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
- Time representations and conversions
- Accessing a DS3231 RTC

Lesson 11 – NTP and Text Management

- Network Time Protocol
- Displaying the time on an LED matrix display
- Changing the default font
- Text Effects
- Using multiple display zones

Lesson 8 – Reading Analog and Digital Pins

- The millis() built-in function
- Setting pin modes (review)
- Reading a digital input
- Debouncing buttons
- Classroom Exercise – Button debouncing
- Analog vs Digital Inputs
- ADC resolution and you
- Reading A0 on an ESP8266
- Classroom Exercise – Utilizing A0 input to set display brightness

Note On Integer Data Options

- We've already looked at bytes (8 bits) and integers (32 bits on ESP8266, or 4 bytes)
- There are some additional data types in Arduino for working with integers
- You will often see unsigned numeric data types declared like this:

```
    uint16_t someinteger;
or
    uint8_t somebyte;
```

- What are these weird data types? They are used for more clearly declaring unsigned integer variables

Arduino convention	Traditional C convention
Uint8_t	byte or unsigned char
Uint16_t or Uint32_t	unsigned int
Uint32_t or Uint64_t	unsigned long

You can find these– and more!– in the **c_types.h** header file

Linux: ~/.arduino15/packages/esp8266/hardware/esp8266/3.0.2/tools/sdk/include/c_types.h

The millis() built-in function

- millis() is a built-in function that returns the number of milliseconds since the last time the processor was started or reset
- The return value from millis() is 64 bits wide; this means you can must use either a long or uint64_t for capturing the value
- Thusly:

```
uint64_t elapsedtime;
elapsedtime=millis();
```

- Or:

```
long elapsedtime;
elapsedtime=millis();
```

- Using smaller variable storage will result in a truncated value; this may be desirable in certain cases, but is probably not what you want!

Review: Pin Modes

- Microcontroller GPIO pins can be configured to *either* send data (OUTPUT mode) or receive data (INPUT mode)
- Output pins are declared like this:
  ```
  pinMode(6, OUTPUT);
  ```
- As you might expect, input pins are declared this way:
  ```
  pinMode(2,INPUT);
  ```
- There is an additional mode available for input pins, where an integral (part of the microcontroller SoC) pullup resistor can be engaged; doing it this way will “pull up” the input line to the running voltage of the processor:
  ```
  pinMode(1,INPUT_PULLUP);
  ```
- The latter is the preferred way to handle buttons, such that the button is wired to ground and pressing it will drop the input line to zero voltage

Read a Button – Sketch 9A

- Go ahead and load SKETCH9A into your IDE
- You may recall from the functional diagram of your LED displays that GPIO1 (Wemos pin D2) is connected to the button on top of your display
- Pins are read with the `digitalRead()` function

```
#define BUTTON D3
int ivalue;
void setup()
{
  Serial.begin(9600);
  pinMode(BUTTON,INPUT_PULLUP);
}

void loop()
{
  ivalue = digitalRead(BUTTON);
  Serial.println(ivalue,DEC);
  delay(250);
}
```

- Upload the sketch and try pressing the button on your display
- What output do you see in the serial monitor?
- Why are you seeing the output that you are?

Debouncing Buttons

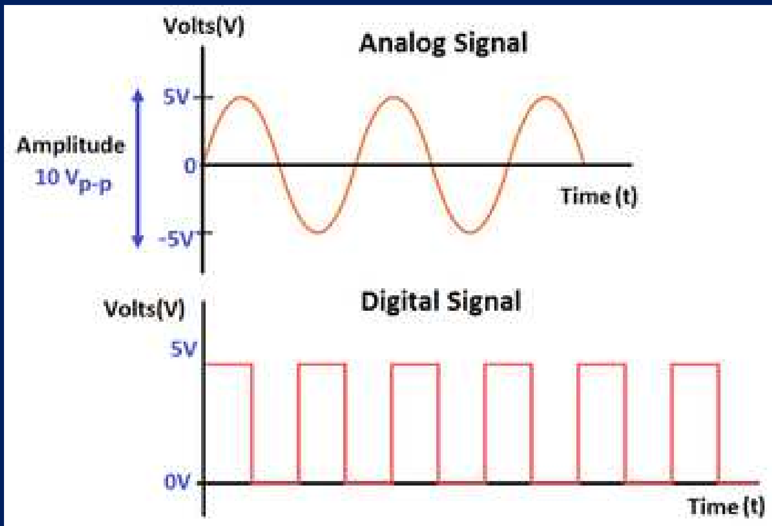
- Buttons use mechanical contacts, and mechanical contacts take some time to “settle” into their final state when actuated.
- While the settle time of several milliseconds is unnoticeable to us, it’s very noticeable to a computer or microprocessor because of their fast sampling speed
- The result is that a microcontroller will see a single button press by an operator as multiple button presses and releases as the contacts “bounce” several times before settling into a connected state
- As a result of this behavior, it becomes necessary to “debounce” button contacts
- Debouncing can be performed in hardware (tricky) or in software (much easier and cheaper.)

