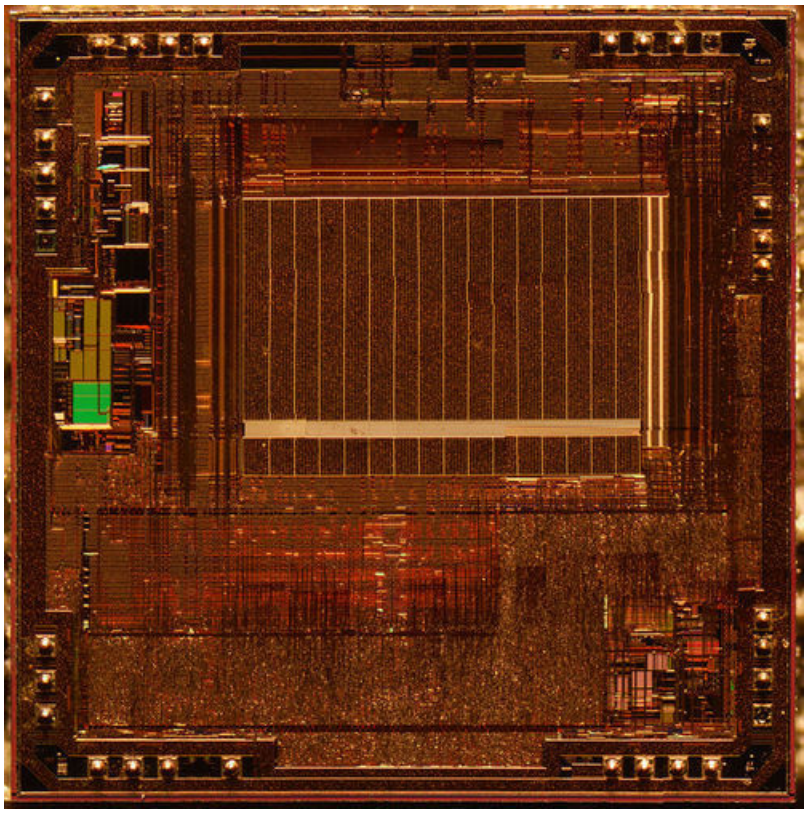
adafruit learning system

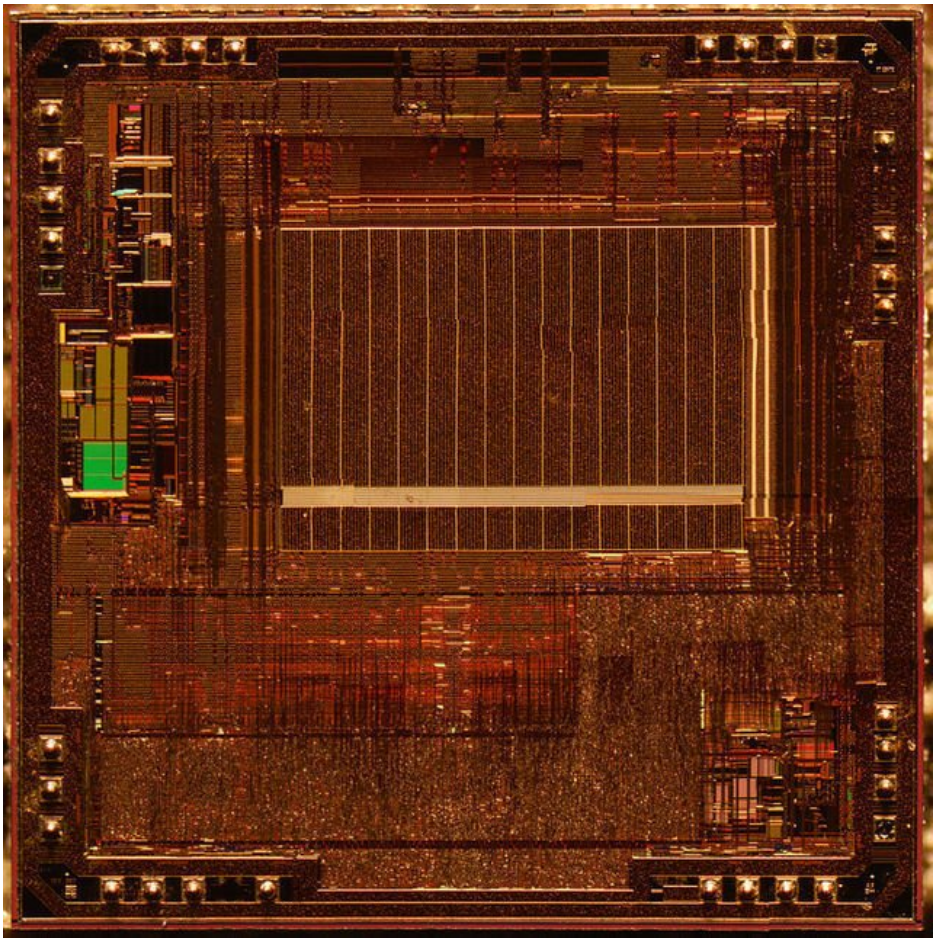
Digital Circuits 7: MCUs... how do they work?

