Firmware 101

What is firmware?

Permanent software programmed into a read-only memory

Might be updatable, but might not ever be updated

Firmware typically runs on the ‘bare metal’ of the hardware without an operating system

Goals

Learn the basics of program execution

Get a sense of how data is handled by your CPU

Learn how data gets in and out of your system

Programming basics for dedicated hardware

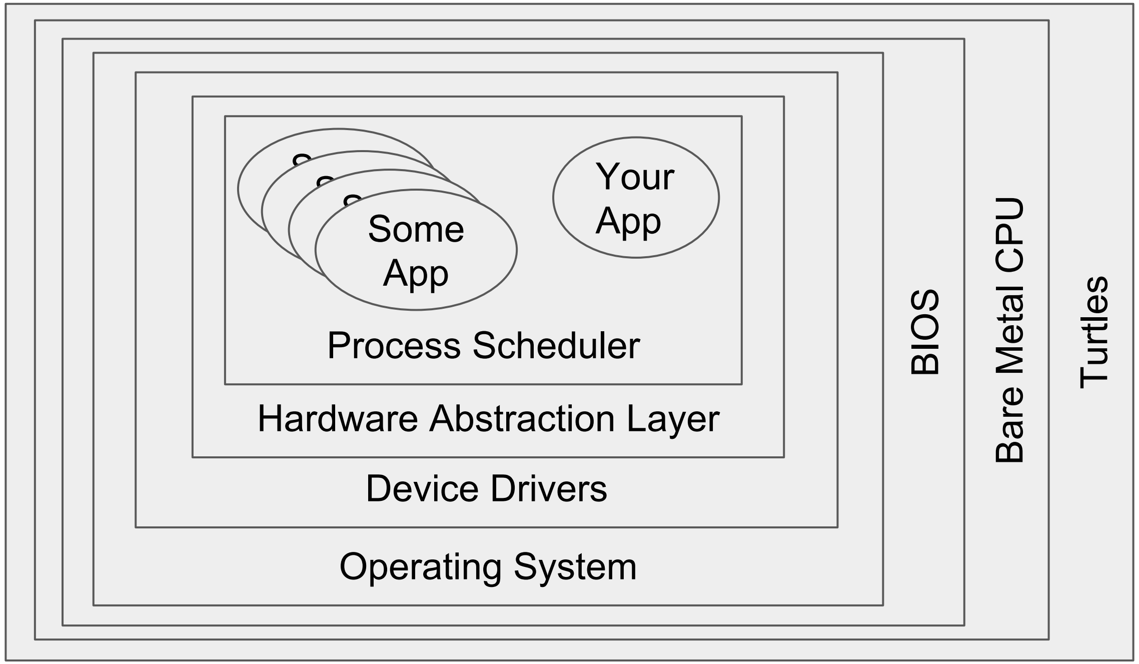
Learn about development environments

Prevent common pitfalls

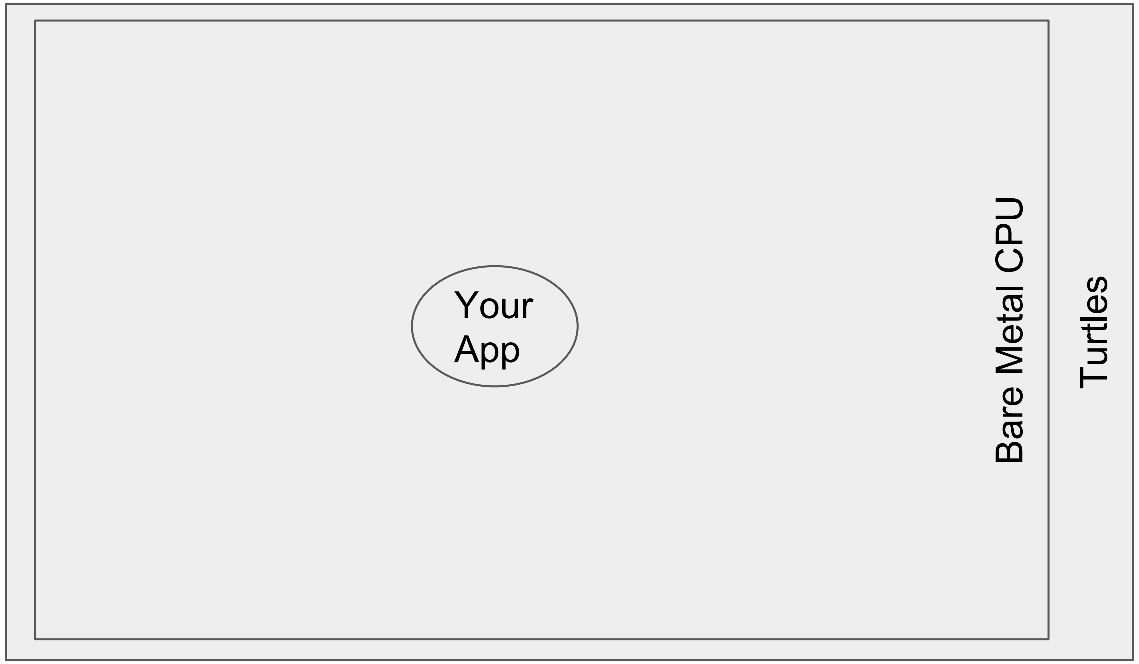
Think up snide remarks on the presentation

