SEICHE 2023
Intermediate Arduino Programming for IoT

Instructor: Paul Frommeyer

www.paulfrommeyer.com

Corporate Sponsor: DXC Technology

REMINDER

YOU MUST HAVE A *WORKING* LED MATRIX DISPLAY TO TAKE THIS CLASS!

If displays are lost or damaged, replacements are $70 with 1-week lead time for replacement

Lesson Plan Overview

Lesson 1: 7Feb23 – Review, Addressing, and Pointers
- Overview of lesson plan
- Class Exercise: Validation of sketch uploading
- Class Exercise: Validation of binary create and upload
- More on the Preprocessor
- Memory Organization and Variables
- Review of addressing and pointers
- Using pointers to variables
- Declaring pointers in function calls
- Call-by-value Function Calls
- Call-by-reference Function Calls

Lesson 2: 14Feb23 – More Gory Details
- Class Exercise: Call-by-reference variables
- Introduction to Complex Data Structures
- Using complex data structures
- Storage declarations and Arduino

Lesson 3: 21Feb23 – Fonts Redux
- MD_Parola library font usage review
- The MD_Parola font format
- Designing your own fonts
- Declaring your own fonts
- Using your own fonts with MD_Parola

Lesson 4: 28Feb23 – Intro to Visual Studio Code
- Introduction to VSC
- https://code.visualstudio.com/docs/introvideos/overview
- VSCode Installation
- VSCode Integration with Arduino and ESP8266

Lesson 5: 7Mar23 – Filesystems
- Introduction to mass storage filesystems
- The SD FAT, FAT16, and FAT32 filesystems
- The exFAT filesystem
- Introduction to SPIFFS
- Class Exercise: Using SPIFFS

Lesson 6: 14Mar23 – WiFi
- Review of WiFi technology (but no gory details)
- Class Exercise: Review of WiFi *access* with WiFi Manager
- Creating access points with ESP8266
- Class Exercise: Creating an access point

Lesson 7: 21Mar23 – Web Technology
- Introduction to HTML
- Web server and client architecture
- Creating web servers with ESP8266
- Class Exercise: A basic web server for time and temperature

Lesson 8: 28Mar23 – Arduino Web Services
- Storing web pages using SPIFFS
- HTML forms
- Form processing with Arduino
- Class Exercise: Displaying text entered on a web page
- Obtaining weather information from the Internet
- Class Exercise: Displaying weather information

Lesson 9: 4Apr23 – IoT Messaging and MQTT
- Introduction to IoT network messaging and MQTT
- Subscribing to an MQTT service
- Publishing to an MQTT service
- Class Exercise: Using MQTT and WiFi

Lesson 10:11Apr23 – Version control and Git
- The need for document version control
- The Git version control system
- Installing Git
- Using Git
- Introduction to Github
- Using Github
- Class Exercise: Signing up for a Github account

Lesson 11:18Apr23 – Using VSC
- VSC plugin review
- Using VSC with Arduino
- Using VSC with Git

Flash Drive Contents Reminder

- When I update parts of your flash drives, you can either *recopy the entire section*, or *just copy the updated part*.
- However, **you will need to explicitly copy any new files so that they are locally accessible on your laptops**
- Copying just *lesson* files will– wait for it!– *copy only the lesson files.*

Make certain you have copied everything in RED!

You should already have previously copied <u>everything else</u>!

FLASH DRIVE

- Archive–BasicProgrammingWith Arduino-Fall2022
- Archive-IntroductionToArduino-Spring2022
- IntermediateProgramming-Software
- IntermediateProgramming-Lessons
- IntermediateProgramming-Documentation

IntermediateProgramming-Software

ESP8266FS-0.5.0.zip

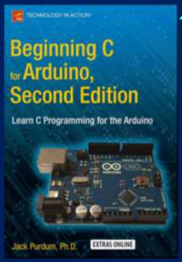
IntermediateProgramming-Lessons

- IntermediateProgramming-Lesson1
- IntermediateProgramming-Lesson2
- IntermediateProgramming-Lesson3
- IntermediateProgramming-Lesson5

IntermediateProgramming-Documentation

- ArduinoReference
  - **Beginning C for Arduino, 2nd Edition**
- BoardsGuides
- D1 Mini Reference
- ElectronicsReference
- IoT Reference
- LED Matrix Display Architecture
- HackSpace Magazine

Can't do reading homework without the files!


Beginning C for Arduino, Second Edition
Learn C Programming for the Arduino
Jack Purdum, Ph.D.

Lesson 5 – Filesystems

Part A

- Introduction to mass storage filesystems
- The Linux filesystem
- SD FAT, FAT16, and FAT32 filesystems
- The exFAT filesystem

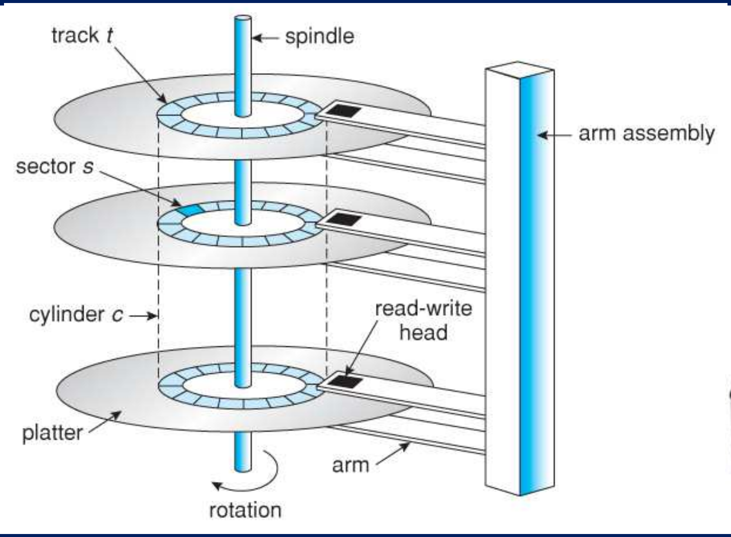
Part B

- ESP8266 flash storage architecture
- SPIFFS
- Installing SPIFFS

Mass Storage Devices

- Mass storage refers to devices used for the storage and retrieval of bulk data by computer sysystems.
- What constitutes “bulk data” depends on the state of the technology for a given era of mass storage
- The first mass storage devices were disk drives, so-called because they used spinning disks of magnetic media with movable read/write heads.
- The first disk drives were the size of washing machines, but were able to store massive amounts of information– 5 *megabytes* for IBM’s first model! [Hey, for the 1960’s, that was a staggeringly huge amount of data!]
- Modern disk drives can store up to *26 terabytes* of information on a single drive

