SEICHE 2022
Basic Arduino Programming

Instructor: Paul Frommeyer

www.paulfrommeyer.com

Corporate Sponsor: DXC Technology

DXC TECHNOLOGY

Lesson Plan Overview

Lesson 1 – Intro and Setup
[may require 2 classes]

- Introduction to class format
- Overview of lesson plan
- Presentation format (monitor, camera, screen, whiteboard)
- Review of microcontrollers and types of boards
- SEICHE LED display architecture
  - ESP8266 pinout
  - High level architecture

Lesson 2 – Laptop operation review – Windows and Linux

- Inventory of USB drives
- Installation of Arduino IDE software
- Installation of CH340/ESP8266 serial port drivers (Windows only)
- Control panel/settings location
- Home directories and folder hierarchy
- Arduino file locations
- Search functions
- (Windows) Device Manager
- (Linux) Konsole
- Copying flash drive contents [critical]
- Open questions and issues
- **IDE essentials**
- Starting the Arduino IDE
- Basic Arduino sketch (program) structure
- Loading example sketches
- Loading and configuring new boards
- Connecting boards
- Identifying the microcontroller serial port
  - Linux
  - Windows

Lesson 3 – Libraries, Sketch structure, Serial Monitor, Variables, Binary Number System Pt1

- Libraries
- Sketch structure (A note on brace formatting)
- The serial port monitor
- Printing to the serial port monitor
- Variables and the assignment operator
- Binary number system Pt. 1.

Lesson 4 – Expressions, Conditionals, Blocks and Functions

- Arithmetic Expressions and Operators
- Incrementing and Decrementing Variables
- Truth Values in C++
- The If-Then Statement
- Code Blocks
- Functions

Lesson 5 – Binary Images, Arrays, Characters, Strings, Loops

- Loading Binary Images
- Arrays
- Characters and Character Codes
- Strings
- Conditional Loops Part 1

Lesson 6 – Loops (cont.), LED Matrix Displays, Nested Loops Advanced Functions, Binary Numbers Part 1

- For-Next Loops
- SPI Peripherals
- Using a MAX7219 LED Matrix Display
- Lighting and clearing individual pixels
- Advanced Functions
- Nested Loops

Lesson 7 – The Binary Number System (may take 2 lessons)

- Numerals vs numbers
- Review: the base 10 system and digit place values
- New: the base 2 system and digit place values
- Bits and bytes and nybbles
- Binary addition and subtraction

Lesson 8 – Producing Sound

- Formatting printed output in Serial Monitor
- Shifting and exponents
- Bitwise operations and masking
- Displaying text on the LED matrix display
- Review of sound wave theory
- Analog vs Pulse Width Modulation
- Producing sound tones with an Arduino microcontroller

Lesson 9 – Reading Analog and Digital pins

- Millis
- Reading buttons
- Debouncing buttons
- Reading analog values from a potentiometer

Lesson 10 – The I2C Bus and Peripherals

- I2C Bus Operation
- Initializing the I2C bus
- Accessing an I2C temperature sensor
- Real Time Clocks
- Accessing a DS3231 RTC

Lesson 11 – NTP and Text Management

- Numeric to ASCII Conversions
- Time representations and conversions
- Network Time Protocol
- Displaying the time on an LED matrix display

Lesson 12 – Text Management

- Changing the default font
- Text Effects
- Using multiple display zones

Lesson 11 – Numeric Conversions, Time Formats, Network Time Protocol

- Numeric to ASCII conversions
- printf and sprintf
- Time bases and formats in Linux and Windows
- Time Data Structures and Conversions
- Network Time Protocol
- Classroom Exercise – Using NTP
- Classroom Exercise – NTP LED Display

Numeric Conversions

- We previously looked at some ways to convert strings (arrays of data type char) into numbers
- Today we will look at two ways to convert numbers into strings
- **Arduino and other C implementations offer the itoa(), ltoa() and ultoa functions to convert from the respective numeric data types into strings**
- **Using these functions requires a character array which acts a buffer to receive the ASCII text result**
- **So as an example:**

```
char textbuffer[33]; long longvariable;
ltoa(longvariable,textbuffer,32);
```

Numeric Conversions (cont.)