Program Execution

Running an App - Generic App on a PC



Running an App - Your App in Firmware



What’s missing?

Your app is responsible for:

- Booting the CPU
- Setting up the hardware
- Managing execution
- Managing memory
- Managing device IO

That’s too much work!

Good news: you don’t need to implement a full operating system

You only need to implement what you want to use

Well, almost...

Firmware: Getting Started

- Study the chip docs for how to boot your CPU
- Study the chip docs for how to setup IO peripherals
- Determine the memory map of your system and write a linker script
- Write a crt0
- Write your app

That’s still too much work!

Good news - most chips have a toolchain or ‘BSP’ (board support package) to do the low level stuff for you.

New list-o-stuff to do:

- Study the chip docs for how to setup and interface your desired I/O
- Download and install a BSP/Toolchain for your chip
- Write your app

Data and Architecture Basics

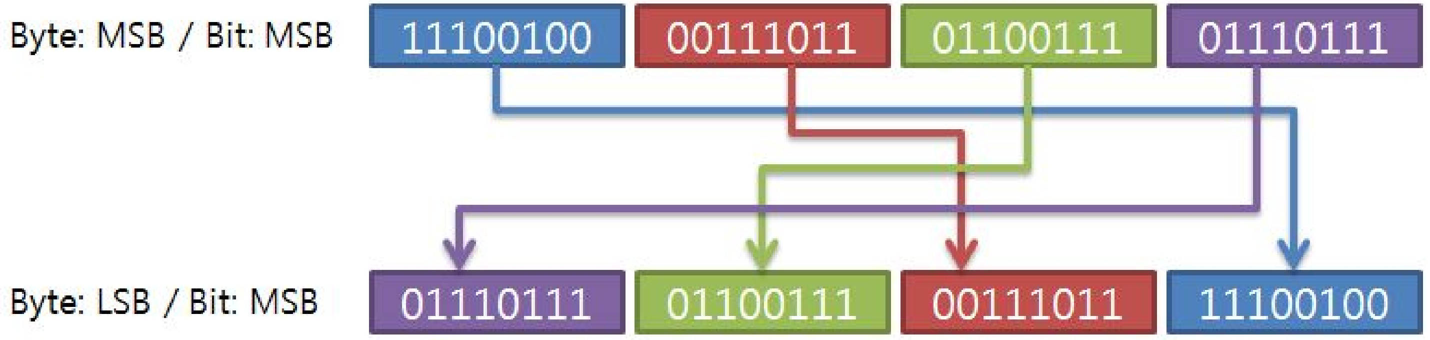
Endianess

Big or Little

- Motorola or Intel

'Network Byte Order' is Big Endian

Bits in bytes are the same, but mult-byte words get swapped



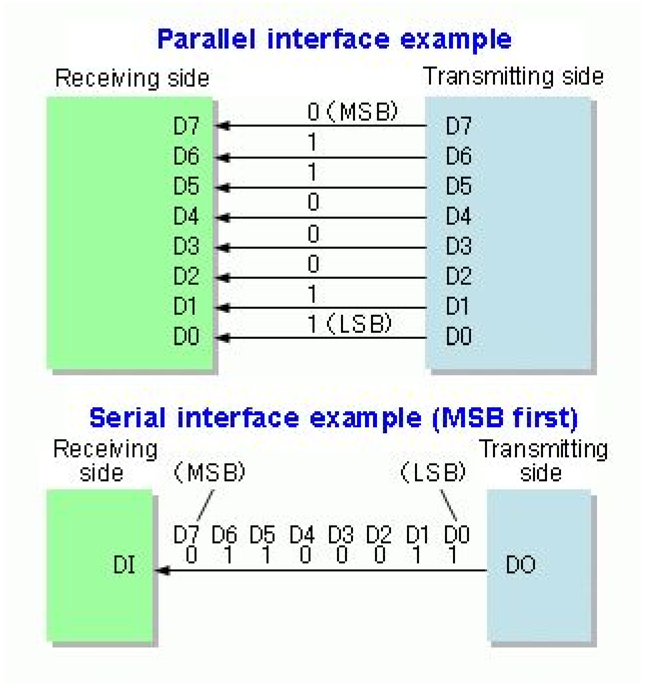
Parallel vs Serial

Serial

- Much less wires
- Slower
- Data usually needs to be buffered

Parallel

- Fast
- Eats up IO



Memory Maps

Different chips are laid out differently

Some areas are read/write, some are read only during program execution

Vectors point to the start of a function

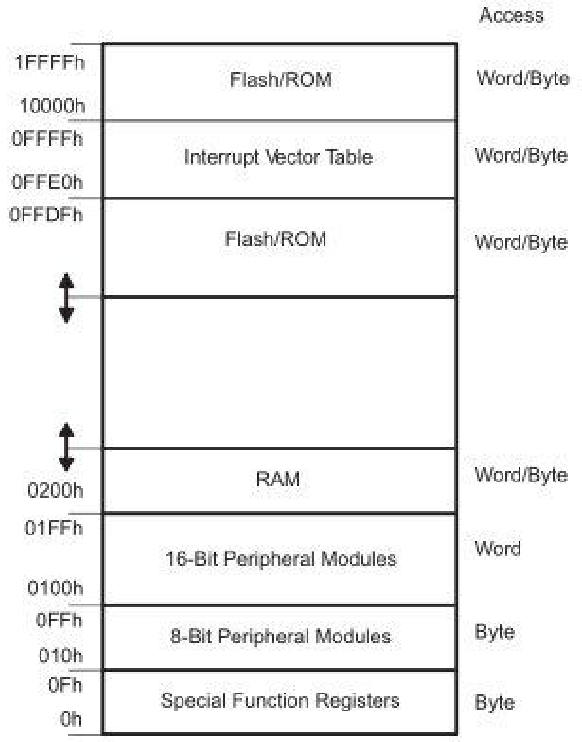


Figure 1-2. Memory Map

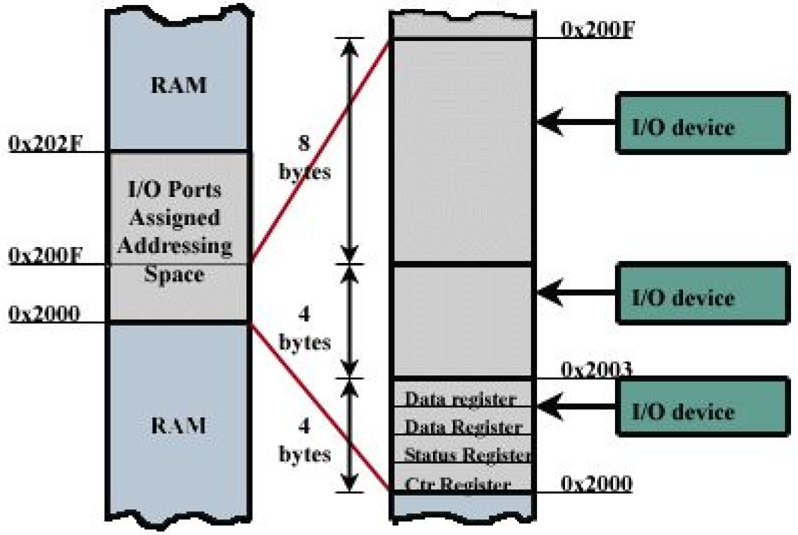
Memory Mapped IO

Sometimes a memory address is really a pointer to some special hardware

Reading/writing to these addresses allow your program to configure the hardware and interact with the outside world