Internals of first commercial disk drive made by IBM

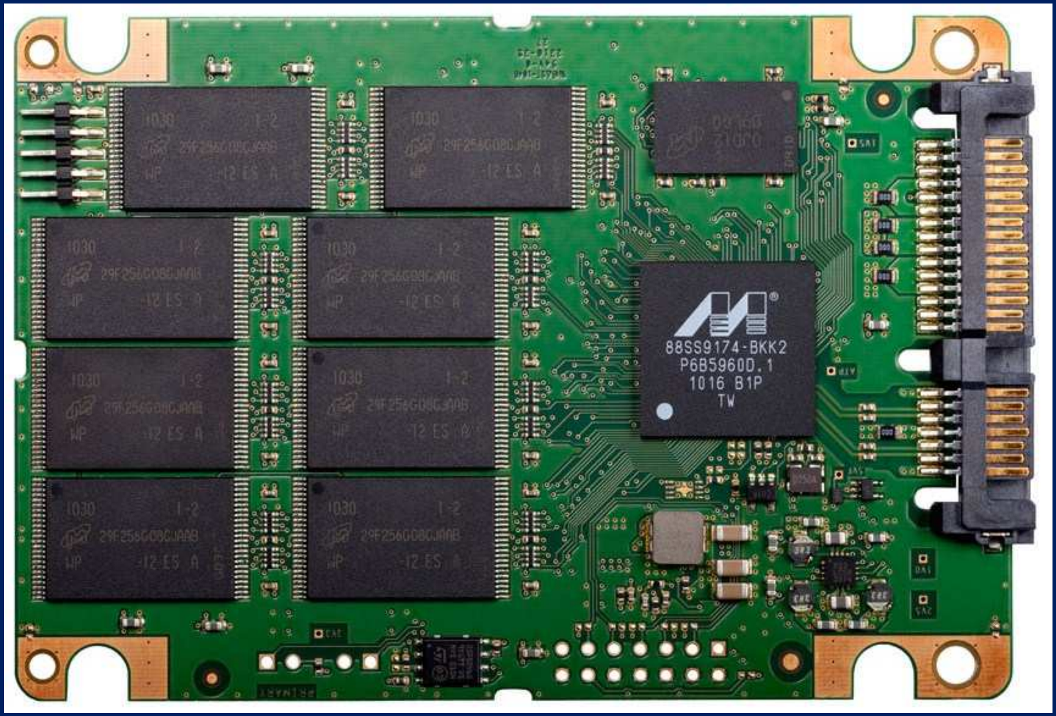


Cutaway view of disk drive mechanical layout. Unchanged since the 1960’s.

Modern 3.5” disk drive internals

Mass Storage Devices (cont.)

As the density and cost of non-volatile NAND gate based silicon memory continues to drop, small flash memory USB jump drives and SD cards have evolved into full-fledged mass storage devices called Silicon Storage Devices (SSDs). SSD’s up to 100 terabytes in size have been constructed.

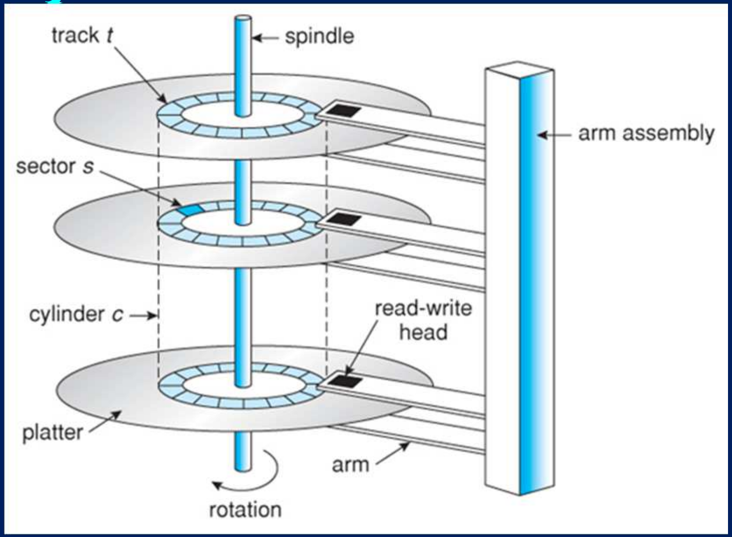


A modern NAND-flash based SSD with SATA connectivity. The chips at the left are the NAND flash chips, the chip at the right is the access controller.

Mass Storage Filesystems

- The original mechanical disk drives were *random access*, which meant that bytes could be written anywhere on the platters. This made them versatile, but also made storing data rather tricky.

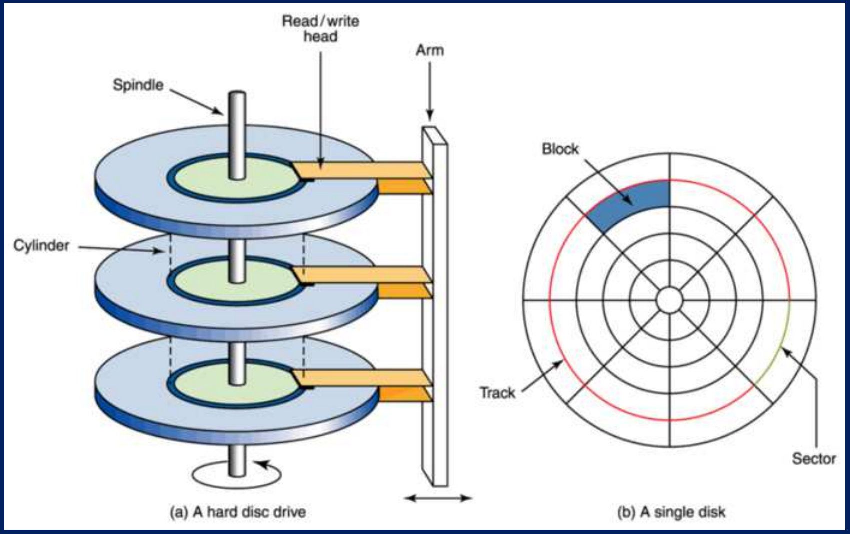
Looking back at the construction of a traditional disk drive, notice that each platter is organized into *tracks*, and each track is broken up into *sectors*. All the sectors of all the tracks at a given position of the arm are called a *cylinder*.



Mass Storage Filesystems

- The original mechanical disk drives were *random access*, which meant that bytes could be written anywhere on the platters. This made them versatile, but also made storing data rather tricky.