Created by Dave Astels



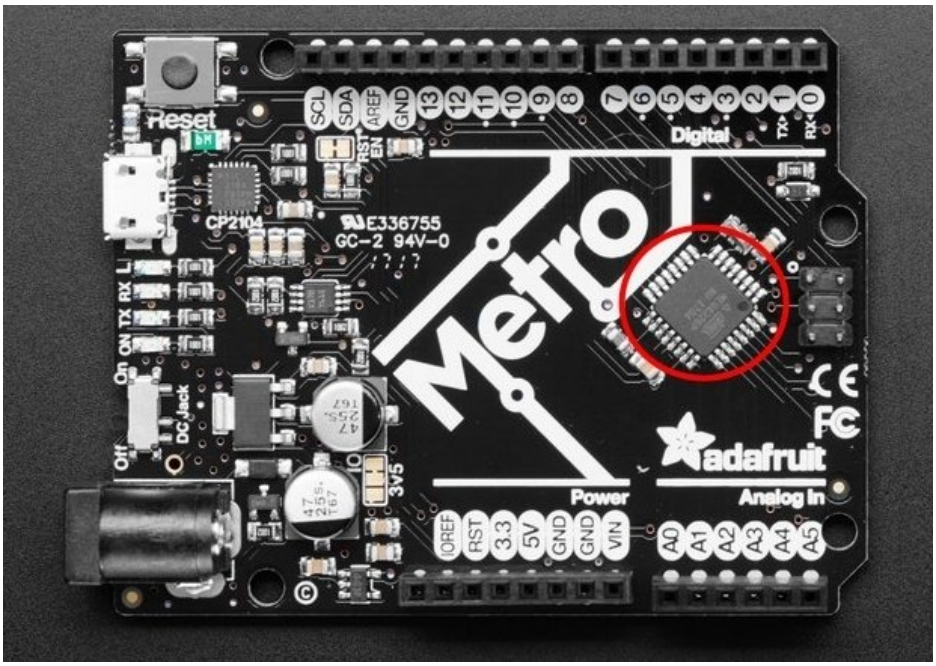
Last updated on 2020-06-15 05:55:21 PM EDT

Overview



Above is a picture of the ATmega328 silicon die/chip. This is what's inside that large, long IC on many Arduino UNO compatibles and the small square IC on an Adafruit Metro Classic (https://adafru.it/METROXMETR) (below) and on an Arduino pro mini. Both packages appear on a number of other comparable microcontroller boards.

In this guide we'll take a tour of the '328, looking at each significant functional piece. This guide is going to be fairly high level, most of the concepts will be applicable to any microcontroller. For full details on the '328 see the datasheet from Microchip (https://adafru.it/Bjh) (note it is very long, technical, and sleep inducing).


Reset
SCL
SDA
AREF
GND
13
12
11
10
9
8
7
6
5
4
3
2
1
0
Digital
TX
RX
RST
EN
CP2104
E336755
GC-2 94V-0
Metro
L
TX
RX
ON
On
Off
DC Jack
3V3
adafruit
Power
Analog In
IOREF
RST
3.3
5V
GND
GND
VIN
A0
A1
A2
A3
A4
A5

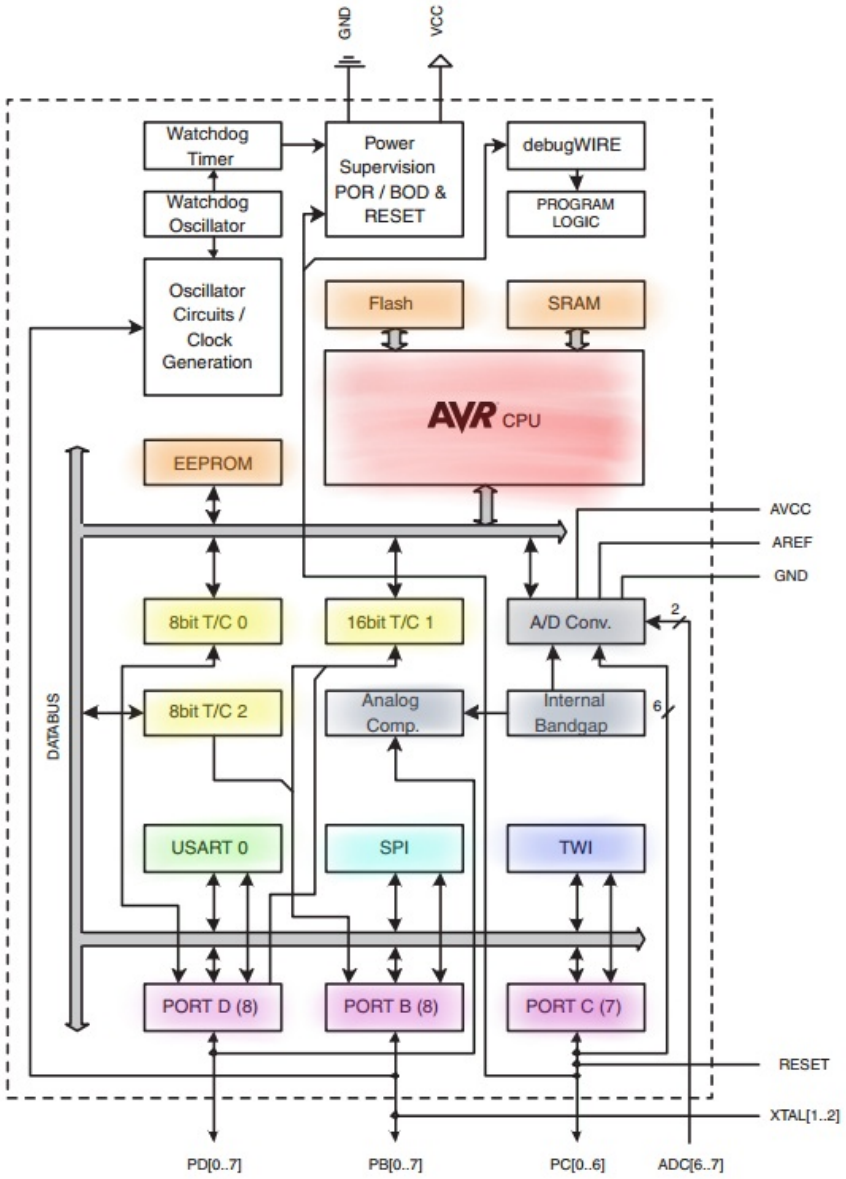
The building blocks

Let's start by breaking the '328 down into it's functional pieces. Then we'll look at each in turn.