- There are several problems with the xtoa functions
- Require careful buffer sizing
- Unavailable for conversion of floating point (decimal fractionated) numbers
- Trailing null processing can be tricksy
- Non-portable
  - Use of the standard conversions requires that the programmer know *in advance* the size of the data type(s) being converted
  - Code must be specific to the type size(s) on a given platform
  - This makes portability difficult at best, where extensive compile-time options must be deployed in an attempt to accommodate all possible platforms where the code might be used
  - This is undoubtedly the biggest problem with the standard conversion functions, and why they are rarely used with Arduino

fprintf(), printf() and sprintf()

- `fprintf()` and `printf()` are functions included in the standard I/O library of most C compilers (**`#include <stdio.h>`**)
- The `fprintf()` function formats and then outputs variables into so-called standard I/O streams. It provides extensive and precision control over how the output is structured
- Because microcontrollers do not have standard I/O streams, `fprintf()` is not used with the Arduino IDE, C++, or Processing
- However, `fprintf()` has a non-I/O counterpart called **`sprintf()`**
- Instead of sending its results directly to an output data stream, `sprintf()` puts the results in a character array buffer
- This makes `sprintf()` extremely versatile, and thus it is the preferred method for performing numeric conversions in Arduino

Output Format Descriptors

- Numerous programming languages make use of constructs called *output format descriptors*
- You have seen very simple ones used with the Serial.println() function; remember DEC, HEX, and BIN?
- The idea behind a format descriptor is that a variable or value is supplied to function along with a description of how that variable or value is to be formatted and then returned from the function.
- Format descriptors can provide extensive and very precise control over how the resulting formatted output looks: left-justified, zero-filled, how many decimal places included in fractional numbers, how many characters converted, etc.
- And this is the level of control which the format descriptors in `sprintf()` provide, making it so very useful
- `sprintf()` calls its format descriptors *format strings*

`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

sprintf() Example

Below is an example of how sprintf() can be used in actual code

```
#include <stdio.h>

int i=99; char c=65; double d=999999999;
float f = 1234567.9133333; unsigned int u = 65536;
char sometext[] = “hello world”;
char buffer[255];

void setup()
{
  Serial.begin(9600);
  sprintf(buffer,”%i @@ %c @@ %d @@ %f @@ %e @@ %u @@ %s, I, c, d, f, u, sometext);
  Serial.println(buffer);
}

void loop()
{
}
```

Timekeeping Epochs

- Nearly all operating systems have the concept of an epoch, the date and time from which the operating system measures time
- For Unix and Linux, the epoch is January 1, 1970
- For Windows, the epoch is January 1, 1601
- By contrast, for humans our epoch is January 1, 0 AD (or BCE if we are being secularly correct about it)
- Most computer systems determine *present time* as the amount of time elapsed since the epoch.
- For Unix/Linux systems, this is seconds passed since the epoch
- For Windows, it's 100-nanosecond (.1 uS) intervals passed since the epoch
- Epochs are not relative to any timezone, so they are all UTC (Universal Coordinated Time aka Greenwich Mean Time aka Zulu Time).

Epoch Problems

- At some point, the variable or register used to store time in the operating system will “roll over” when it reaches its maximum value
- For older mainframes that used January 1, 1900 as the epoch and stored the elapsed delta time using a 2-digit year, the rollover occurred at 23:59PM on December 31, 1999 leading to the so-called Year 2000 Crisis or Y2K
- Many Unix/Linux systems will experience an overflow problem on 19 January 2038 if not fixed beforehand. This is known as the Year 2038 problem.
- GPS (Global Positioning System) calculates its time from an epoch, transmits the offset between UTC time and GPS time, and must update this offset every time there is a leap second, requiring GPS receiving devices to handle the update correctly.