Looking back at the construction of a traditional disk drive, notice that each platter is organized into *tracks*, and tracks are divided into *sectors*. The data at a given track and sector is called a *block*. All the blocks at a given position of the arm are called a *cylinder*.



Mass Storage Filesystems

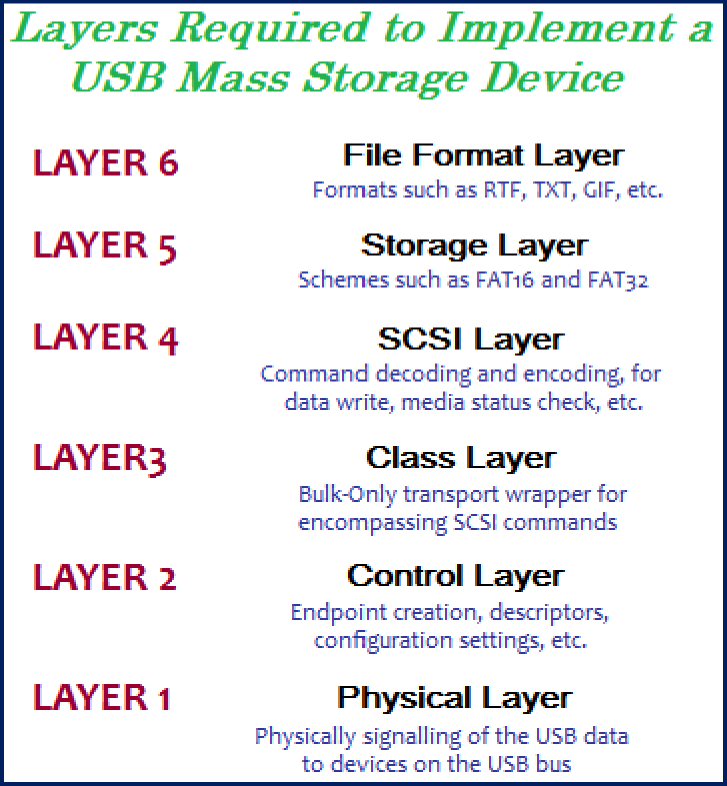
- A whole mess of different schemes were tried for organizing data on a disk drive
  - No sectors; each track was a block of a different size
  - Same number of sectors for each track (highly inefficient, but easy to code for)
  - Different number of sectors per track (preferred; this assures all blocks are the same size)
- But the real challenge was how to organize the data on the disk
  - Present (“expose”) the underlying hardware to the user software, letting the software decide where to put what data
  - Present the storage on the drive as one big “pool” of bytes via a *device driver*, and let the software figure out how to use it
  - Break up the total storage of the drive into smaller chunks, and present those chunks to the user software (preferred)

Mass Storage Filesystems (cont.)

- Once the data was organized, the next question was how to store and retrieve it– because how the data is stored can be different than how it is written or read
- There are three methods for data access the world over:
  - Sequential: Related data (say, a graphic picture) is written and read *sequentially* to the mass storage device, from the first byte to the last
  - Random Access: Data may be retrieved in any order, then “reassembled” by the software. This approach also allows data to be written or read anywhere on the device at any time
  - Indexed: Data is broken up into records, and an index is used to keep track of which records are where on the physical disk
- It is possible to combine these methods, which is what modern drives and operating systems do

Mass Storage Filesystems (cont.)

- The method for “chunking” data on a mass storage device for presentation to a computer is called a *filesystem*.
- There are many, many different types of filesytem. All of them determine how the data on a mass storage device is organized, stored, and retrieved.
- There are now many “layers” of software, firmware, and hardware between a computer and the filesystem on a modern mass storage device.
- This “layer model” allows “abstraction” of the lower layers by the upper layers, so that software need only concern itself with related data (files) and how related data is organized (file formats.)



Layers Required to Implement a USB Mass Storage Device

LAYER 6	**File Format Layer** Formats such as RTF, TXT, GIF, etc.
LAYER 5	**Storage Layer** Schemes such as FAT16 and FAT32
LAYER 4	**SCSI Layer** Command decoding and encoding, for data write, media status check, etc.
LAYER3	**Class Layer** Bulk-Only transport wrapper for encompassing SCSI commands
LAYER 2	**Control Layer** Endpoint creation, descriptors, configuration settings, etc.
LAYER 1	**Physical Layer** Physically signalling of the USB data to devices on the USB bus

Generic Filesystem Architecture

- Related data stored on a device is commonly referred to as a *file*, as an analogy to physical papers stored in a filing cabinet.

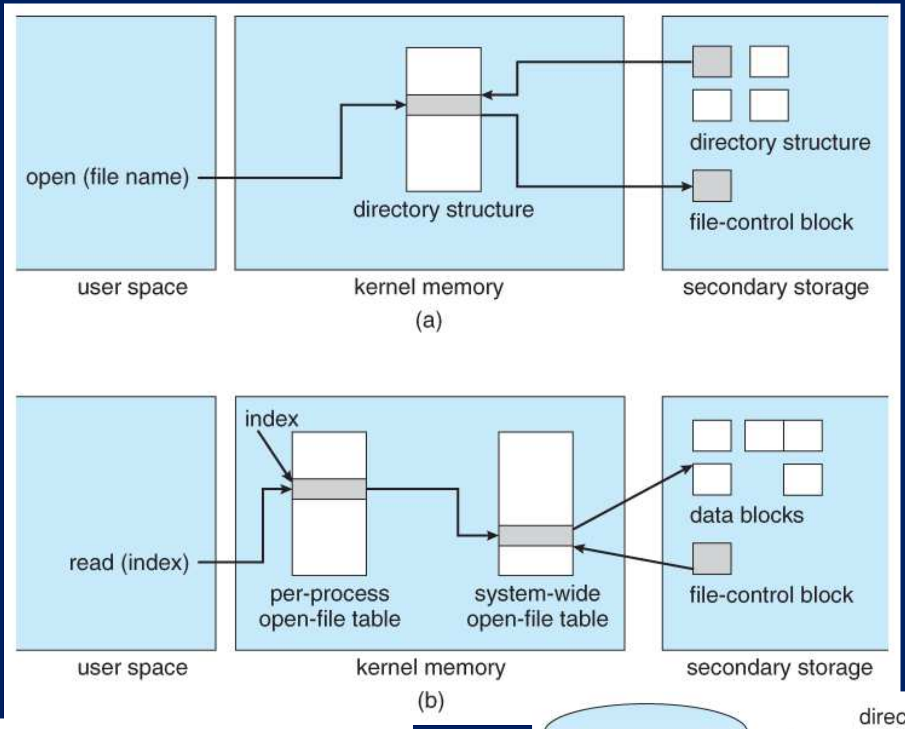
- Files can be stored in one of two ways on a mass storage device:
  – Contiguous: All the data (bytes) of a given file are stored in physically adjacent blocks on the platter(s) of a drive
  – Random: The file data are dispersed across an arbitrary group of blocks on the drive