Below is a block diagram of the '328 that identifies the major functional areas:

- the AVR Central Processing Unit CPU (red)
- memory: Flash, random access memory (RAM), and electrically erasable programmable read only memory (EEPROM) (orange)
- timers/counters (T/C 0, 1, and 2) (yellow)
- analog inputs (grey)
- digital I/O (megenta)
- serial I/O (green)
- SPI (cyan)
- I^2C/TWI (blue)
- control and misc circuitry (white)

Let's walk through each section.


GND
VCC
Watchdog Timer
Watchdog Oscillator
Power Supervision POR / BOD & RESET
debugWIRE
PROGRAM LOGIC
Oscillator Circuits / Clock Generation
Flash
SRAM
AVR CPU
EEPROM
AVCC
AREF
GND
8bit T/C 0
16bit T/C 1
A/D Conv.
2
8bit T/C 2
Analog Comp.
Internal Bandgap
6
DATABUS
USART 0
SPI
TWI
PORT D (8)
PORT B (8)
PORT C (7)
RESET
XTAL[1..2]
PD[0..7]
PB[0..7]
PC[0..6]
ADC[6..7]

CPU

The CPU (Central Processing Unit) is the piece that runs the show. Everything else revolves around it. The CPU used in an MCU is the central defining feature. You will see MCUs described by the CPU they contain: *AVR*, *ARM Cortex-M0+*, etc. The CPU defines the the *size* of the MCU, i.e. the size of the internal data bus. This is indicative of how big of a number can be operated on at once. An 8-bit CPU can operate of 8-bit bytes (values from 0 to 255). In comparison an ARM Cortex CPU has a 32-bit bus and works on 32-bit words (values 0 to 4,294,967,295). That's a bit of a difference.

The CPU also sets how fast the MCU operates, eg. the AVR core in the ATMega328 runs at 16MHz while the M4 core in the SAMD51 runs at 120MHz. A far cry from a 4GHz Intel i7 (which is also 64-bit), but fast for an MCU.

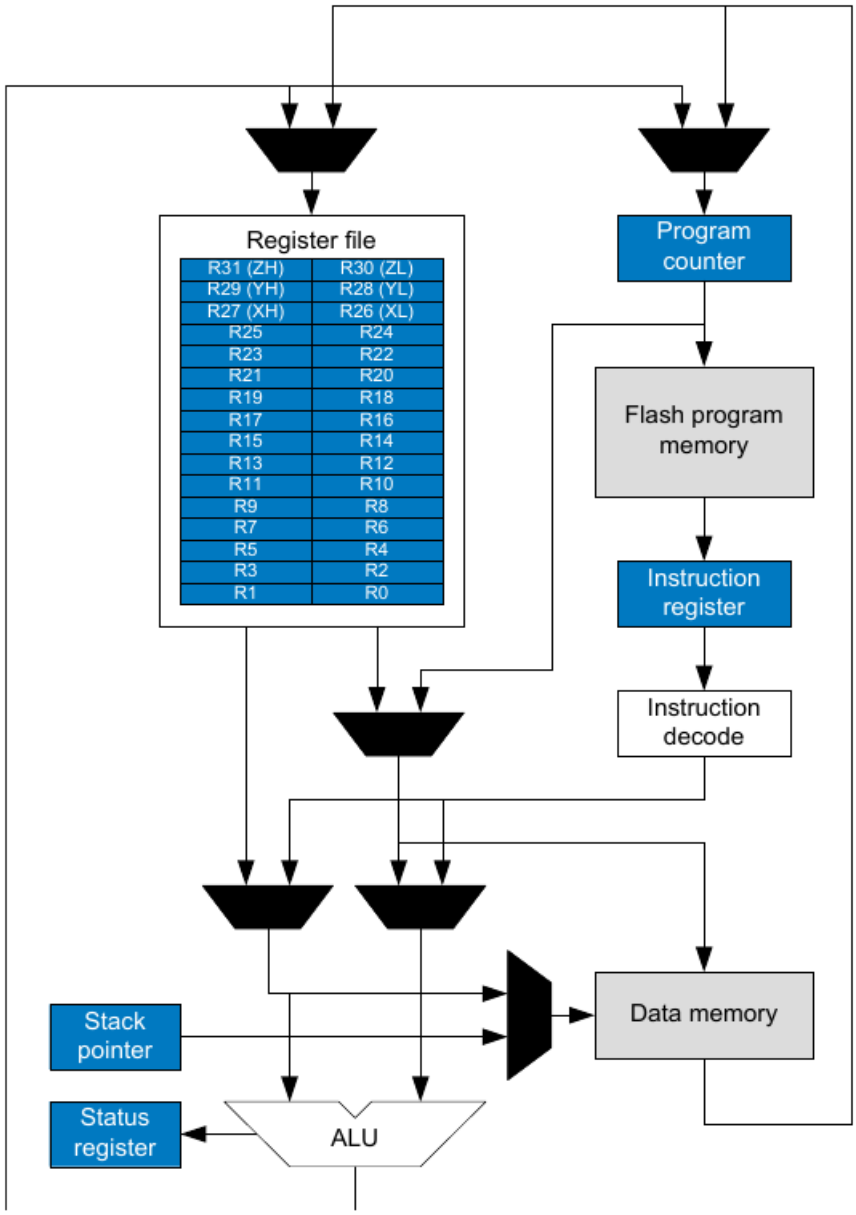
The CPU is the piece that executes your code and makes use of the rest of the MCU. We'll look at CPUs in more detail in a later guide.