C++ struct tm

- C/C++ utilize a standard data structure for storing and handling time
- This data structure is `struct tm`, defined in `time.h`

Member	Type	Meaning	Range
tm_sec	int	seconds after the minute	0-60*
tm_min	int	minutes after the hour	0-59
tm_hour	int	hours since midnight	0-23
tm_mday	int	day of the month	1-31
tm_mon	int	months since January	0-11
tm_year	int	years since 1900	
tm_wday	int	days since Sunday	0-6
tm_yday	int	days since January 1	0-365
tm_isdst	int	Daylight Saving Time flag	

C/C++ ctime and localtime

- However, pursuant to the use of an epoch, it is often necessary to convert between epochal time, local time, and either of those in struct tm format
- `time_t` is the data type most commonly used in C implementations for storing epochal time, and is typically a long.
- `mktime()` is used to convert a struct tm to time_t
  `time_t mktime (struct tm * timeptr);`
- Likewise, `localtime()` is used to convert time_t formatted time into a struct tm
  `struct tm * localtime (const time_t * timer);`

Network Time Protocol

- The Network Time Protocol (NTP) is a networking protocol for clock synchronization between computer systems over packet-switched, variable-latency data networks. In operation since before 1985, NTP is one of the oldest Internet protocols in use.
- NTP is intended to synchronize all participating computers to within a few milliseconds of Coordinated Universal Time (UTC).[1]:3 It uses the intersection algorithm, a modified version of Marzullo's algorithm, to select accurate time servers and is designed to mitigate the effects of variable network latency. NTP can usually maintain time to within tens of milliseconds over the public Internet, and can achieve better than one millisecond accuracy in local area networks under ideal conditions
- Usually seen implemented as client-server