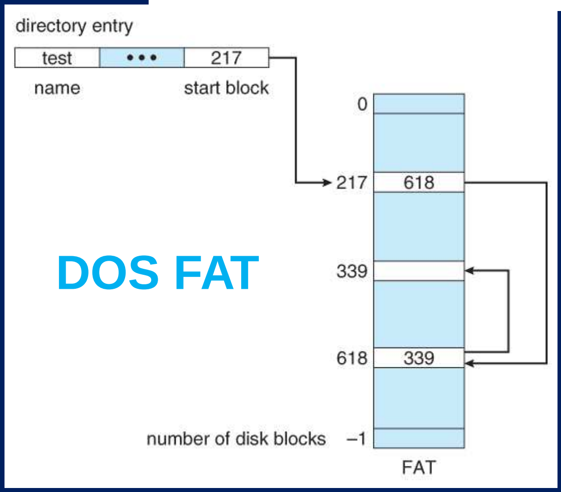
Generic Filesystem Architecture (cont.)

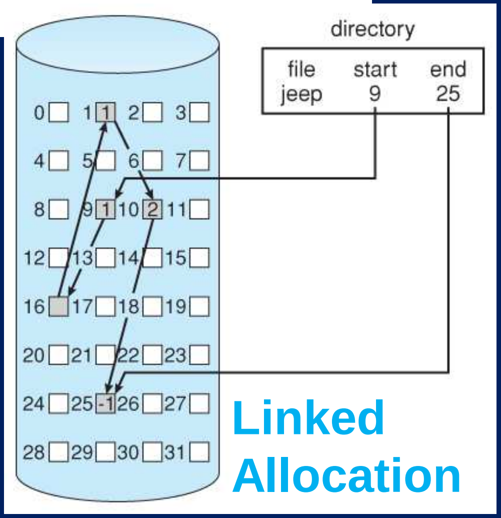
- In either case, to allow the storage of multiple files, an index of some sort must be maintained to keep track of which file is stored where on a drive
- There needs to be an “index to the index”, which is called variously the superblock or volume control block
- The index in turn has various names: file index, catalog, or, more commonly, *file allocation table (FAT)*
- The FAT contains entries for each file called *inodes* or *file control blocks*. The FCB’s contain:
  - The file *name*
  - File metadata: creation time, access time, and (optionally) permissions, owner, file type and access method for that file
  - The size of the file *in blocks*
  - Either:
    - A list of the blocks the file is using on the drive, that is, a map of which blocks make up the file and the order those blocks need to be accessed to reassemble the file (e.g. DOS FAT)
    - Just the first block the file is using, if the file is comprised of linked blocks (e.g. HFS)

Generic Filesystem Architecture (cont.)

- Modern operating systems abstract the details of the filesystem from user programs
- This abstraction is performed by the device drivers in the operating system kernel (the part of the OS that is always in memory)
- Typically, the directory structure of a given filesystem is cached into memory for easy access
- When a process (program) opens a file, a *filehandle* is created which is a data structure that allows the program to communicate with the OS to read and write to the particular file that was opened.
- Modern OS's present most files to programs as a byte stream, leaving it up to the program to determine any additional formatting of the byte stream that may be needed.
- Older OS's could perform some pre-formatting of the byte-stream