The CPU in the '328 is an 8-bit AVR core that Amtel started developing in 1996 (Atmel is now part of Microchip). The '328 uses what's called a Harvard architecture, which means that it has separate memory systems for program (flash) and data (static RAM) storage. To enhance performance, the CPU has 32 general purpose 8-bit internal memory locations, called registers. These are part of the CPU, meaning no external memory access needs to be made to read or write them. If data can be kept in registers as much as possible, performance can be kept very high, even though the whole microcontroller has a relatively slow, small CPU.

The rest of the CPU is fairly typical:

- a program counter to keep track of where the instruction being executed is,
- instruction decode logic to make the rest of the CPU do what it is required to execute each instruction,
- registers to hold data in the CPU for manipulation
- an Arithmetic and Logic Unit (ALU) to preform math and logic operations,
- a stack pointer to help manage function call and return as well as function local data,
- and a status register to keep track of CPU state (e.g. the last operation resulted in a zero, or a carry, etc.) and control conditional jumps.
- support circuitry (mostly buses and multiplexers) to move data around inside the CPU as well as between it and the system's memories.

The diagram below shows these pieces and how they connect/communicate. Note that the grey memory blocks are not part of the CPU.


Register file
R31 (ZH)
R30 (ZL)
R29 (YH)
R28 (YL)
R27 (XH)
R26 (XL)
R25
R24
R23
R22
R21
R20
R19
R18
R17
R16
R15
R14
R13
R12
R11
R10
R9
R8
R7
R6
R5
R4
R3
R2
R1
R0
Program counter
Flash program memory
Instruction register
Instruction decode
Stack pointer
Status register
ALU
Data memory

Memory

We looked at memory of various sorts in Part 5 (https://adafru.it/Bji) of this series. The '328 is a typical MCU in that it has some flash (32K) as well as some RAM (2K). It also has a bit of EEPROM (1K). That's not a lot, but plenty for a simple embedded system that typically doesn't have to do much.

To compare with early microcomputers, a loaded Apple][+ had 12K of ROM (somewhat comparable to flash) and 48K of RAM. This is a flip from what we see in MCUs that typically have far more flash than RAM. The big difference is that all of the program code typically goes into the flash on the MCU, whereas the ROM on the Apple][+ (and general purpose computers in general) have enough to get a program loaded into RAM and execute it.

It's interesting that CircuitPython on the new ARM based boards pretty much goes back to this earlier model: flash holds the CircuitPython runtime which loads your Python code from the flash filesystem (accessed as a USB drive) into RAM, where it's executed.

Speaking of the ARM chips, the lastest MetroM4 board has a SAMD51 MCU with 512K of flash and 192K of RAM, plus additional external flash.

When you write a program using the Arduino environment and load it onto a board, it gets stored in flash. RAM is used for your variables and the program stack.

Timers/Counters

Have you used pulse-width modulated (PWM) outputs to dim a light emitting diode (LED) or control the speed of a motor? Then you've used timer/counters. To generate a PWM signal, a counter counts from 0 to it's maximum, let's say that's 256 to keep things simple (so that'd be an *8-bit counter*). A PWM generator has its output low when the count is below a specified value, and high when it's equal to or above. If we wanted full brightness on an LED, we'd set that PWM threshold to 0 so that the output will always be high. If we wanted it at 50% we'd set the threshold to 128 (assuming a fairly linear response curve for the LED, which is probably reasonable at the 50% point, but generally not near the minimum and maximum values). That way the output would be low half the time and high half the time.

Counters are useful when you need to keep track how often some event occurs. If that event can be converted to a digital pulse, you can count it. If the event corresponds to one revolution of a wheel, for example, counting them and checking the count every so often would tell you how fast the wheel is turning.

That brings us to another use of timers: keeping track of time. We can set up a timer to let us know every time a second has gone by. We could regularly check the time to see if a second has passed (which is usually referred to as *polling)* or we could have the timer generate an interrupt every second. Which we use partly depends on how critical it is that we do something as soon as possible when that time is up. We could set this up to happen once (let me know in one second) or repeatedly (let you know once per second, every second). It all depends on what you need to do.

The '328 has three timers, each with somewhat unique capabilities. TC0 and TC2 are 8-bit while TC1 is 16-bit. They provide a wealth of features that we won't go into in detail.

