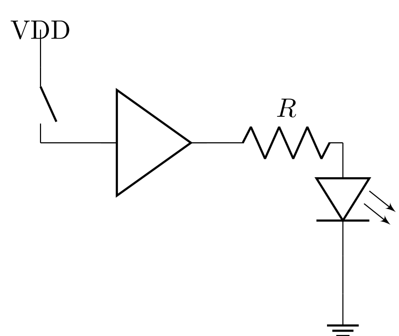

Figure 2: Faulty LED Control Circuit

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 3.

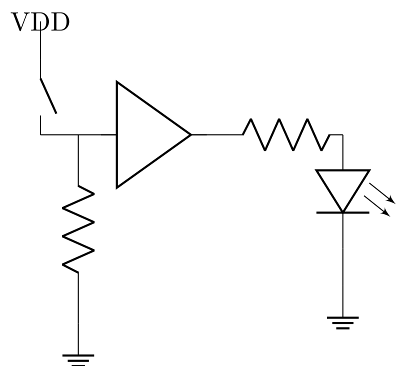

Figure 3: LED Control Circuit

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 4.

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.

A common exam question is "how many times was the interrupt handler interrupted?" or "how many times was the CPU state restored?".

Exceptions

We mentioned several other types of exceptions. While interrupts are usually handled at the end of instructions and the CPU resumes at the next instruction, faults happen in the middle of an instruction and execution resumes in the same instruction. Just as with interrupts, higher-priority exceptions may preempt lower-priority exceptions.