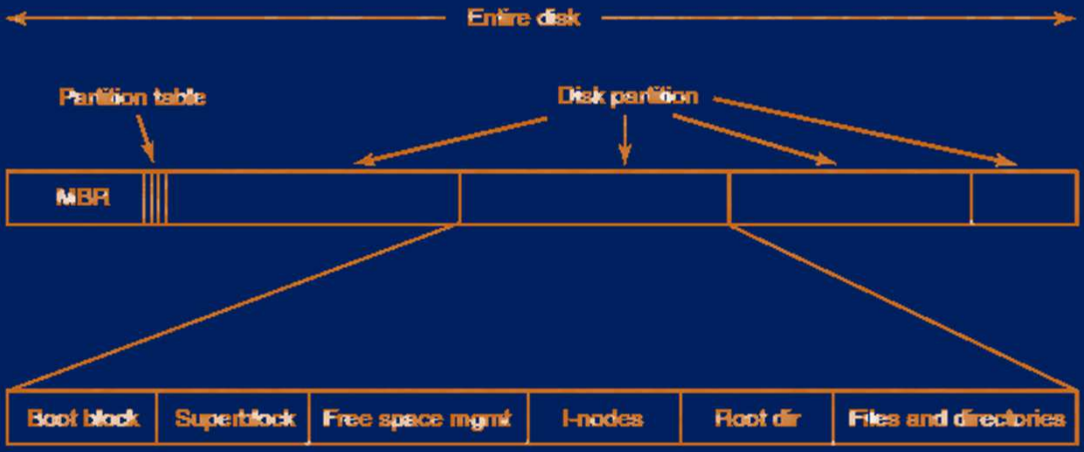


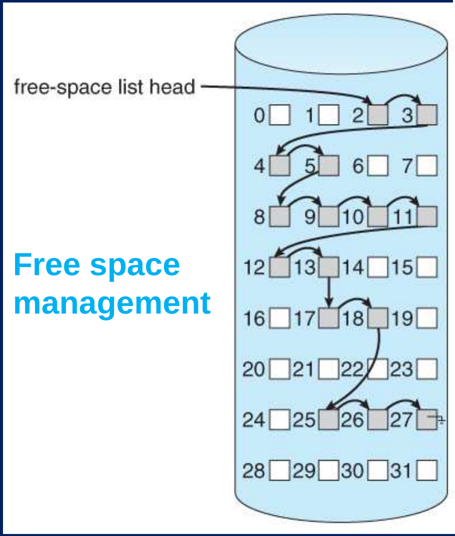


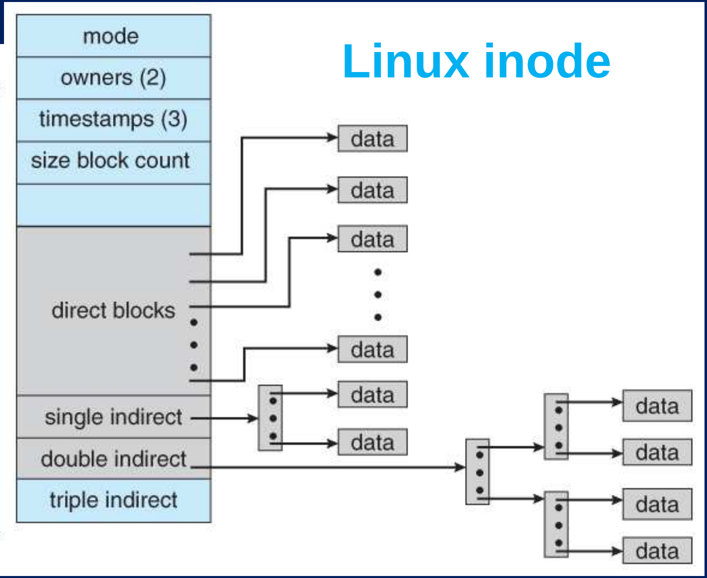


Linux Filesystem Architecture

- Linux uses a very sophisticated filesystem called EXT4 (the fourth version since the original ext or ext1)
- EXT4 uses *combined indexing*, which permits multi-level linking, allowing very large files
- Multiple *partitions*, each containing *it's own* ext4 filesystem, are possible
- A *Master Boot Record* (MBR) is used to map the partition bootblock where OS startup code is located
- A *superblock* provides a map of all the file *inodes*
- Each inode is organized as shown at right, providing a *linked list* of the blocks in a given file
- Free space is also a linked list of unused blocks





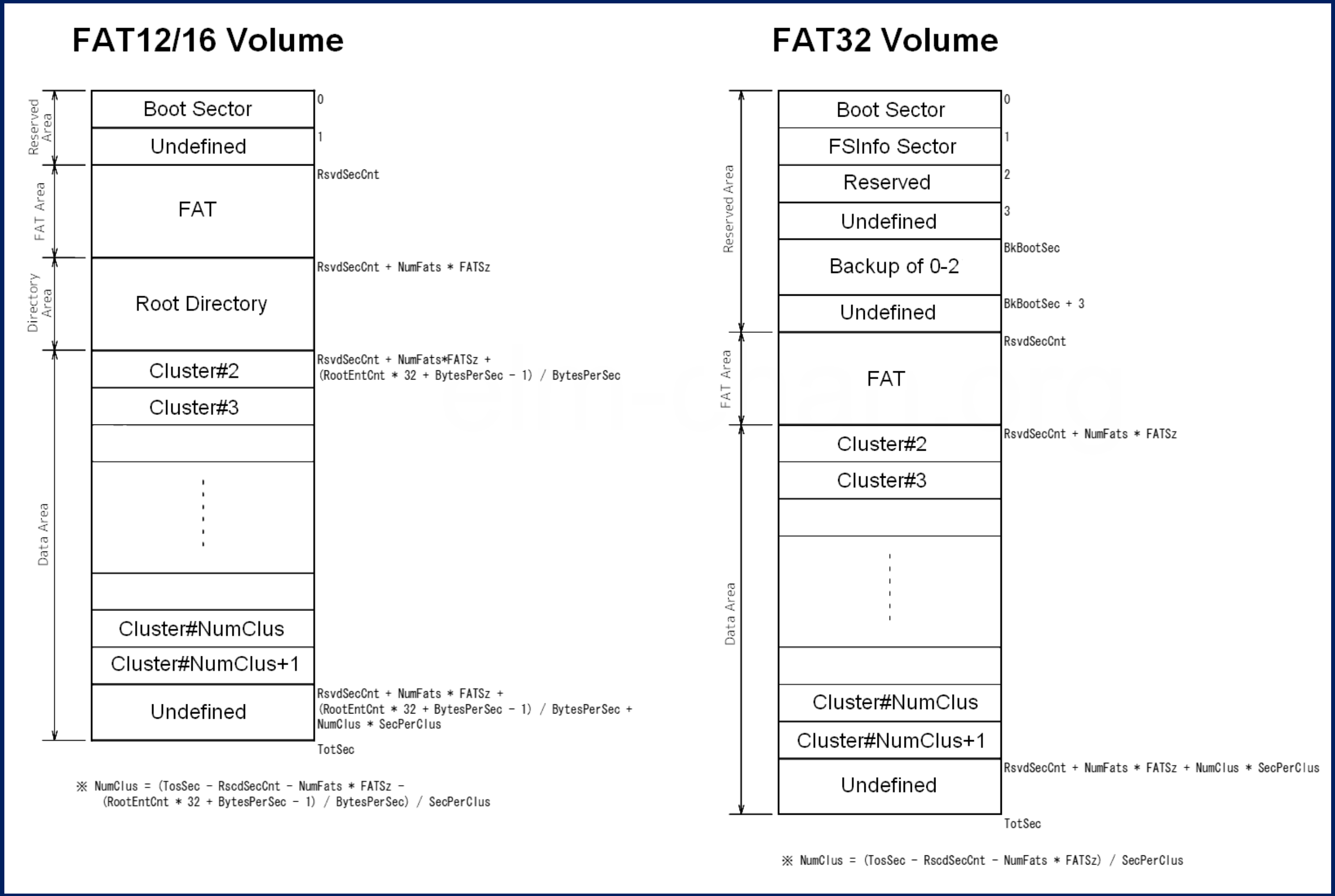


The FAT Filesystem

- The FAT file system originated around 1980 and is the filesystem that was first supported by MS-DOS. It was originally developped a simple filesystem suitable for floppy disk less than 500k bytes in size. Over the time, its specs has been expanded to support laeger media as increasing its capacity. FAT is the abbreviation of File Allocation Table, which is the array to manage allocation of data area and the name of the file system itself. Currently there are three FAT sub-types, FAT12, FAT16 and FAT32. These are developped in order of the number and completely backward compatible with older one. (FAT16 always include FAT12, FAT32 includes all FAT types)EXT4 uses *combined indexing*, which permits multi-level linking, allowing very large files
- FAT *sectors* are 512 bytes in size; the term sector means the same thing as *block*
- FAT uses *logical volumes* contained inside the physical storage volume (similar to multiple partitions on Linux)
- A FAT logical volume consists of three or four areas, each of them consists of one or more sectors and located on the volume in order of as follows:
    1. Reserved area (volume configuration data)
    2. FAT area (allocation table for data area)
    3. Root directory area (not present on FAT32 volume)
    4. Data area (contents of file and directory)
- The most important data structure in the FAT volume is BPB (BIOS Parameter Block), where the configuration parameters of the FAT volume are stored. The BPB is placed in the boot sector. The boot sector is often referred to as VBR (Volume Boot Record) or PBR (Private Boot Record), but it is simply the first sector of the reserved area, the first sector of the volume.
- The another important area is FAT. What this structure does is define a linked list of the extents (cluster chain) of a file. Note that both directroy and file is contained in the file and nothing different on the FAT. The directory is really a file with a special attribute that indicates its content is a directory table.