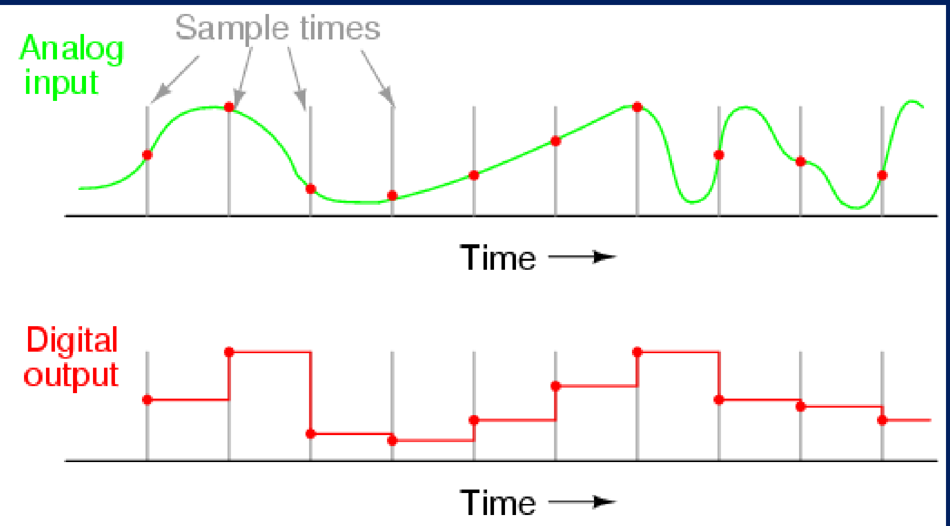
Class Exercise - Debouncing

- Class discussion: how can we use software to debounce a button press – that is, record only the first connection of the contacts?
- Go ahead and modify your sketch to use one of the ways that was discussed
- Did it work? If so, why not?

Analog vs Digital Inputs

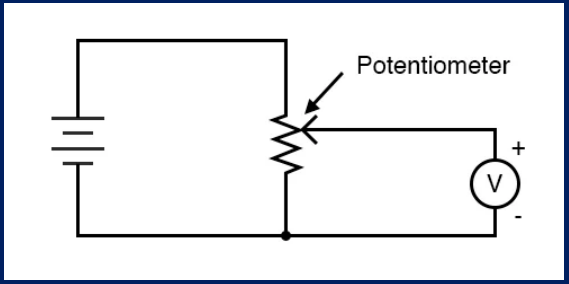
- Digital signals are *discrete or finite*; they can only have certain values
- Analog signals (like audio)
- Circuitry that converts from analog to digital is called an Analog-to-Digital Converter (ADC)
- ADCs perform conversion by sampling an analog signal *at intervals*, then generating a digital value that represents the signal *at that point in time*.





Reading Analog Inputs

- The sampling interval of an ADC determines it's *resolution*. Common resolutions for microcontrollers are 10-bit and 20-bit.
- The ADC resolution means that the analog input will have values *within the range of that number of bits*, that is, 2^10 or 2^20 possible values.
- The ESP8266 has a single analog input, on pin A0, and it uses a 10-bit ADC.
- To provide stable input, ADCs connected to potentiometers are connected using voltage divider (one end of the potentiometer is connected to ground, the other to VCC).



Read Analog Input – Sketch 9B

- Go ahead and load SKETCH9B into your IDE
- You may recall from the functional diagram of your LED displays that A0 on the ESP8266 is connected to the potentiometer (knob) on the back of the display
- Pins are read with the `analogRead()` function; *analog pins do not need to be initialized*

```
int avalue;
void setup()
{
  Serial.begin(9600);
}

void loop()
{
  avalue = analogRead(A0);
  Serial.println(avalue,DEC);
  delay(250);
}
```

- Upload the sketch and rotate the potentiometer on your display
- What output do you see in the serial monitor?
- Why are the largest and smallest what they are?