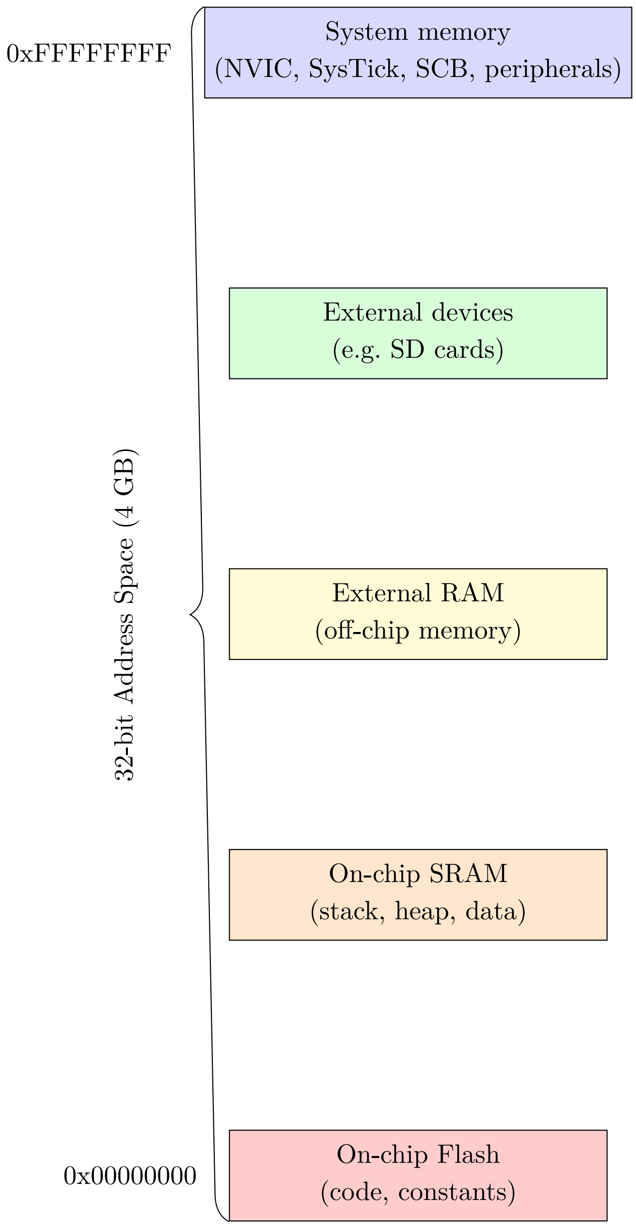

Figure 4: MCU Memory Layout

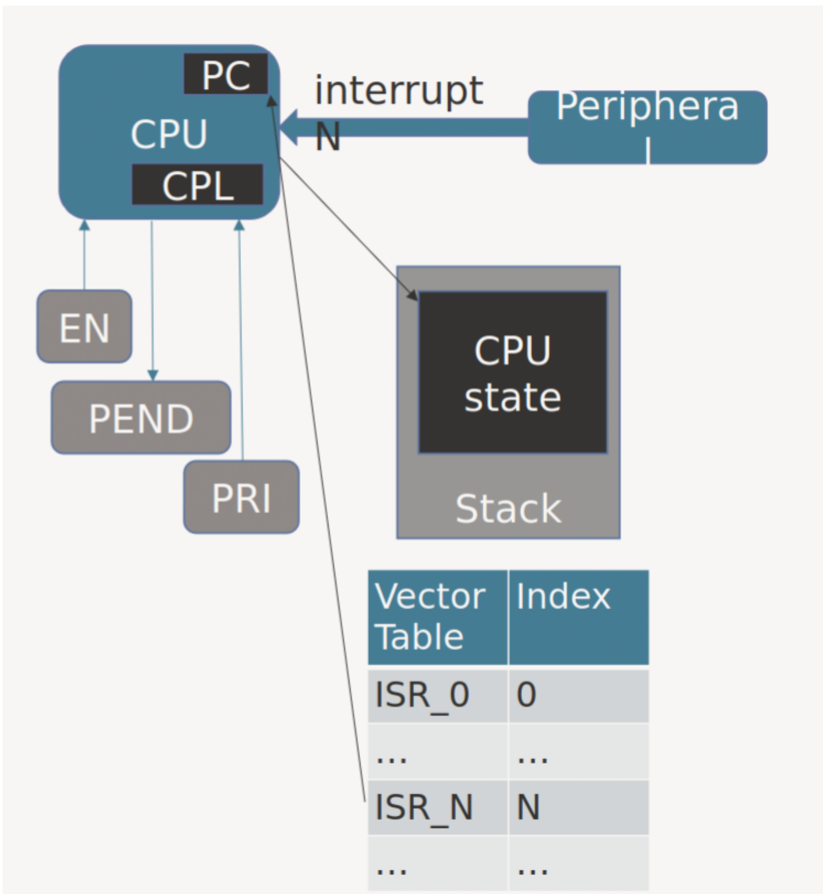

Figure 5: Interrupt Flow

Timers

In MCUs, we often want a way to periodically do something or many somethings. Every useful MCU has a hardware timer system. A *system ticker* (systick) is a special timer reserved for OS operations, but there is always a timer subsystem outside the CPU for general use.

The systick timer is a piece of hardware inside the MCU that generates an systick interrupt signal at fixed intervals. Every interrupt has a dedicated ISR to handle it, and in the case of systick the ISR is `SysTick_Handler`.

In the ARM Cortex-M, the system timer is built into the hardware of the CPU. Every time an IRQ is raised, the Nested Vectored Interrupt Controller (NVIC) hardware will determine whether or not to handle the systick interrupt.