FAT Filesystem Types and SFNs

- **The FAT type (FAT12, FAT16 or FAT32) is determined by the count of clusters on the volume and NOTHING ELSE**
- ***A volume with count of clusters <=4085 is FAT12.***
- ***A volume with count of clusters >=4086 and <=65525 is FAT16.***
- ***A volume with count of clusters >=65526 is FAT32.***


FAT12/16 Volume
Reserved Area
FAT Area
Directory Area
Data Area
Boot Sector
Undefined
FAT
Root Directory
Cluster#2
Cluster#3
Cluster#NumClus
Cluster#NumClus+1
Undefined
0
1
RsvdSecCnt
RsvdSecCnt + NumFats * FATSz
RsvdSecCnt + NumFats*FATSz + (RootEntCnt * 32 + BytesPerSec − 1) / BytesPerSec
RsvdSecCnt + NumFats * FATSz + (RootEntCnt * 32 + BytesPerSec − 1) / BytesPerSec + NumClus * SecPerClus
TotSec
※ NumClus = (TosSec − RscdSecCnt − NumFats * FATSz − (RootEntCnt * 32 + BytesPerSec − 1) / BytesPerSec) / SecPerClus
FAT32 Volume
Reserved Area
FAT Area
Data Area
Boot Sector
FSInfo Sector
Reserved
Undefined
Backup of 0-2
Undefined
FAT
Cluster#2
Cluster#3
Cluster#NumClus
Cluster#NumClus+1
Undefined
0
1
2
3
BkBootSec
BkBootSec + 3
RsvdSecCnt
RsvdSecCnt + NumFats * FATSz
RsvdSecCnt + NumFats * FATSz + NumClus * SecPerClus
TotSec
※ NumClus = (TosSec − RscdSecCnt − NumFats * FATSz) / SecPerClus

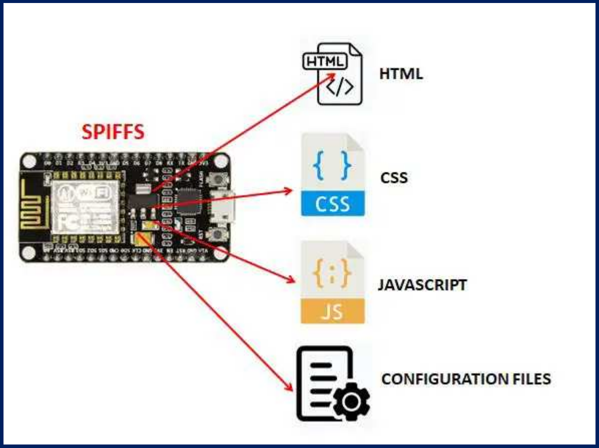
- ***Short file name (SFN) is the basic feature of FAT volume. The directory is really a file with a special attribute. It contains table of directory entries that contain meta data of the files on the volume. Size of a directory entry is 32 byte long and it corresponds with a file or directory on the volume. Maximum size of a directory is 2 MB (65536 entries).***
- The SFN file name is stored in 8-byte body + 3-byte extension. The dot in the file name to separate body and extension is removed on the directory entry.

FAT Long File Names (LFN)

- When add the long file name (LFN) as a new feature to the FAT filesystem, a backward compatibility with the existing systems is required. Following are concrete examples required.*A volume with count of clusters <=4085 is FAT12.*
  - *The existence of LFN needs to be invisible on the existing systems, especially any file API on the MS-DOS and Windows.*
  - *LFN needs to be located physically near the direcroty entry of the corresponding file to prevent bad effect to the performance.*
  - *If disk utility found the LFN information recorded somewhere on the FAT volume, the filesystem needs to keep sanity and be not affected.*
- SFN, often called the 8.3 format name, is a traditional style file name originally used on the MS-DOS in format of body (1-8 character) plus optional extension (1-3 characters). These two parts are separated with a dot (.). The allowable charactes for the SFN are ASCII alphanumerics, some ASCII marks ($%'-_@~`!(){}^#&) and extended characters (\x80 - \xFF).
- SFN is stored in the SFN entry in OEM code set (used on MS-DOS) depends on the system locale. Low-case character in the file name is converted to up-case and then stored and matched, so that the case information of SFN is lost.
- Length of LFN can be up to 255 characters. The allowable characters for the LFN are white space and some ASCII marks (+,;=[]) in addition to the SFN characters. Dots can be embedded anywhere in the file name except the trailing dots and spaces are treated as end of the name and truncated off on the file API. Preceding spaces and dots are valid, but some user interface, such as Windows common dialog, rejects such file name.
- LFN is stored in the LFN entry without up-case conversion. Character code used by LFN is in Unicode.
- Because different character codes are used by SFN (OEM code set) and LFN (Unicode), generic implementation needs to convert those codes. This is not the matter on the OEM code is single byte code, however, when the OEM code page is in double byte character set (DBCS), a huge (several hundreds KB) conversion table is needed, so that it is difficult to implement LFN in the small embedded systems with a limited memory.

ESP8266 Flash Storage Architecture

- The SPI accessed flash storage on the ESP8266 where Arduino binary code is stored ***is not erased when a new sketch is uploaded!***
- This means that the on-board flash can be used for the storage of persistent data– configuration files, web pages, what-have-you.
- The filesystem format for storing this data is called SPIFFS – SPI Flash File System.
- SPIFFS lets you access the flash chip memory like you would do in a normal filesystem in your computer, but simpler and more limited. You can read, write, close, and delete files. SPIFFS doesn’t support directories, so everything is saved on a flat structure.
- Using SPIFFS, you can:
  - Create configuration files with settings;
  - Save data permanently;
  - Create files to save small amounts of data instead of using a microSD card;
  - Save HTML and CSS files to build a web server;
  - Save images, figures and icons;
  - And much more.
- Before you can begin using SPIFFS, we will need to install a file upload plugin for the Arduino IDE