Network Time Protocol

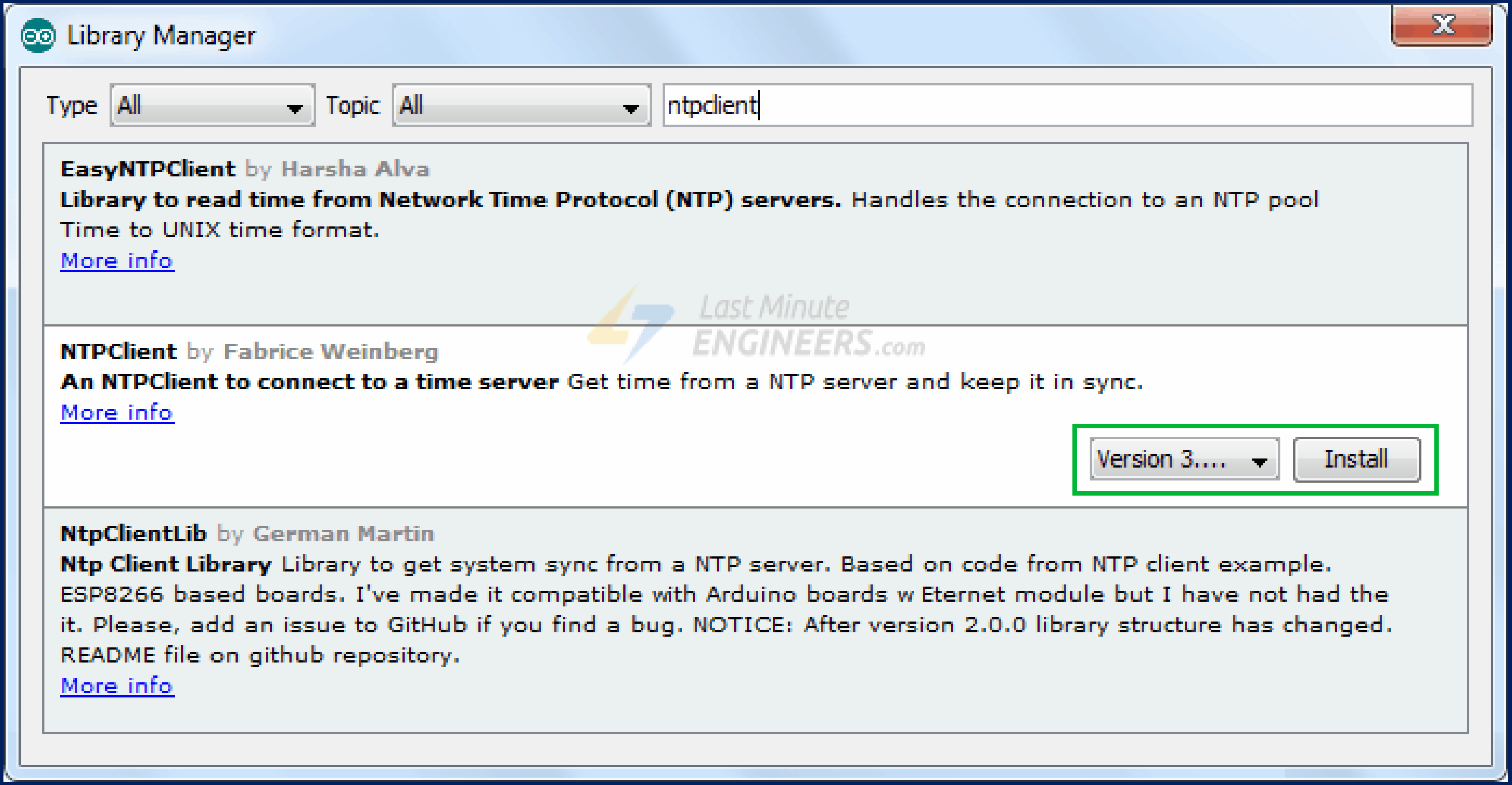
- The Network Time Protocol (NTP) is a networking protocol for clock synchronization between computer systems over packet-switched, variable-latency data networks. In operation since before 1985, NTP is one of the oldest Internet protocols in use.
- NTP is intended to synchronize all participating computers to within a few milliseconds of Coordinated Universal Time (UTC).[1]:3 It uses the intersection algorithm, a modified version of Marzullo's algorithm, to select accurate time servers and is designed to mitigate the effects of variable network latency. NTP can usually maintain time to within tens of milliseconds over the public Internet, and can achieve better than one millisecond accuracy in local area networks under ideal conditions
- Usually seen implemented as client-server

ESP8266 and NTP

- With network capable microcontrollers, the typical practice is to capture UTC from the NTP server, then make whatever adjustments are needed for local time.
- A good NTP client implementation will only require polling of the NTP servers at wide intervals, in most cases several hours apart
- In no case should NTP servers be polled more frequently than every 15 minutes, lest the polling be interpreted as a denial-of-service attack by the organization running the servers
- If the organization is NIST– the Federal Government– things could go very badly wrong indeed if a DoS attack was perceived!

The NTP Library

- As usual, we will need to install an NTP client library in order to connect to NTP servers
- Go ahead and install this one now



EasyNTPClient by Harsha Alva
Library to read time from Network Time Protocol (NTP) servers. Handles the connection to an NTP pool Time to UNIX time format.
More info

NTPClient by Fabrice Weinberg
An NTPClient to connect to a time server Get time from a NTP server and keep it in sync.
More info

NtpClientLib by German Martin
Ntp Client Library Library to get system sync from a NTP server. Based on code from NTP client example. ESP8266 based boards. I've made it compatible with Arduino boards w Eternet module but I have not had the it. Please, add an issue to GitHub if you find a bug. NOTICE: After version 2.0.0 library structure has changed. README file on github repository.
More info

