SEICHE 2022
Basic Arduino Programming

Instructor: Paul Frommeyer

www.paulfrommeyer.com

Corporate Sponsor: 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 4/5 – The Binary Number System Cont. (may take 2-3 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
- Formatting printed output
- Shifting and exponents

Lesson 6/7 – Integers and Arithmetic Operators (may take 2 lessons)

- **The integer data type**
- **Variables and the assignment operator**
- **Autoincrement and autodecrement**
- **Arithmetic Operators**
- **Assigning arithmetic results to a variable**
- **Bytes as data types**

Lesson 8 – Communicating with Sensors

- **Reading a potentiometer**
- **Reading a button**
- **Initializing I2C sensors**
- **Reading I2C sensors**

Lesson 9 – Characters and Arrays

- The hexadecimal number system
- Representing characters as numbers
- The ASCII character code
- The char data type
- Strings and character arrays

Lesson 10 – Comparison Operators

- ==, =>, <=
- strcmp
- Binary comparisons
- Uploading a sketch to the microcontroller

Lesson 11 – Control Structures

- if-then-else
- while
- for-next

Lesson 12 – Programming the LED matrix

- Initializing the SPI interface
- Controlling display brightness
- Lighting and clearing a single pixel
- Displaying text on the LED matrix
- Default fonts
- Nested for-next loops

Lesson 4 – Expressions, Conditionals, Blocks and Functions

Part A

- Statement syntax review
- Arithmetic Expressions and Operators
- Incrementing and Decrementing Variables
- Truth Values in C++
- The If-Then Statement
- Brace Style and Comments

Part B

- Code Blocks
- Functions
- Function defintions (part 1)

Statement Syntax Review

- Statements in a C++ program execute *sequentially*, top-to-bottom. In this sense sketches/programs are like a food recipe.
- Most statements appear on lines by themselves; this is convention. The language requires only that they be separated by whitespace (space, tab, newline). Note that newline is considered whitespace.
- All statements, including assignments and defintions, must be terminated with a semicolon (the C++ counterpart to an English period.)
- Constants are numbers which appear "as themselves" in C++ code, e.g., `3.1415926`
- Variables are containers for numbers or other data.
- Variables must be declared as a particular data type.
- The `setup()` and `loop()` sections begin and end with open { and close } braces, respectively.
- **C++ is case sensitive!** `An_Integer_Variable` and `an_integer_variable` are not the same entity!

Brief Note On Sanitizing Data

- Some of you have run into issues when directly copying text from PDF files (our lessons) into the IDE. This is due to hidden characters which get copied from the PDF and confuse the IDE. Windows is particularly prone to this
- The solution is to first paste the text into a plain text editor. This strips out any hidden characters and leaves only the program code
- On Windows, the tool for doing this is Notepad
- On Linux (Kubuntu), the Kate text editor will work
- Both can be accessed via the Start/Applications menu
- For best results: copy from the PDF, paste into the editor, deselect the text, the reselect, copy, and paste into the IDE
- Yes, the IDE should do this for us, but here we are

Brief Note On `delay()`

- We will cover functions in more depth later
- For now, know that `delay()` is a function which causes the processor to pause (wait) for that amount of time before continuing on in a sketch.
- The syntax (formal usage rules) for delay() is:

```
delay(number of milliseconds to wait)
```

- It is very important to use delay statements, or some other means of slowing output, when using the serial monitor
- This is because Arduino boards run fast enough that their serial output, unchecked, can overload the computer and operating system the IDE is running on
- This is not a problem when running a board standalone (not connected to the IDE), only when using the Serial Monitor.

Arithmetic Expressions and Operators

- An arithmetic expression is a combination of arithmetic operators that produces a numerical result.
- Arithmetic expressions are normally used as arguments (data input) to statements, such as the assignment operator =.
- The basic C arithmetic operators, as previously mentioned, are + (addtion), - (subtraction), * (multiplication), / (division), % (modulus or modulo or remainder) and ^ (exponentiation).
- There are also binary operators which operate on the raw binary data that defines a constant or variable. More on those later.