Analog Input

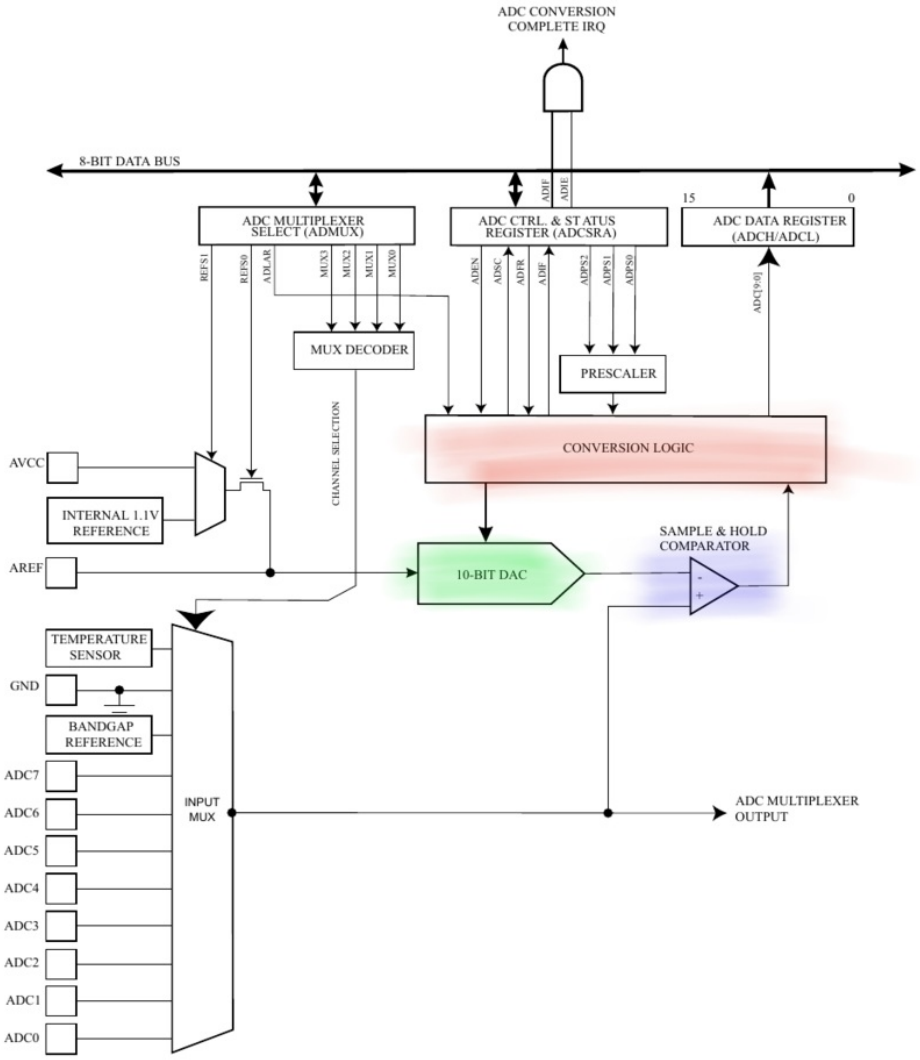
MCUs are typically used to interface with the real world, and that world is analog. That means that instead of the binary high and low logic levels, voltages vary smoothly. Digital electronics (which includes MCUs) can't directly work with analog voltages, so they need to quantify such voltages into digital values.

This quantification involves converting the analog signal into a number with some number of bits. As a simplest case example, we could have a 1-bit analog to digital converter. If the analog signal is below half Vcc (2.5 volts for a 5 volt MCU) we could convert it to a low (aka 0), otherwise we could convert it to a high (aka 1). The more bits we convert to, the better the approximation to the actual analog signal.

Measuring light using a photocell is a great example of this. It's a light sensitive resistor that you use in series between Vcc and ground with another resistor. As the strength of the light falling on the photocell changes, so does its resistance, and so does the voltage at the point between it and the other resistor. If we measure that voltage we can get a sense of what the strength of the light is. For example, assume that we're measuring a signal between 0 and 5 v. Let's say our input signal is 2 v. That's 2/5 of Vcc, i.e. 40%. If our ADC is 8-bit (i.e. converts to an 8 bit number) the result would be 102. If it's a 16-bit ADC the result would be 26,214. The more bits in the ADC result, the finer it's resolution, i.e. the less change each increase of 1 in the result indicates (0.195 v for an 8-bit ADC, 0.0000763 v for a 16-bit ADC). 10 or 12 bit ADCs are pretty typical. The '328 has a 10-bit ADC, so it's resolution is 5 v / 1024 = 0.00488 v.

Although the '328 has a single ADC, it has an analog multiplexer providing 6 inputs, one of which can be read at a time. These are similar to the digital multiplexers we've seen in previous parts of this series, but work with analog signals rather than digital ones.

So, if more bits gives you a more accurate measurement, why not have as many as possible? Why 10 bits rather than 16? Have a look at the block diagram below of the '328's analog to digital converter. See that red block labeled "CONVERSION LOGIC"? It outputs a 10-bit value to a digital to analog converter (DAC) (in green). That seems a little backwards.. what's up with that? Well, see how the output of the DAC feeds into comparator (in blue)? The other input to the comparator is the output from the analog multiplexer, i.e. the analog signal being measured. The contents of that conversion logic block is a successive approximation circuit. Its output is converted to an analog signal and compared to the input. The result of that comparison feeds back to improve the next approximation.



What's successive approximation? It's a way to incrementally find an answer by trying a guess, seeing how it works out, then adjusting the guess to get closer to the desired outcome. It's like a game of "getting warmer". Let's say we start at the mid-point of the conversion range. Let's use 2 volts from above as the input signal with the 10-bit converter.