Classroom Exercise - NTP

And here is our class sketch for NTP. Go ahead and load and run!

```
#include <NTPClient.h>
#include <ESP8266WiFi.h>
#include <WiFiUdp.h>

const char *ssid     = “CFC_Public";
const char *password = "";

const long utcOffsetInSeconds = 3600;

char daysOfTheWeek[7][12] = {"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"};

// Define NTP Client to get time
WiFiUDP ntpUDP;
NTPClient timeClient(ntpUDP, "pool.ntp.org", utcOffsetInSeconds);

void setup(){
  Serial.begin(9600);

  WiFi.begin(ssid, password);

  while ( WiFi.status() != WL_CONNECTED ) {
    delay ( 500 );
    Serial.print ( "." );
  }

  timeClient.begin();
}

void loop() {
  timeClient.update();

  Serial.print(daysOfTheWeek[timeClient.getDay()]);
  Serial.print(", ");
  Serial.print(timeClient.getHours());
  Serial.print(":");
  Serial.print(timeClient.getMinutes());
  Serial.print(":");
  Serial.println(timeClient.getSeconds());
  //Serial.println(timeClient.getFormattedTime());

  delay(1000);
}
```

Classroom Exercise – NTP Display

- Let’s see how y’all do with this one!
- Combine SKETCH10B and SKETCH11A to display NTP time on your LED matrix display
- Remember that you can use sprintf() to create a string that MD_Parola will print on the LED display
- It might be easier to copy the necessary bits of 10B into 11A, but that’s entirely up to y’all!
- Go to it and good luck! ☺

Formal End of Lesson 10

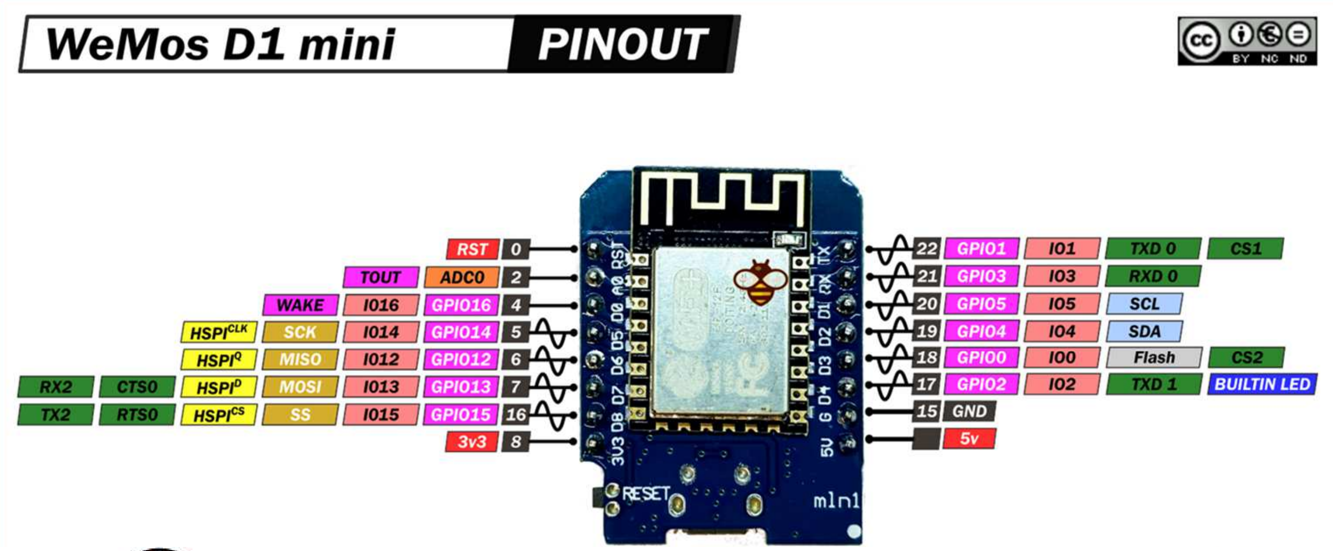
HOMEWORK!