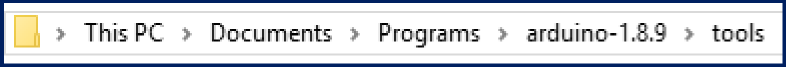


Arduino File Uploader Installation

https://randomnerdtutorials.com/install-esp8266-filesystem-uploader-arduino-ide/

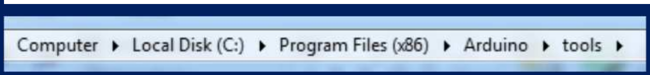
- Go to the Tools directory of the Arduino IDE. You will do this in your *file explorer*, not the IDE!
- Windows 7 Path (see below for Win 10):
  `Computer\C:\Program Files (x86)\Arduino\tools`
- Linux Path:
  `/usr/local/src/arduino-1.8.19/tools`
- Copy the following *file* from your Lesson 5 directory or your Software directory to the arduino/tools directory:
  ESP8266FS-0.5.0.zip
- Unzip the file into the tools directory; you should get the following:
  `<home_dir>/Arduino-<version>/tools/ESP8266FS/tool/esp8266fs.jar`
- Start (or restart if you got ahead) your Arduino IDE
- To check if the plugin was successfully installed, open your Arduino IDE and select your ESP8266 board. In the Tools menu check that you have the option “ESP8266 Sketch Data Upload“.

Win10



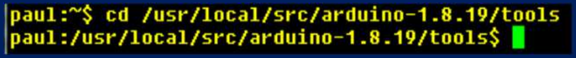
This PC › Documents › Programs › arduino-1.8.9 › tools

Win7



Computer ▸ Local Disk (C:) ▸ Program Files (x86) ▸ Arduino ▸ tools ▸

Kubuntu



```
paul:~$ cd /usr/local/src/arduino-1.8.19/tools
paul:/usr/local/src/arduino-1.8.19/tools$
```

Formal End of Lesson 5

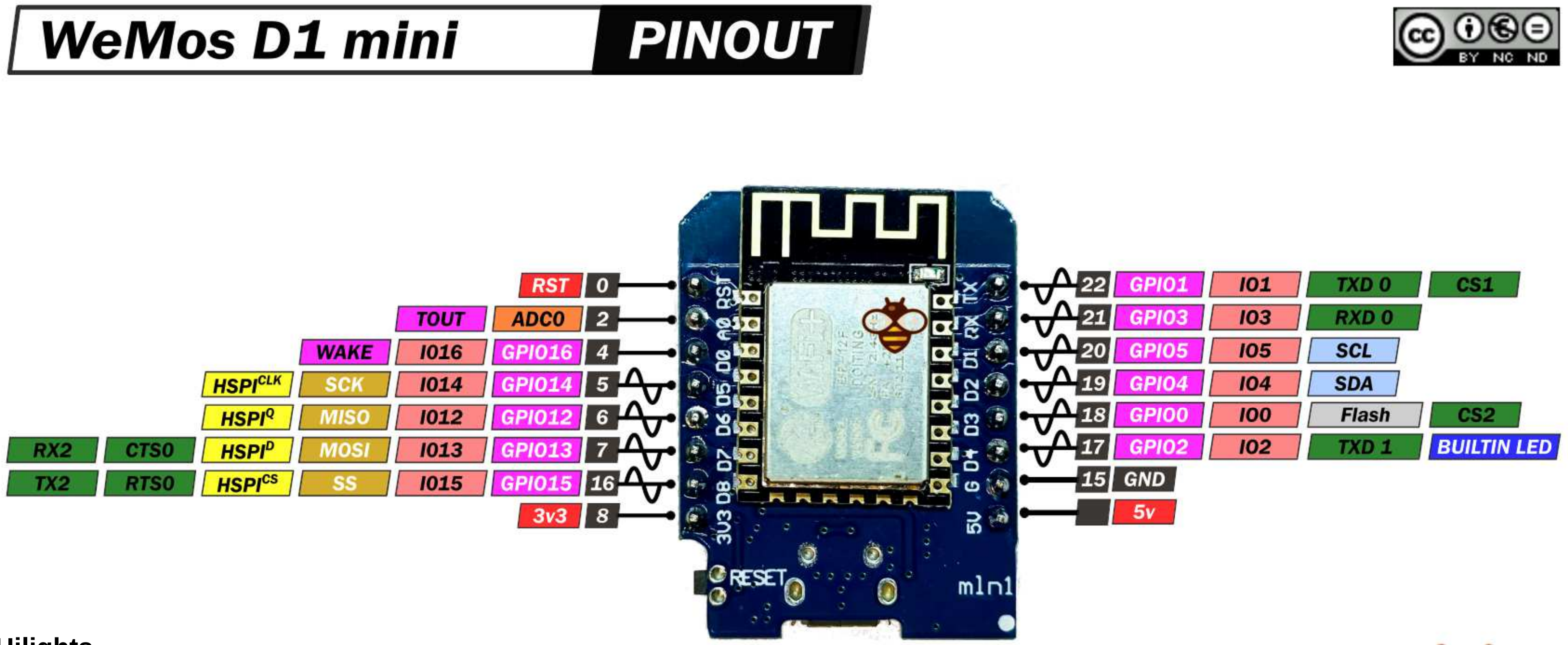
HOMEWORK

- **Review SPIFFS usage per https://randomnerdtutorials.com/install-esp8266-filesystem-uploader-arduino-ide/**

In next week's exciting episode

- Review of WiFi technology (but no gory details)
- Class Exercise: Review of WiFi access with WiFi Manager
- Uploading a web page using SPIFFS
- Class Exercise: Create a basic web server