We start with a guess of 512. That converts to 2.5 v which is greater than the input (2 v). That means that the next guess needs to be lower. A straightforward approach is to use a binary search technique: make the next guess 256. That converts to 1.25 v which is less than the 2 v input. The next guess will be 384 which converts to 1.88 v which is still less than the 2 v target. So the next guess will be larger: 448 which is 2.19 v. Now it's too big so the next guess has to be lower: 416 which is 2.03 v. That's getting close but still high. The next guess is 400 (1.95 v). Up for the next guess: 408 (1.99 v). Up a bit more for the next guess: 412 (2.01 v). That's just over the target: down for the next guess of 410 (2.00 v). Bingo. Let's look at what we just did.

512 (2.5v) too high (decrease by 256)
256 (1.25v) too low (increase by 128)
384 (1.88v) too low (increase by 64)
448 (2.19v) too high (decrease by 32)
416 (2.03v) too high (decrease by 16)
400 (1.95v) too low (increase by 8)
406 (1.99v) too low (increase by 4)
412 (2.01v) too high (decrease by 2)

410 (2.00v) just right.

The key is to change the guess each time by decreasing powers or 2. This will gradually converge on the target value. It can take several steps to do that. In fact, in the worst case the number of steps required will be the number of bits in the resolution of the converter. So a 10-bit converter will take 10 steps in the worst case. If the converter has more bits, the worst case conversion time gets longer. That's why we don't just have 16 or 32 bit converters. There's always a trade-off. Always.

Digital I/O

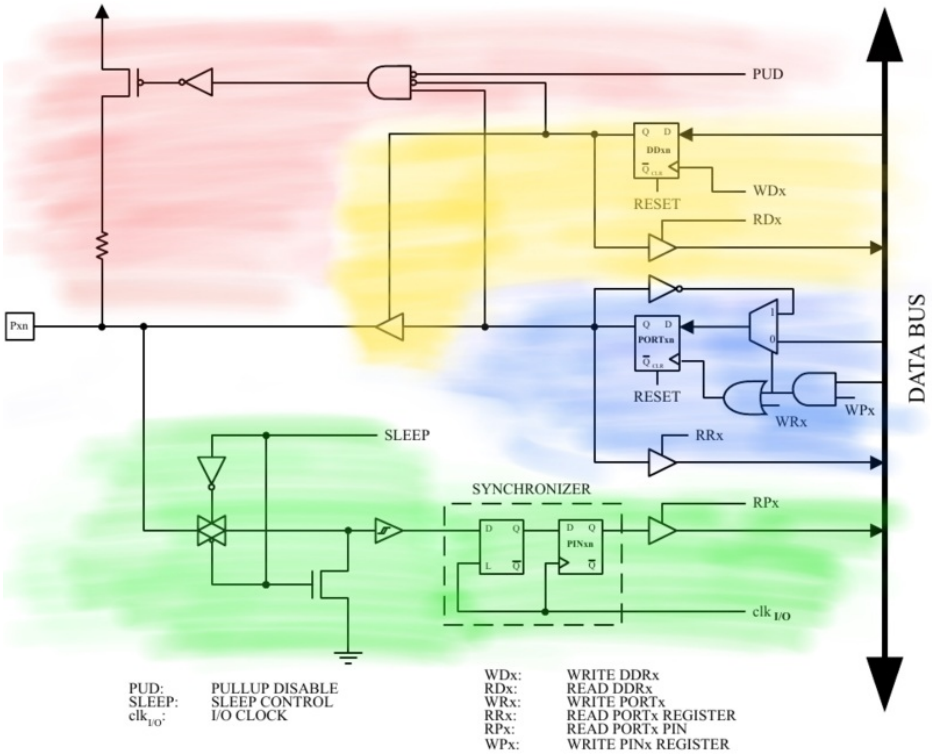
This one is straight forward. If you're reading this, you've more than likely used programming statements in the Arduino Integrated Development Environment (IDE) such as `pinMode`, `digitalRead`, `digitalWrite`, and the like. That's digital input and output (I/O). Digital I/O is just connections to the outside world that can be set high or low, or can read signals as high or low. The '328 has 3 digital I/O ports, 2 8-bit and one 7-bit. Most of the time you will be working with single bit inputs or outputs; the Arduino library has explicit support for this in the above functions. The digital I/O lines can be set (as the name suggests) to be inputs or outputs.

Below is the simplified design of an digital I/O pin.

The yellow section of the diagram controls the direction of the pin. The direction control enables/disables a 3-state buffer on the output line. If enabled, the output signal (generated by the blue section) is routed through to the pin.

Additionally inputs can have a pullup resistor attached internally. Pullup resistors (or simply *pullups*) are used to make sure that an input line is held at a logical high unless whatever it's connected to forces it low. In the diagram, the pullup circuitry is highlighted in red. The AND gate enables the pullup (connecting the top of the resistor to Vcc) when the pin is outputting a logic high, or the pin is being used as an input and a pullup has been requested on the pin.

The green section reads the value of the pin, disconnecting it when in sleep mode, and cleaning up the timing with a couple D-FlipFlops (the SYNCHRONIZER).



Bigger MCUs with more pins will generally have more digital I/O lines letting you connect to more things: more switches, more lights, more solenoids, etc.

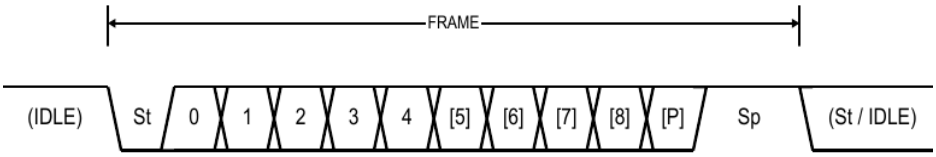
Serial I/O

Really, SPI and I^2C are serial as well, but I'm treating them separately; they are separate circuits on the '328 and provide different protocols.