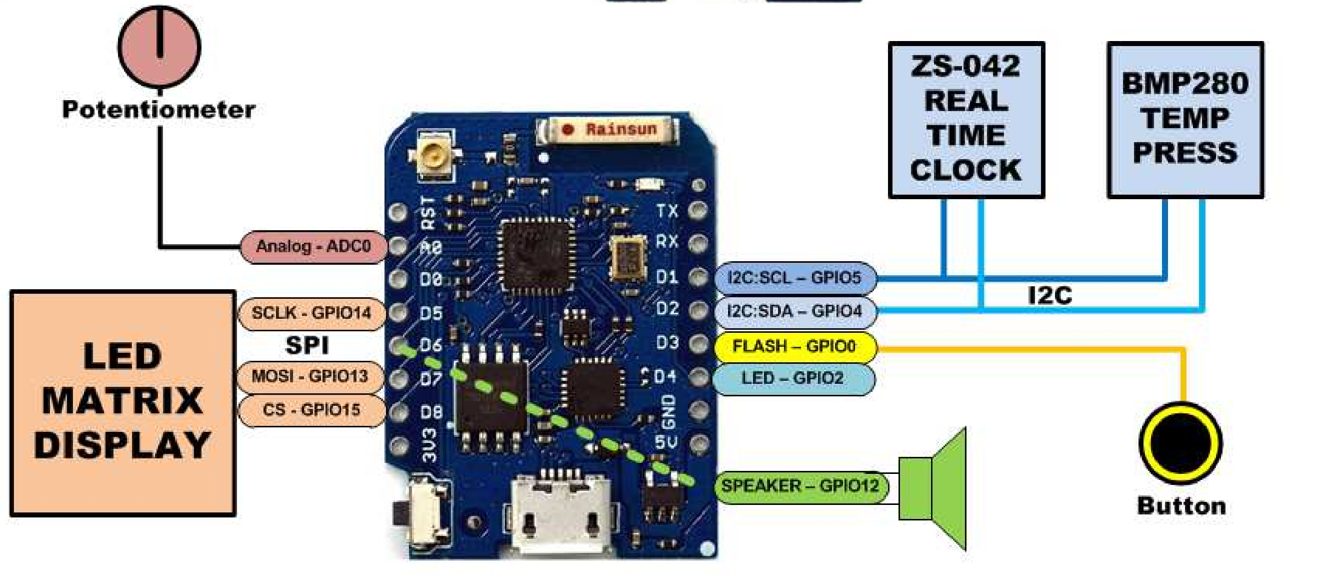
- **Read up on how NTP works**
- **Study the MD_Parola documention and see if you can figure out how to blink the colon when displaying time**