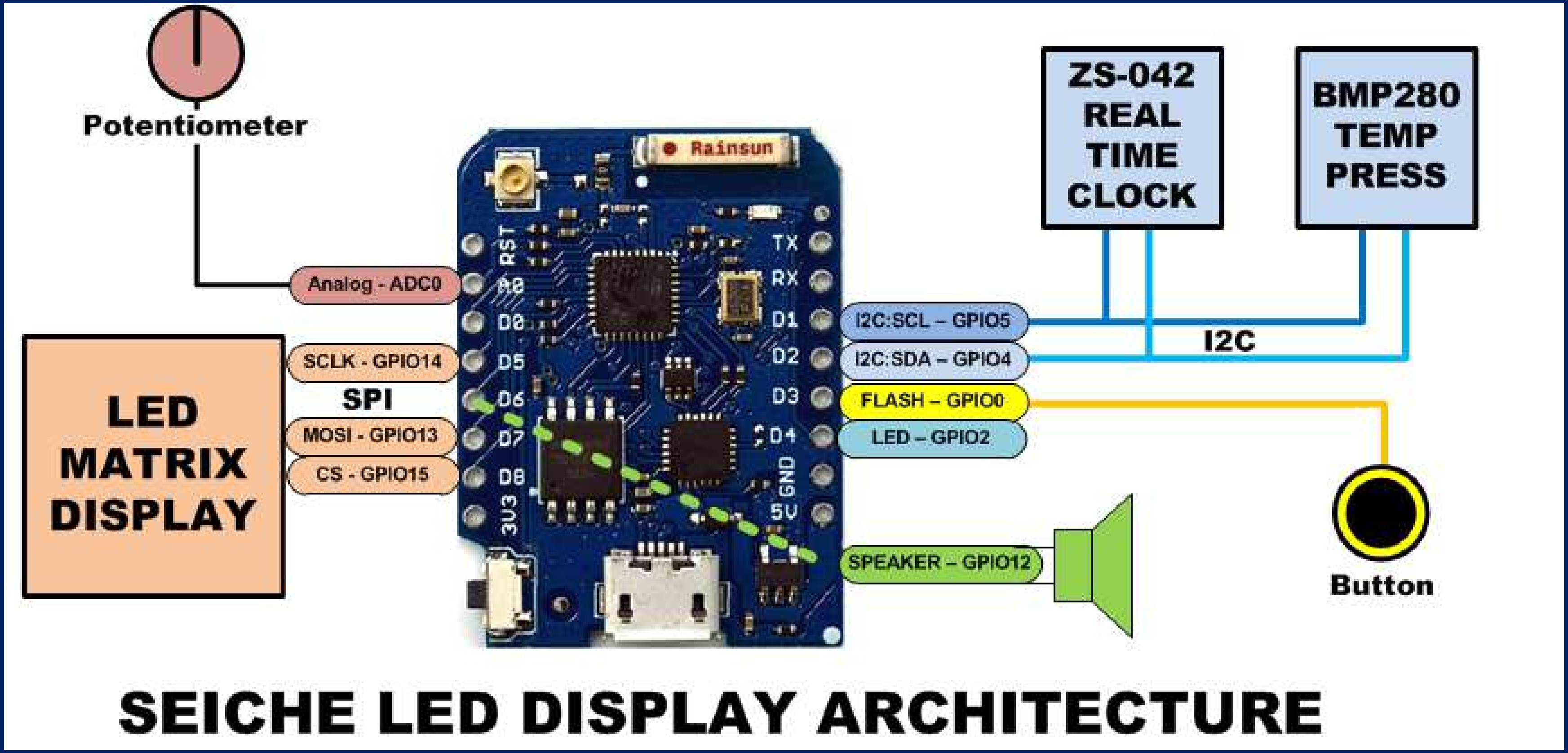
Our Microcontroller: The WeMos D1 Mini



Hilights

- 11 digital IO: all are interrupt and pwm capable (except D0/GPIO16)
- 1 analog input (3.2V max input): A0/ADC0
- Micro USB <u>or</u> Type-C USB Port (clones usually have micro USB)
- Two SPI interfaces (one is used for on-board flash memory), one I2C interface, two serial ports
- Built-in WiFi (client or standalone access point modes) and Bluetooth
- Compatible with MicroPython, Arduino, NodeMCU
- Uses the CH340 USB-to-serial driver (installation usually needed on Windows)
- Extremely low cost (approx $3.00 US on Amazon; one of the two least expensive components in your kits)

SEICHE LED Display Architecture



- Red MAX7219 8x32 LED matrix display (SPI)
- ZS-042 real-time clock module (I2C)
- BMP280 Temperature and Pressure Sensor (I2C)
- Piezoelectric speaker (PWM)
- 10KΩ Potentiometer (Analog-to-Digital Converter)
- Button (Pullup and interrupt)

`sprintf()` Function Syntax

- `sprintf()`'s formal syntax is:
  `sprintf(`char ***`buffer`**, const char ***`format`**, *variable list***`)`**
- **`buffer`** is an array of type char (the parameter when passed can also be a pointer [address] for such an array) which will contain the formatted output of the function
- **`format`** is an unmodifiable (constant) array of char containing the format descriptors for all variables subsequently passed to the function, which is to say, it's the *format string*.
- The variable list are just the comma separated variables that are to be formatted
- All variables passed in a single call to `sprintf()` are combined into a single output string
- `sprintf()` automagically puts a terminating NULL (\0) at the end of the ASCII output

`sprintf()` Format Strings

- `sprintf()` format strings contain conversion specifiers that specify how each individual variable is to be converted
- `sprintf()` conversion specifiers have the following general syntax (all begin with a percent-sign):

%[flags][minimum field width][.][precision][length][conversion character]

  - % - special token that indicates the start of a conversion specifier
  - Flags – these modify the behavior of the specification
  - Minumum field width – as it says on the tin; this the minimum number of characters to be converted
  - . – The period is a separator between field width and precision
  - Precision – means one of the following depending on the variable type and conversion specifier
    - The maximum number of characters to be generated from a string
    - The number of digits after the decimal point for type float conversions (e, E, or f)
    - The zero-filled minimum number of digits for an integer
  - Conversion character – A single character which determines the output type of the conversion specifier; a single character that specifies the type of output format for the corresponding data or variable

`sprintf()` Conversion Specifiers

Below are the general conversion specifiers and what they do.

Specifier	What it does
d, i	int - integer; signed decimal notation
o	int – unsigned octal (no leading zero)
x, X	int – unsigned hexadecimal, no leading 0x
u	int – unsigned decimal
c	int – single character, after conversion to unsigned char
s	char * - characters from string are printed until \0 (NULL) or *precision* is reached
f	double – decimal notation of form [-]mmm.ddd where number of decimals is specified by precision; precision of zero (0) suppresses the decimals altogether
e, E	double - exp notation; default precision of 6, 0 suppresses
g, G	double – Use %f for <10^4 or %e for >10^4
p	void * - print output as a pointer, platform dependent
n	Number of characters generated so far; goes into output
%	No conversion, put a % percent sign in the output

`sprintf()` Conversion Specifiers

Below are the flags and what they do.

Flag	What it does
-	Left justification
+	Always print number with a sign
spc (space)	Prefix a space if first character is not a sign
0 (zero)	Zero fill left for numeric conversions
#	Alternate output form depending on conversion character o – first digit will be zero x or X – 0x or 0X (respectively) prefixed to non-zero results e, E, f, g and G – Output will always have a decimal point g and G – trailing zeroes will never be removed