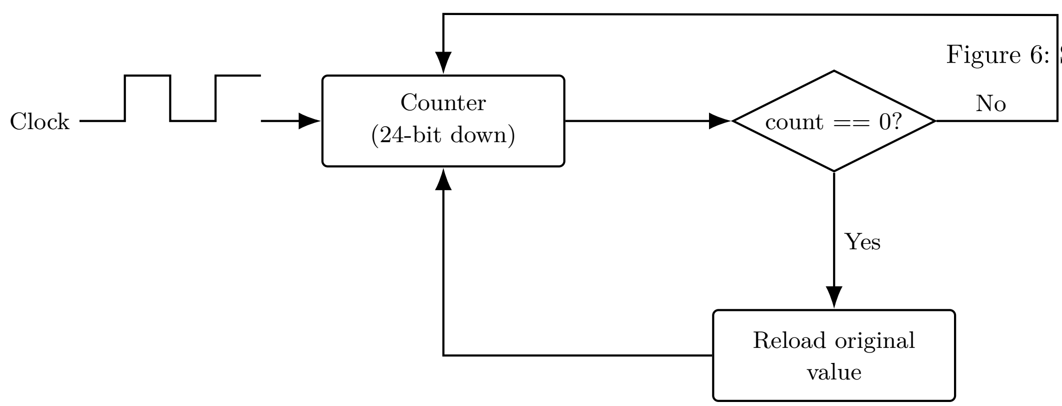

Figure 6: System Timer

When the counter hits zero, `COUNTFLAG` is set to 1. The choice of clock input can vary, as most MCUs have multiple. The clock input is ANDed with a flag to enable and disable.

For a timer, the desired interval T is

$$T = \frac{(\text{Prescalar} + 1) \times (\text{Reload} + 1)}{f} \tag{2}$$

Let's do an example. Suppose the clock tick frequency f is 80MHz and the goal is a systick interval s of 10ms. What must the reload value R be?

$$\begin{aligned} R &= sf - 1 && (3) \\ &= 10ms \times 80MHz - 1 && (4) \\ &= 800000 - 1 && (5) \\ &= 799999 && (6) \end{aligned}$$

Timers can do far more than just triggering interrupts. They can also

- Capture external events
- Drive certain peripherals like GPIO
- Output precisely timed signals

In output mode, hardware constantly compares the counter with a value stored in the *compare and capture register* (CCR), as in figure 7.

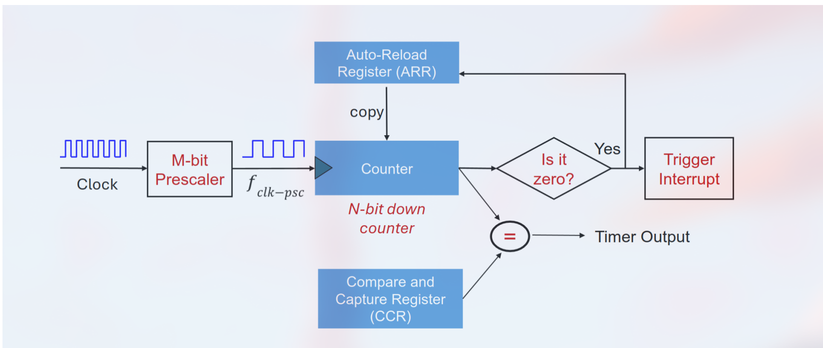

Figure 7: Timer Output Mode

In general purposes timers, you can count down, up, or up and down.

The output mode determines what happens when the CCR and counter are equal. One output mode is active high. When the timer value equals the CCR, then the timer output is set to high and stays there. Another output mode is toggle, where the output switches whenever the timer reaches the CCR value.

Pulse width modulation (PWM) is a convenient way to generate variable duty cycles. You just toggle the output each time it reaches a given value of the CCR.

$$\text{Duty Cycle} = \frac{\text{CCR}}{\text{ARR} + 1} \tag{7}$$

$$\text{Period} = (1 + \text{ARR}) \times \text{Clock Period} \tag{8}$$

PWM outputs a digital waveform (i.e. either 0 or 1), with a specific frequency and duty cycle.

We'd like to be able to have a DAC with a strong drive strength like we have with GPIO. We don't just want on/off states. We want multiple voltage levels. Consider a square wave with a varying duty cycle: We average the area under the curve. The duty cycle of the wave is then the analog level. We can average using a low-pass filter. As long as the wave frequency is much higher than the signal we want to model, no one will notice the averaging. We can accomplish averaging with low-pass filters. Each timer has a PWM mode where the output starts high, and goes low when CNT == CCRx. The higher the duty cycle, the higher the average. If the pulse frequency is much higher than that of the desired signal, the noise will be imperceptible.