CLASS ASSIGNMENT

- The MD_Parola library method `setIntensity(uint8_t intensity)` changes the brightness of your LED display

Combine SKETCH8A and SKETCH9B

- **Using SKETCH8A and SKETCH9B as references, create a sketch which sets the display brightness based on the setting of the potentiometer**
- **Note: There are 255 possible brightness settings, and 1024 possible potentiometer values, so you will need to map one to the other!**
- **Hint: Start with Sketch 8A and add the analog read statements there**

I've structured this as an in-class assignment so that I'm available for any questions

Formal End of Lesson 9

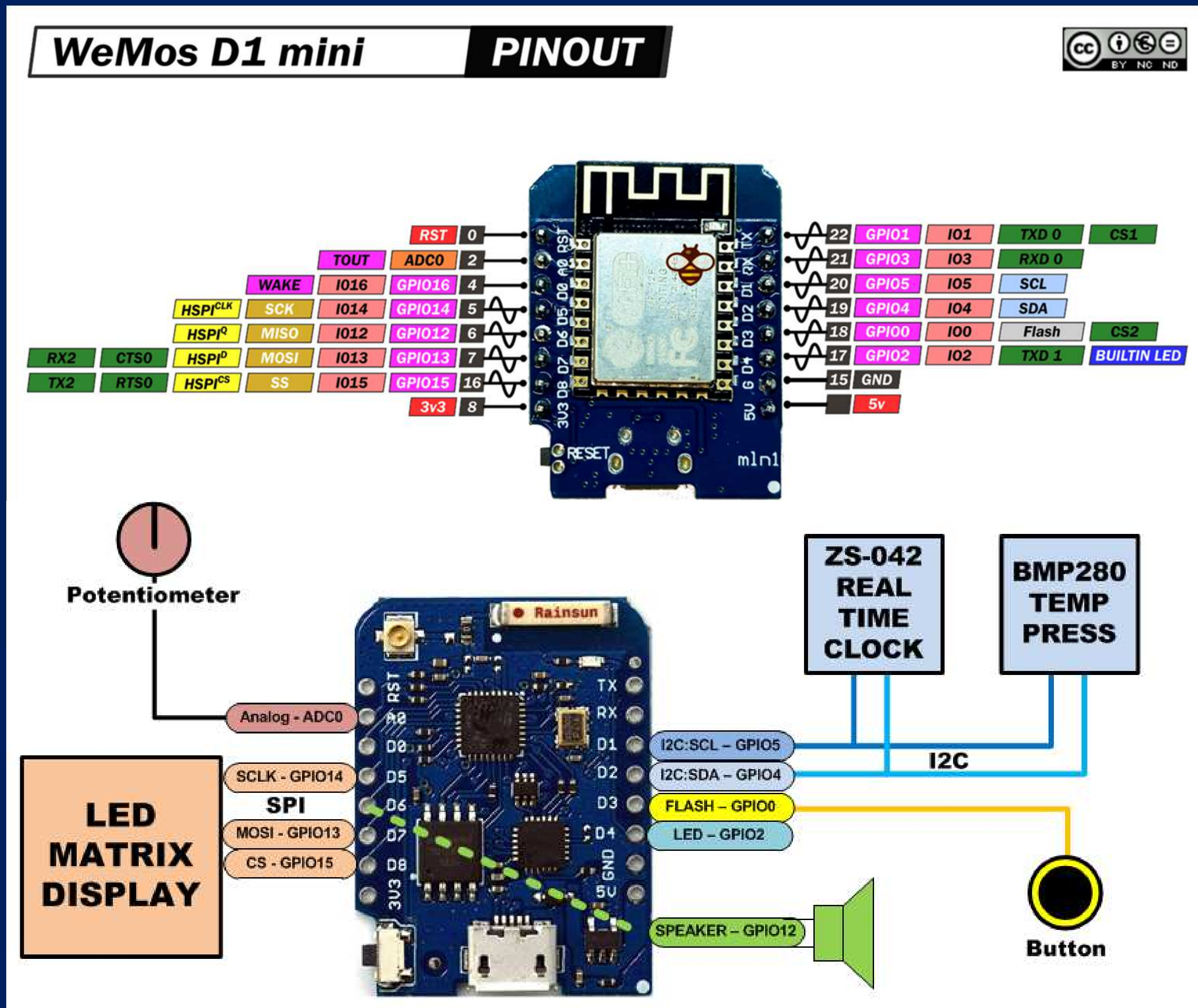
HOMEWORK!