In next week’s exciting episode

- Changing the default font
- Text Effects
- Using multiple display zones

LESSON REFERENCE



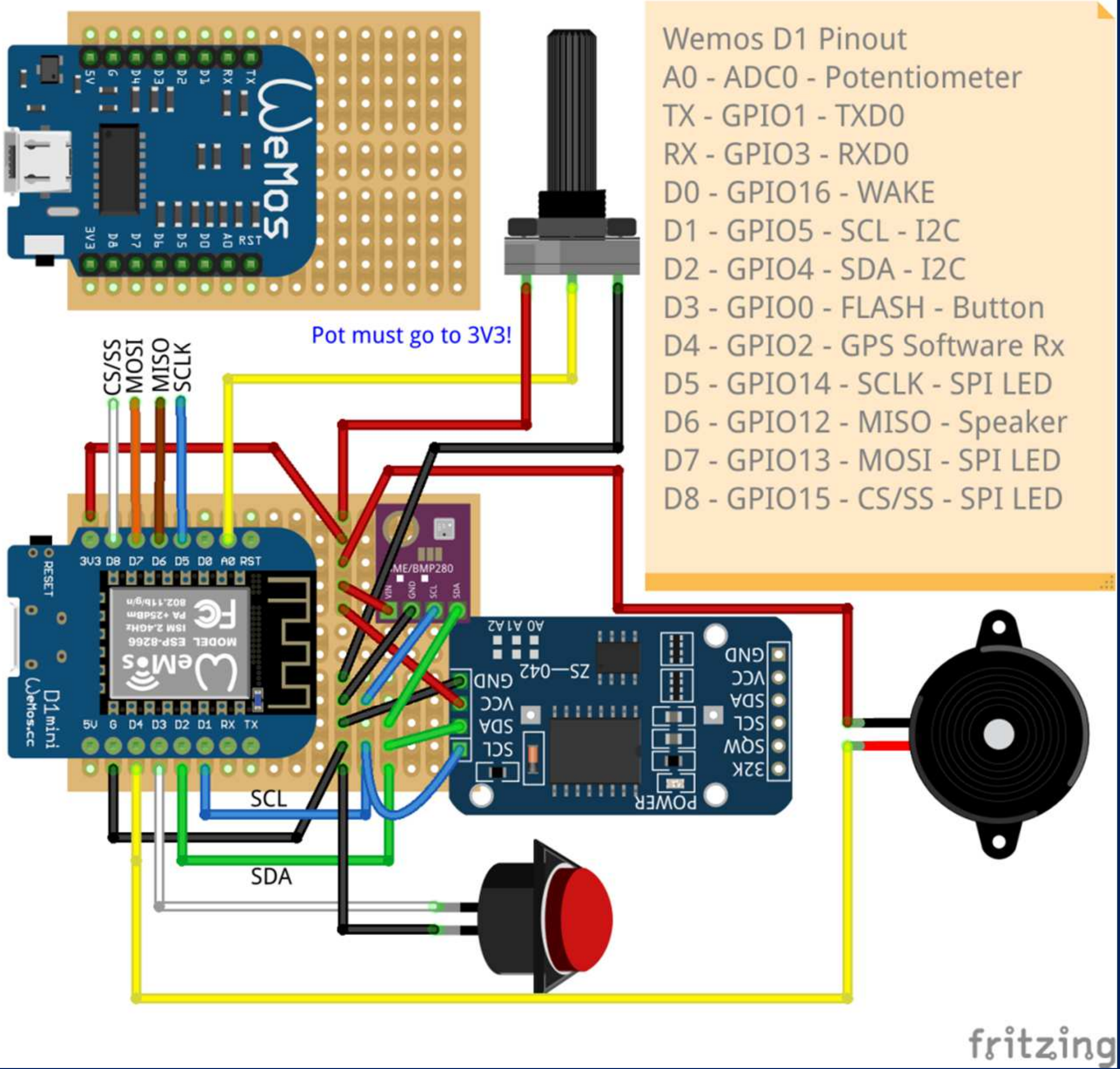


SEICHE LED DISPLAY ARCHITECTURE

LESSON REFERENCE

Pin Assignment Notes
GPIO16/D0 - HIGH at boot - No interrupt, no PWM or I2C - Unused
GPIO2/D4 - HIGH at boot - Input pulled up, output to onboard LED - - probable GPS RX software serial
GPIO12/D6 - Piezo Speaker (not used in SPI LED Matrix)
GPIO[12],13,14,15/D6,D7,D5,D8 - MISO,MOSI,SCLK,CS - SPI
GPIO4,5/D2,D1 - SDA,SCL - I2C
ADC0/A0 - Analog Input - Potentiometer 3.3V divider
GPIO0/D3 - Input pulled up - FLASH button, boot fails if pulled low - button to ground

SPI - LED matrix : 12,13,14,15
I2C - RTC,BMP280 : 4,5
Serial RX - GPS : 16 SS
Input pullup with interrupt - Button : 0
Piezo Speaker : 2
Analog Input - Potentiometer : ADC0



Wemos D1 Pinout
A0 - ADC0 - Potentiometer
TX - GPIO1 - TXD0
RX - GPIO3 - RXD0
D0 - GPIO16 - WAKE
D1 - GPIO5 - SCL - I2C
D2 - GPIO4 - SDA - I2C
D3 - GPIO0 - FLASH - Button
D4 - GPIO2 - GPS Software Rx
D5 - GPIO14 - SCLK - SPI LED
D6 - GPIO12 - MISO - Speaker
D7 - GPIO13 - MOSI - SPI LED
D8 - GPIO15 - CS/SS - SPI LED