Debouncing

When a mechanical button is pressed, there is a period where the signal is neither high nor low, because of vibrations in the conducting plate it connects or other non-ideal physical effects. This is known as *bouncing*, and it's rectified by adding a Schmitt trigger and RC circuit to smooth out the signal.

However, in real world systems we rarely have just one button. We often have a matrix input, like a keypad. Our Schmitt trigger setup doesn't work here, but luckily a software solution exists. Set up the CPU connected to the keypad to scan each key for being pressed. It doesn't need to check constantly, just often enough that it will catch a button press.

The work of scanning can be done incrementally using a timer interrupt. On each interrupt, the ISR will:

- read all the columns
- put the value read for each key of the current row into its own history byte
- turn off the voltage for the row
- turn on the voltage for the next row (for the next ISR invocation)
- return

The keys on a keypad can still bounce. Pressing and letting go of a button may look something like `00000001001011111...1111101000000`. In this example, the button bounces on press and release. However, the history byte eventually stabilizes and is full of entirely `1`s or `0`s. To detect a press or release, we search all the history bytes that represent the keys. The first time we detect a change, like in `00000001`, we say that is the start of a press or release. We say the press or release is done when the history byte queue is back to only one number.

We want to scan the keys faster than they can be pressed or released, but slower than the total bounce time for any key. Say a button can bounce for 10ms, and we scan one row of the 4-row keypad every 1ms. Then when you press the button, the instance you read a one, you do not know if the input is stabilized. It could still bounce, since it has been 0ms. The second time you read a bit, it could still be bouncing, since 4ms is within the 10ms bounce time. The third time you read a bit, it could still be bouncing, since it has only been 8ms. By the fourth bit you read, you are certain it is a one.

Multiplexing

Multiplexing is when you rotate through a task fast enough that it gives the appearance of being simultaneous. Key scanning is a specific example of input multiplexing and encoding. Another example is driving displays. Turn on one SSD at a time. Rotate through them rapidly enough that your persistence of vision makes it appear they are all on simultaneously and displaying different digits.

Direct Memory Access

Moving memory from one location to another makes poor use of your CPU. Moving data is so common that we build a co-processor called a *Direct Memory Access* (DMA) controller to do this without our CPU.

DMA is also useful for allowing peripherals to use memory independent of the CPU

The workflow for reading data from a peripheral with a CPU is:

1. Copy data from peripheral data register to buffer in memory

2. Copy data from buffer in memory to CPU

A DMA can just copy data directly from the register to the buffer. There are two kinds:

- Flow-through: DMA controller is used as an intermediate buffer (useful if we need to change the size of data type)

- Fly-by: DMA controller only sets up the bus between the source and destination

A DMA sits on the bus, stores a list of source/destination pairs being serviced, and is triggered by software, peripherals, timers, or other interrupts.

Digital-to-analog Converter

A digital-to-analog converter (DAC) converts digital data into a voltage signal.

$$\mathrm{DAC}_{\mathrm{output}} = V_{\mathrm{ref}} \frac{\text{Digital Value}}{2^N - 1} \tag{9}$$

N is the *resolution* of the DAC, and 2^N is the number of voltage levels the DAC can generate.

Applications of DACs include digital audio, video display, and waveform generation.

A 5-bit DAC looks like Figure 8.