Chip docs typically refer to these addresses as ‘registers’

Your CPU uses registers too

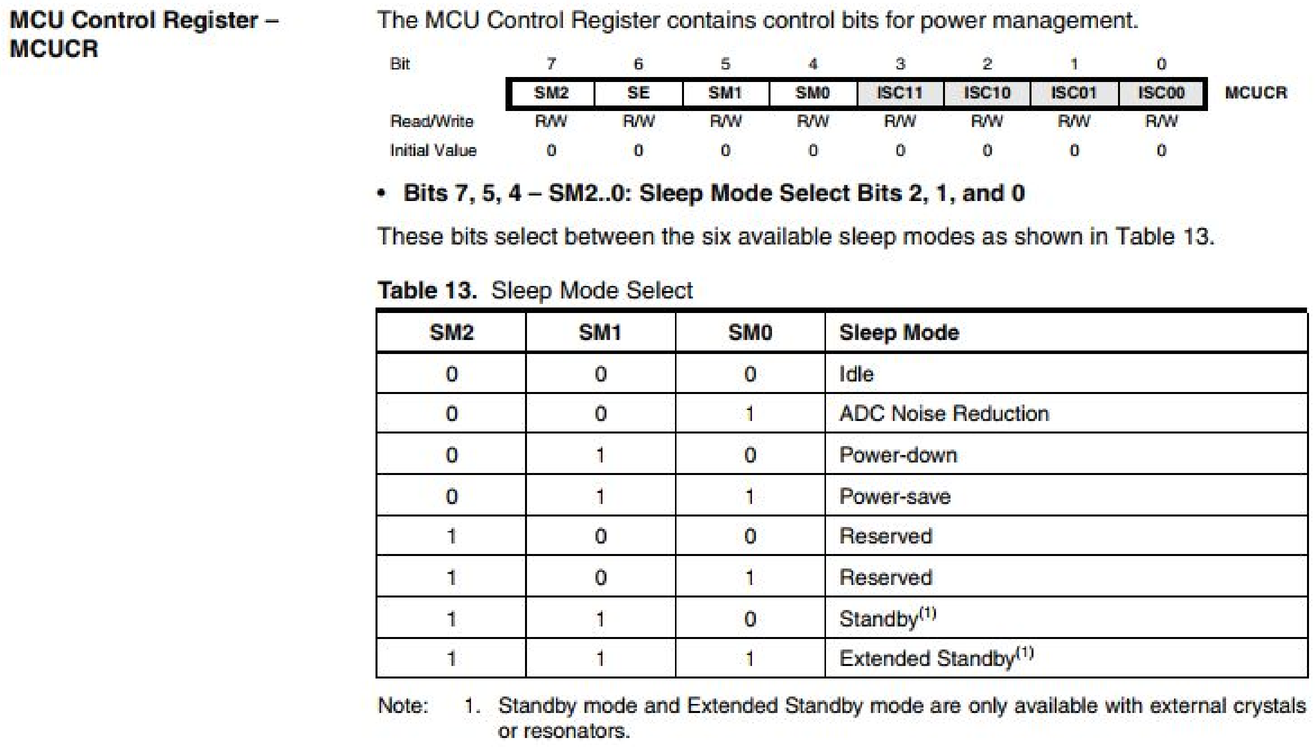


Registers

Bits and bytes

Read? Write? Both?

SM2? SM1? WTF0?



MCU Control Register – MCUCR

The MCU Control Register contains control bits for power management.

Bit	7	6	5	4	3	2	1	0	
	SM2	SE	SM1	SM0	ISC11	ISC10	ISC01	ISC00	MCUCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bits 7, 5, 4 – SM2..0: Sleep Mode Select Bits 2, 1, and 0**

These bits select between the six available sleep modes as shown in Table 13.

Table 13. Sleep Mode Select

SM2	SM1	SM0	Sleep Mode
0	0	0	Idle
0	0	1	ADC Noise Reduction
0	1	0	Power-down
0	1	1	Power-save
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Standby[1]
1	1	1	Extended Standby[1]

Note: 1. Standby mode and Extended Standby mode are only available with external crystals or resonators.