So when we talk about serial input and output (I/O), we generally mean I/O using a USART (Universal Synchronous Asynchronous Receiver Transceiver).

Serial means one bit at a time. If we want to communicate an 8-bit value, one bit goes over the wire at a time. For an 8-bit parallel connection, all 8 bits go over 8 separate wires at the same time. The advantage of serial is that only 1 wire is needed for the data, the disadvantage is that it takes longer (one bit at a time!) and timing is more of an issue.

So now we have *synchronous* and *asynchronous*. This has to do with how we handle the timing issue. I.e. how can we indicate/determine when each bit is on the wire. For synchronous communication a clock is provided along with the data. Data values are synchronized with edges of the clock signal. For example data could be valid (and should be read) on a falling edge of the clock. In the '328 this is programmable. For asynchronous communications, both sides agree on how fast data will be transferred (the baud rate). There is a standard frame format involved to transfer one word of data. These standards have been around for a long time.

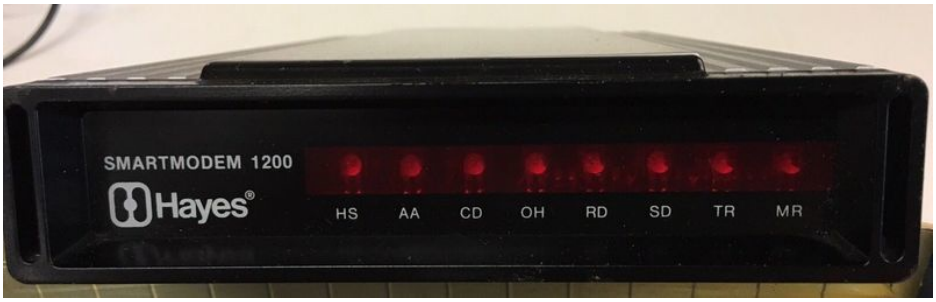


Formats

A word can be between 4 and 8 bits, and there is an optional parity bit that is used for error detection. If a parity bit is present, the receiving hardware can use it's value to detect a problem in the transmission. The parity bit is computed by XORing the bits of the data word. That will be the value of the parity bit if even parity is specified. The value for odd parity is that result inverted. Like the baud rate, word size and parity have to be agreed upon.

The line is usually held high, and transmission of a word begins with it going low for a bit's worth of time. Then it's followed by the data bits, possibly a parity bit, and a double width stop "bit".

Serial I/O takes a minimum of 3 wires: transmit (Tx), receive (Rx), and ground. More lines can be used to coordinate communication. Back in the day serial was one of the staple communication methods between devices. Modems which allowed data communication over the telephone system (anyone see the movie *WarGames* or the show *Halt and Catch Fire*) were serial in nature.



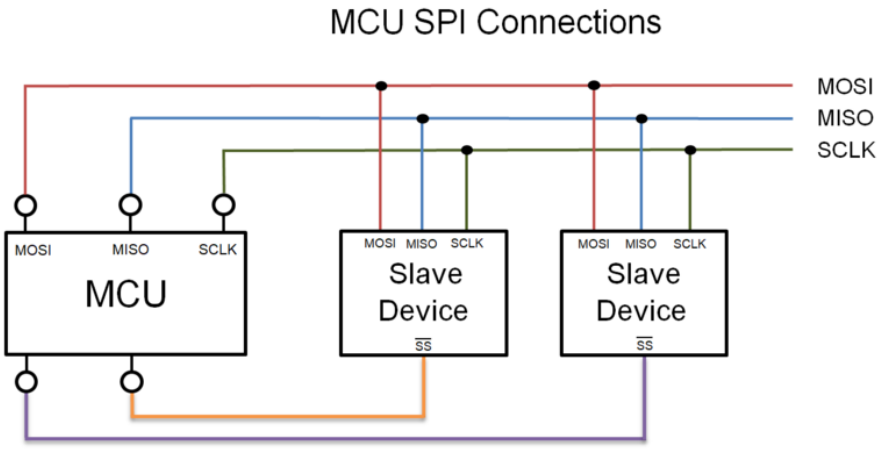
Printers were often serial as well. While parallel printer interfaces were common with home computers, larger installations had their printers some distance from the computer, and serial interfaces were much better at handling longer cables. Terminals (keyboard and display) were serial.

Even these days MCUs will pretty much always have serial I/O support, although it's often used with USB for connecting to a Windows/OSX/Linux system. Most MCU boards have an onboard converter to go between USB and serial (e.g. the Arduino UNO) or have USB support directly in the MCU itself (e.g. the ATmega32u4 MCU). Boards that don't (I'm thinking of the Arduino Pro Mini that uses the '328) will have an FTDI connector. That's just a serial interface with a specific arrangement of signals on a pin-header to which you can connect a USB <-> serial cable, or simply a logic-level serial cable.