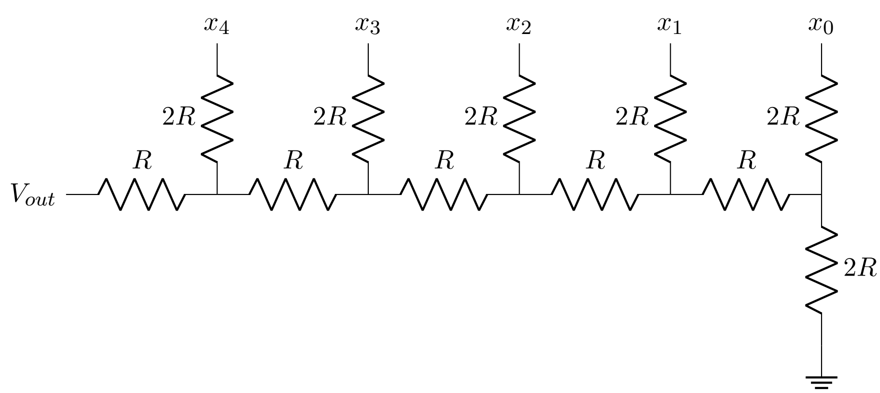

Figure 8: 5-bit DAC

The DAC conversion is nearly instantaneous. Some settling time is required due to capacitance and inductance in the circuit. No iteration or convergence is needed. However, resistor tolerance is a limitation. The higher N of the DAC, the more precise the resistors must be.

In place of a resistor ladder, a 2^N-to-1 mux can be used to select one voltage level instead.

Analog-to-digital Converter

As the name implies, an *analog-to-digital converter* (ADC) converts analog sound to a digital signal, like a microphone. The *resolution* N of an ADC is the number of binary bits in the ADC output.

An analog signal is continuous time and continuous amplitude, while a digital signal is discrete time and discrete amplitude

$$\text{ADC Output} = round\left((2^N - 1)\frac{V_{\text{input}}}{V_{\text{ref}}}\right) \tag{10}$$

V_{ref} is the maximum input voltage that can be converted by the ADC.

The least significant bit (LSB) holds the lowest value of the encoded analog signal. The maximum error due to quantization is

$$\pm\frac{1}{2}LSB = \pm\frac{1}{2}\frac{V_{\text{ref}}}{2^N - 1} \tag{11}$$

An ADC does not have infinitely fast conversion. The *sampling rate* is the number of output samples available per unit time.

The sampling frequency needed to prevent aliasing, which causes distortion, is given by the Nyquist-Shannon Sampling Theorem,

$$f_{\text{sampling}} \geq 2f_{\text{signal}} \tag{12}$$

This class focuses on *successive-approximation* (SAR) ADCs, as in Figure 9.

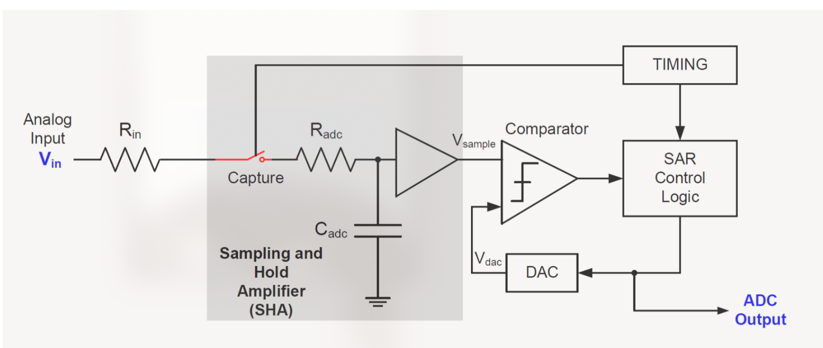

Figure 9: SAR ADC

Jitter is small rapid variations in a waveform resulting from fluctuations in the voltage supply or mechanical vibrations or other sources. Jitter is noise from the surroundings, from the microcontroller, and from the ADC itself. One of the most reliable ways of defeating noise is taking more samples than needed and averaging them out.