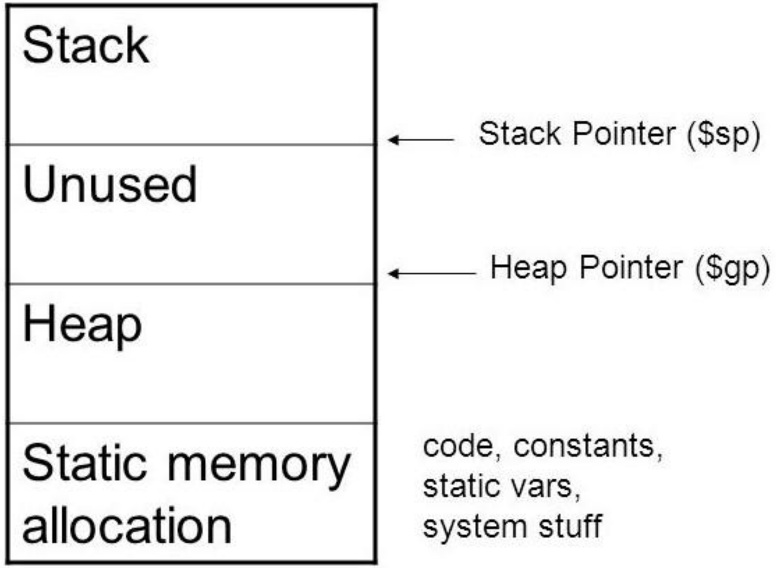
RAM: The Stack and The Heap

Don't cross the pointers
 - True for Ghostbusters and embedded dev

Stack grows in one direction, heap in the other - think about caves

Stack is used to track where we are in execution, and to pass vars into a function