- **No homework this week!**

In next week's exciting episode

- I2C Bus Operation
- Accessing a temperature sensor
- Accessing a DS3231 Real Time Clock (RTC)
- Setting an RTC

LESSON REFERENCE

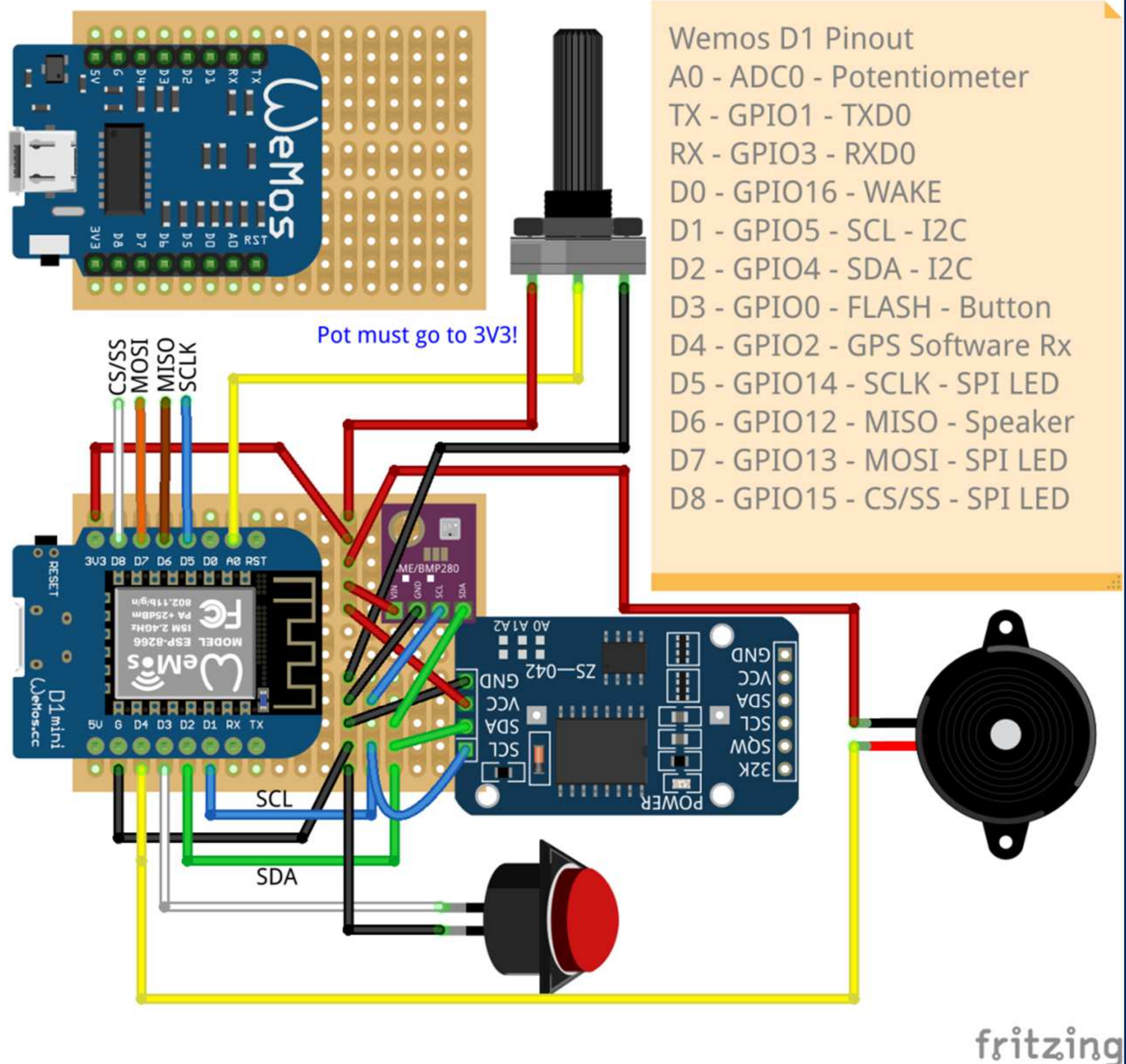


SEICHE LED DISPLAY ARCHITECTURE

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