One type of this averaging out is called "moving window" sampling, so named because one averages N samples in a small section (window) of a signal, then moves the window by one and repeats the process.

```
#define HISTSIZE 128
```

```
int his1[HISTSIZE] = { 0 };
int sum1 = 0;
int pos1 = 0;

int his2[HISTSIZE] = { 0 };
int sum2 = 0;
int pos2 = 0;

...

reading = ADC1->DR;
sum1 -= hist1[pos1];
sum1 += hist1[pos1] = reading;
pos1 = (pos1 + 1) % HISTSIZE;
val = sum1/HISTSIZE;
sprintf(line, "%2.3f", val * 3 / 4095.0);
display1(line);

ADC1->CHSELR = 0;
ADC1->CHSELR |= 1 << 5;
while(!(ADC1->ISR & ADC_ISR_ADRDY));
ADC1->CR |= ADC_CR_ADSTART;
while(!(ADC1->ISR & ADC_ISR_EOC));

reading = ADC1->DR;
sum2 -= hist2[pos2];
sum2 += hist2[pos2] = reading;
pos2 = (pos2 + 1) & (HISTSIZE - 1);
val = sum2 >> 7;
// division is painfully slow
// if HISTSIZE is a power of 2
// replace divisions with a
// bitwise AND operation and
// right shift
sprintf(line, "%2.3f", val * 3 / 4095.0);
display2(line);
```

Serial Peripheral Interface

MCUs often need to communicate with another MCU or peripheral, on the same board or across boards. Parallel interfaces can send and receive multiple bits at the same time. Serial interfaces, however, send and receive one bit at a time.

It may seem like parallel is the obvious choice, but of course we wouldn't introduce serial interfaces if they didn't have some advantages. Parallel interfaces require more pins, wires and larger PCB footprint. Wires next to one another leads to crosstalk, electromagnetic interference. In practice systems tend to use serial interfaces. Since such interfaces send and receive only one bit at a time, there can either be one wire for both directions or one wire in each direction.

Cross-talk can be mitigated with twisted wires, but not eliminated

There are two kinds of peripherals, *synchronous* and *asynchronous*. Synchronous protocols have a clock that dictates the send/receive rate. Examples include Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I2C). Asynchronous protocols have no clock. The most famous example of an asynchronous protocol is Universal Asynchronous Receiver and Transmitter.

SPI is one of the most common serial interfaces. It operates in Master/Slave mode. There must be at least one master, while there can be multiple slaves. The master defines the clock and delivers a clock signal to each data recipient. It operates by shifting in bits each clock cycle.

API devices are designated as masters or slaves. A master device initiates all data transfer operations. It drives the clock pin and data pin. It also drives slave select pins if slaves are selectively enabled. A slave device responds to transfer operations. Signals are named according to the devices' standpoint. E.g. MOSI: master out, slave in is the data transmitted by a master device and received by a slave device.

What does this look like in practice? Say we wish to send `0xB1`. Each rising clock edge latches one bit into the slave. First, the master selects the slave by asserting the slave select signal. We send 8 bits, so there are 8 clock pulses. Finally, one each clock pulse, one bit is transferred.

The master-slave terminology is being phased out in industry. Updated datasheets might refer to SPI pins with terminology such as

The API clock doesn't need to be perfectly periodic and continuous. As long as it's not too fast, it can pause as needed since the slave takes the data bits only at clock pulses anyway.

SPI supports multiple slaves, which can share MOSI, MISO, and clock lines. However, each slave needs its own select signal line. The slave selected drives the MISO line. No more than one slave may be

Legacy Pin Name	Updated Pin Name
MOSI (Master Out, Slave In)	SDO (Serial Data Out), TX/TXD (Transmission)
MISO (Master In, Slave Out)	SDI (Serial Data In), RX/RXD (Reception)
SCK (Serial Clock)	SCK (Serial Clock)
NSS (Negative Slave Select)	CS (Chip Select)

Table 1: SPI Pin Names

selected at a time. SPI also supports more than one master. It is an error for multiple master devices to assert CS at the same time. Some coordination is needed to ensure that one master device reads/writes at any time.

SPI is supported by LCD displays, OLED displays, SD card interfaces, and hundreds of other devices.