Heap is dynamic memory and pre-allocated static memory

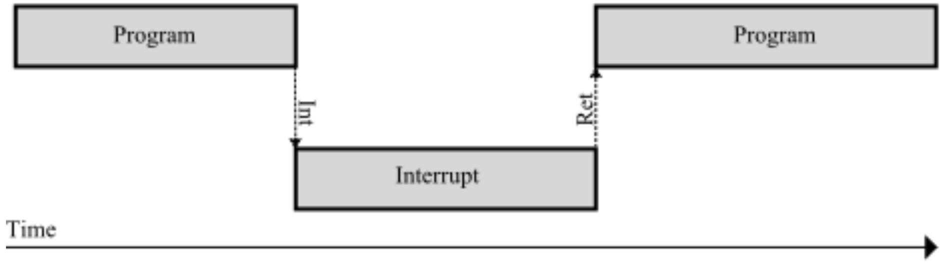


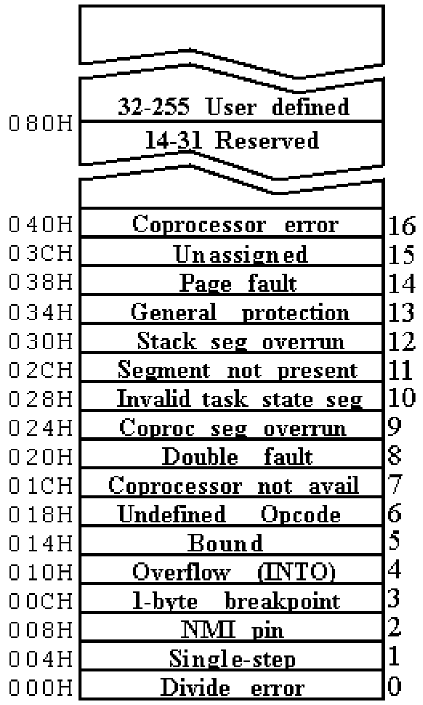
Interrupts

Pauses program execution when some hardware condition is met

CPU jumps to the function pointed to by the interrupt vector table entry

Interrupts need to be handled quickly





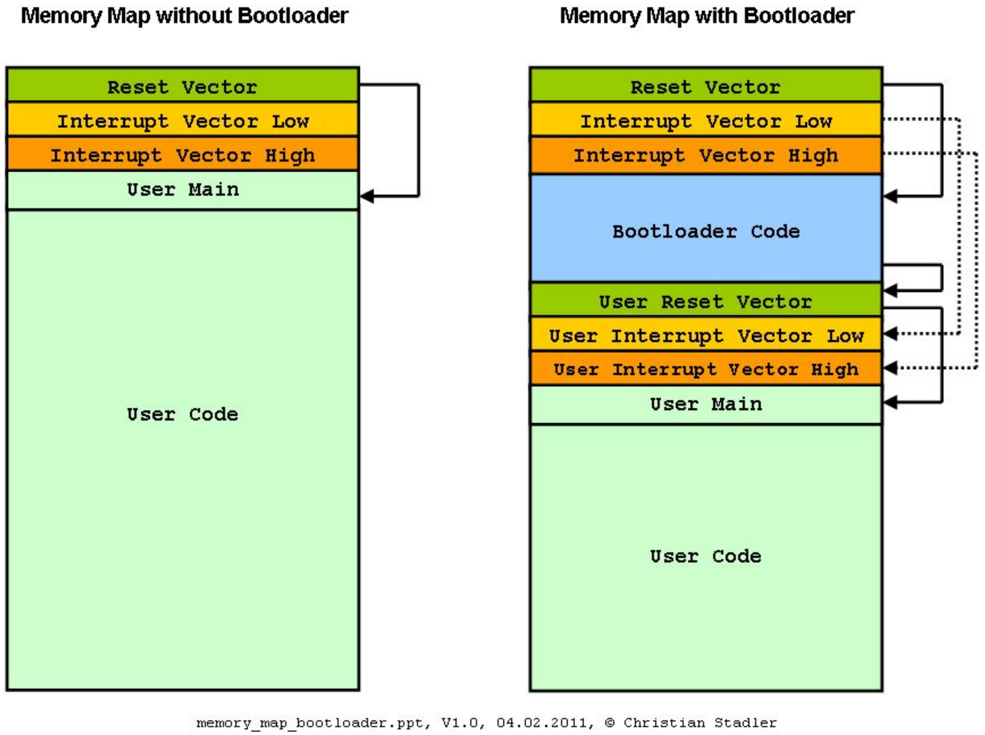
	32-255 User defined	
080H	14-31 Reserved	
040H	Coprocessor error	16
03CH	Unassigned	15
038H	Page fault	14
034H	General protection	13
030H	Stack seg overrun	12
02CH	Segment not present	11
028H	Invalid task state seg	10
024H	Coproc seg overrun	9
020H	Double fault	8
01CH	Coprocessor not avail	7
018H	Undefined Opcode	6
014H	Bound	5
010H	Overflow (INTO)	4
00CH	1-byte breakpoint	3
008H	NMI pin	2
004H	Single-step	1
000H	Divide error	0

Boot!

CPU jumps from the address in the reset vector

CPU doesn't care if data is code or data

CPU = honey badger



Peripherals and IO

GPIO

General Purpose Input/Output

The basic building block of getting signals in and out of your CPU
- All CPUs have some manner of GPIO

'Bit banging'
- Cheap, no special hardware needed to emulate some fancy peripheral
- Takes lots of CPU processing time for anything non-trivial

Operation is typically as simple and reading/writing a register

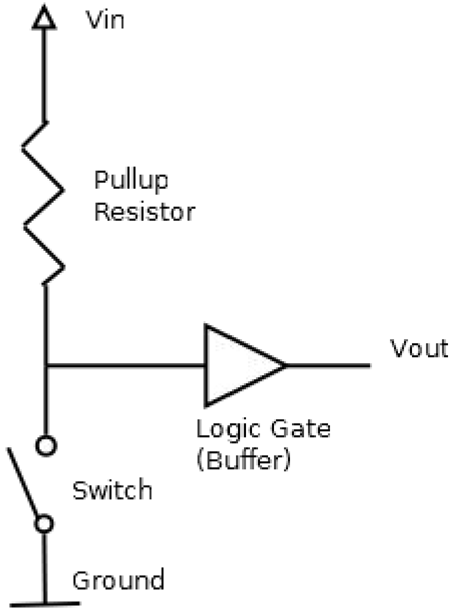
Some chips have configurable pull up or pull down resistors too

Pull up/down resistor?