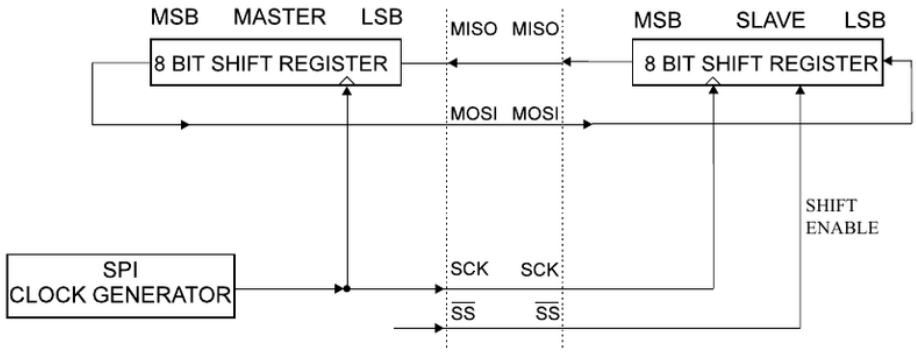
Serial support on MCU boards is now typically logic level (i.e. signals are 0v for a low, or 3.3/5 v for a high). This is fine because serial connections these days are typically short: between the MCU board and a computer, or MCU board and a GPS sensor (which are often serial). When serial was used to connect terminals and printers to minicomputers, those terminals would be in a different room, a different floor, or even a different building. Logic level signals don't work over those distances. A standard called RS232 was developed to handle the challenge. It uses higher voltages

(+/- 12 v was common) to generate more robust signals that could be used over those distances without degrading too much.

SPI



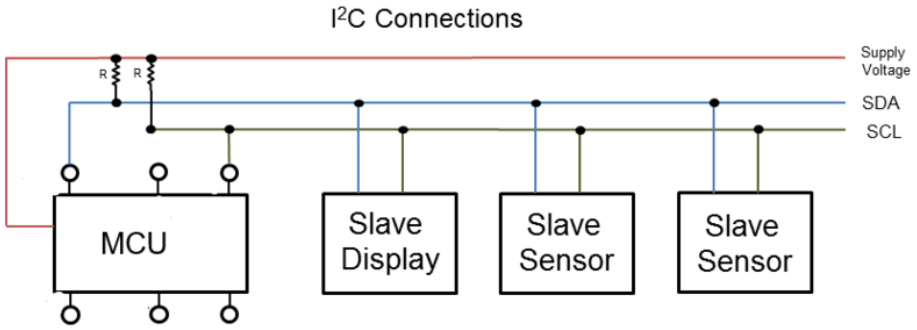
SPI Is another serial communication interface. It stands for *Serial Peripheral Interface.* SPI is full duplex (it can send data in both directions at the same time due to having separate lines for data travelling in each direction: MISO and MOSI, like serial with its Rx and Tx) and uses a shared clock signal. It's high speed and designed for very short distance connections. Examples of SPI peripherals are SD Cards and LCD displays. If you're using any one the *M0 Express* boards, you'll be familiar with another use: external SPI flash chips. Finally, the ICSP connector on the Arduino UNO and the like is simply a SPI connector. One thing I've used SPI for is to connect the nRFL01+ mesh radio (https://adafru.it/Bjj) board to '328 based boards. Basically it's great for high speed peripherals that aren't far from the MCU.



The SPI main in the '328 is on the left and the secondary in the SPI device is on the right. Each side has an 8-bit shift register that moves data in and out, using the clock generated by the main. That's pretty much it. The one other thing with SPI that you may have noticed is the select signal. Multiple SPI secondary devices can be connected to a single main, but each needs a unique select signal from the main to enable it. The select signal disables the shift register as well as putting its MISO output into 3-state mode, disconnecting it from the bus. This allows the main to select the one SPI secondary device that it wants to communicate with.

That's about it. SPI is a very simple communication system, much simpler than I^2C. There is no addressability, the device being communicated with is selected using a hardwired signal line. This means that there is minimal overhead in the data, allowing it to be streamed at high speed. It also means that it's only feasible for use with a small number of devices.

I2C



Inter-Integrated Circuit (*I²C*) is yet another serial communication capability, and is often called Two Wire Interface (*TWI*). It's synchronous like SPI; the main device generates and provides a clock. If you've worked with it you'll have noticed the clock (SCL) and data (SDA) lines. That's where the *two-wire* in the name comes from. Because there is only a single data line, I²C is half duplex, meaning that data can only go in one direction at a time.

One great advantage of I²C is that it is bus based, with each device having an address. The address of a specific device is included in the data in order to talk to that device. This is one of the primary differences from SPI: device addressing is part of the communication, not part of the hardware connection. This makes it far more flexible in terms of supporting many devices without additional hardware. The downside is that there's overhead in the communication itself, meaning that data can't be moved around as quickly. Another downside is that since device addresses are part of the communication, they have to be unique on the bus. If you need to use multiple copies of a device with the same address, you need to use an I²C multiplexer. Some devices offer an alternative: they have pins that are used to set their address. The 23017 (https://adafru.it/sCR) is a prime example of this.

A great many of Adafruit's sensor breakouts (https://adafru.it/Bkr) are I²C based which makes putting together projects quick and easy, with just a few wires.

Wrap up

We've had a look at what's inside the ATMega328 and discussed the various pieces. The '328 is a simple, 8-bit MCU. Even so, it is representative of most MCUs you will find; they all have the same basic components although their size, capabilities, and/or complexity will vary.

Now that we have a general understanding, let's deepen it. How? By designing and building one, of course. We're makers, after-all... let's make an MCU! That will be done in the next tutorial in this series. See you next time.

Series Index

1. Binary, Boolean, and Logic (https://adafru.it/BJk)
2. Some Tools (https://adafru.it/BJl)
3. Combinational Circuits (https://adafru.it/BJm)
4. Sequential Circuits (https://adafru.it/BJn)
5. Memories (https://adafru.it/Bji)
6. An EPROM Emulator (https://adafru.it/BIT)
7. MCUs... how do they work? (https://adafru.it/BJo)