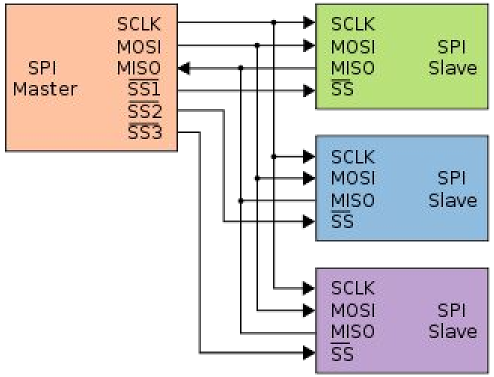
Forces a signal to be either 0v or Vcc

Logic circuits only understand high or low!

‘Floating’ is ‘undefined’ ie ‘probably a bug’



Some other peripherals



SPI

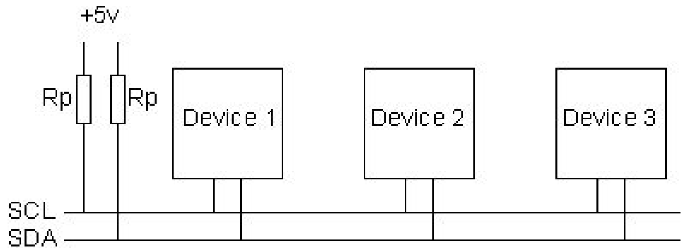
- Serial Peripheral Interface
- One master, multiple slaves with ‘chip selects’

Serial

- RS232/422/485
- Asynchronous, stupid simple - but limited devices on bus

I2C

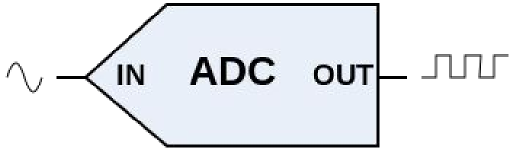
- Inter-integrated Circuit
- No master, devices have addresses, no selects



Still more peripherals

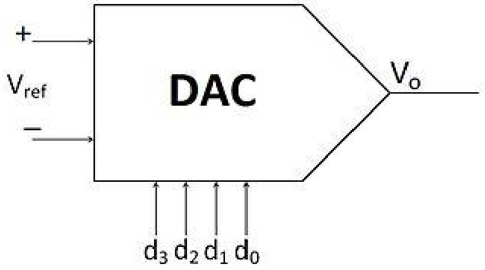
ADC

- Analog to Digital Convertor
- Turn an analog signal into a digital approximation
- Speed, bit width, sample and hold, serial/parallel



DAC

- Digital to Analog Convertor
- ADC, but backwards



CAN, Ethernet, etc, etc...

Programming Basics

Typical program structure

1. Setup hardware
2. Enter a never-ending loop waiting for something interesting to happen
3. Do something amazing when things get interesting

Example:

1. Set GPIOs for buttons to input. Setup serial port
2. Wait until user presses a button
3. Print insults about the user on the serial port
4. Lather, rinse, repeat from step 2

Polling vs Interrupts

A CPU can only do one thing at a time

Polling requires you periodically check for data
- Hard to do other things when you need to go check for something
- Easy to write programs that stall until something happens
- Absolute timing is hard, relative timing is easy

Interrupts
- You have to set them up and write a handler
- Things happen in the background and fire the handler when data is ready
- “Wait until …” code gets a little more interesting
- Great way to set up absolute timers, ie, wall clock

DMA

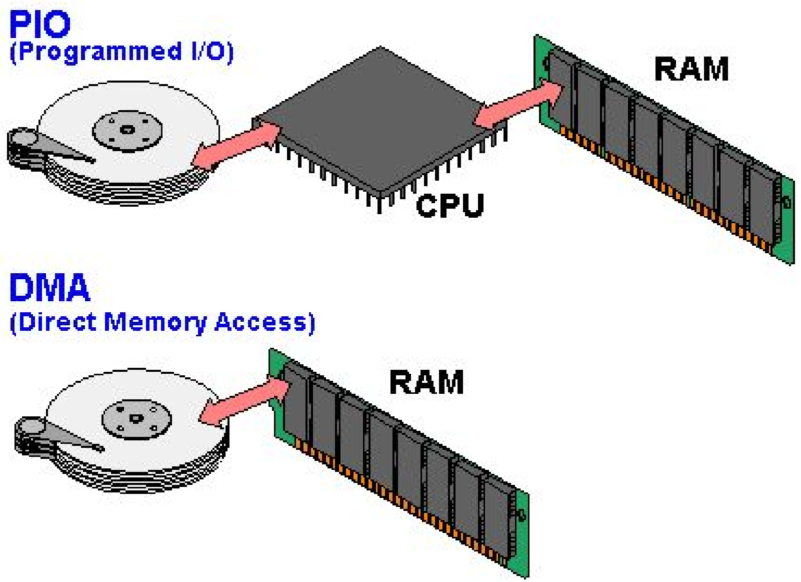
Direct Memory Access

Bypass the CPU for read/write to a device

Much faster than polled IO

Happens in the background

Not all devices support DMA



Development Environments

Programming Language

Can I use Python/Ruby/Node.js/etc?

- Probably not what you want
- Will eat up a lot of space just to get you to ‘hello world’

So, assembly only then?

- Nah, that’s too painful
- Only needed if you need to eek out every last little bit of power or you’re into pain

But I don’t like C!

- Tough cookies?
- Seriously

IDEs and Toolchains

Arduino IDE

- Good for beginners, but very limiting

Chip maker's IDE

- Best chance of support
- Likely need to shell out the bucks if you are doing anything more than blinking an LED

GCC

- There's probably a toolchain for it
- Lots of developers, but there might be ragged edges and pitfalls

RTOS

Real Time Operating System

Super simple task scheduling and memory management kernel

ChibiOS, MQX, VxWorks, Linux (!), others

Abstracts hardware and data passing

- But at the cost of increased complexity and code usage

Makes team development a bit easier

Worth it?

Debugging

Blink an LED

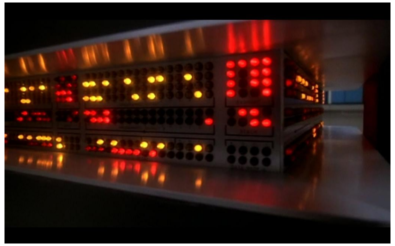
- Simple but not much info
- Use a block of GPIOs with a logic analyzer for signal timing

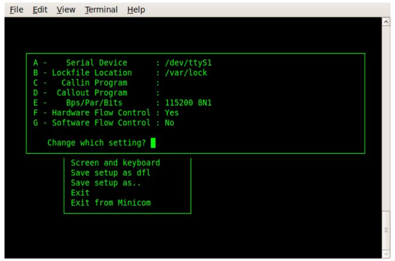
Setup a serial port

- Can eat up a lot of code space
- Not exactly real time, but close

Use JTAG or other chip specific debug interface

- Can set breakpoints, alter memory on the fly
- Pausing for a breakpoint won’t pause external data streams!





```
A -    Serial Device      : /dev/ttyS1
B - Lockfile Location     : /var/lock
C -   Callin Program      :
D -  Callout Program      :
E -    Bps/Par/Bits       : 115200 8N1
F - Hardware Flow Control : Yes
G - Software Flow Control : No

   Change which setting?

        Screen and keyboard
        Save setup as dfl
        Save setup as..
        Exit
        Exit from Minicom
```

Tips and tricks

malloc() and dynamic memory is not your friend

- Static memory allocation protects against memory overruns
- Easier to see if you're going to run out of memory during operation with static analysis

Be super careful about loops and recursion

- Easy to get stuck forever in a loop

Limit pointer redirection

- Remember the CPU will attempt to execute data as though it was instructions if it gets confused

Tips and tricks

Bad

```
if(condition)
  do_task();
```

Good

```
if(condition){
  do_task();
}
```

Code defensively!

- Be verbose, avoid syntax shortcuts
- Clever, compact code is always harder to debug than verbose, obvious code

Computers make very fast, very accurate mistakes

- Try to think like the computer does, not what you want it to do

That thing that will happen like once in a million times will happen every 5 minutes

- CPUs run in the mhz

Read NASA's 'Rules for Developing Safety Critical Code'

- http://pixelscommander.com/wp-content/uploads/2014/12/P10.pdf

Snide Comments?

@hamster