Incrementing and Decrementing Variables

- Incrementing a variable means adding to it, and likewise decrementing means subtracting from it.
- C++ provides autoincrement operators which can be applied to numeric variables. These are traditionally written in postfix form (after the variable name.) They can be applied to a variable pretty much anywhere, however that can lead to chaos, so they are generally written as unique statements, with some exceptions as we'll see.
- The autoincrement variable is ++. It is used like this:

```
some_variable_name++;
```

  Autoincrement will add one (1) to the variable it is applied to.
- The autodecrement variable is --. It's use is similar:

```
some_variable_name--;
```

Autoincrement/decrement

Let's watch how autoincrement works. Enter the following sketch and upload it to your board. Be sure to have the Serial Monitor open (and set for 9600 speed) to watch the output.

```
byte testvar = 0;

// the setup function runs once when you press reset or power up
void setup() {
  // Setup port for serial monitor
  Serial.begin(9600);
}

// the loop function runs over and over again forever
// so will increment testvar forever
void loop() {
  Serial.println(testvar,DEC);
  testvar++;
// Do not reduce delay below 250 to avoid locking up IDE or laptop
  delay(500);
}
```

Describe your results. What is happening, and why?

Self-Assignment

- In C++, and other languages, there is a sometimes confusing operation in which the = assignment operator is used to add the value of a variable back to *itself*.
- The key to how this works is in the phrase “the value of a variable”.
- Written out, the process works like this:
  - Take the value of the current contents of a variable
  - Then add those contents to the same variable
- In program code, this looks like:

```
a = a + 5;
```

- It works because the value of A is taken before the arithmetic operation is performed. So if a contains the value 8, the what’s really happening is this:

```
a = 8 + 5;
```

Self-Assignment

Let's modify our sketch to use self-assignment instead of autoincrement. Replace the testvar autoincrement as shown below, then upload the sketch and watch the results in the monitor.

```
byte testvar = 0;

// the setup function runs once when you press reset or power up
void setup() {
  // Setup port for serial monitor
  Serial.begin(9600);
}

// the loop function runs over and over again forever
// so will increment testvar forever
void loop() {
  Serial.println(testvar,DEC);
  testvar = testvar + 8;
// Do not reduce delay below 250 to avoid locking up IDE or laptop
  delay(500);
}
```

Describe your results. What is happening, and why?

Truth Values in C++

- The C++ language has a limited concept of “truth”. This is exactly equivalent to whether a circuit is on or off.
- As you might expect from previous lessons, truth for an Arduino is, then, expressed as either a zero (false) or *anything else* (true).
- The IDE provides a “shorthand” when we need to explicitly refer to these values:
  TRUE is a word that is “mapped” to the number 1
  FALSE is a word that is “mapped” to the number 0
- Why do we care? Because all arithmetic expressions evaluate to (produce) a truth value. This is used in more advanced computer operations, and is critical for *conditional expressions*, which we will look at next.
- So to give some simple examples:
  - The result of `9-9` would be false, because the result is *zero*
  - The result of `0 + 1` would true, because the result is *one*.
  - The result of `2 + 2` would be true, because it is *not zero*
- There is also a negation operator in C++, which evaluates to the opposite of the truth value it is applied to.
- The negation operator is the exclamation mark, !, and it works like this:

```
!TRUE
!some_variable
```

Conditional Operators

- Because truth is so import for programming, C++ and other languages provide conditional operators, sometimes called comparison operators, which can evaluate the value of one variable against a constant, another variable, or even an arithmetic expression.
- The conditional operators in C++ are:
  - Logical equals; this is done with a double equals sign: ==
  - Logical not equal; this is a negated equals sign: !=
  - Greater than; this is the (wait for it) greater-than sign: >
  - Less than; this is the (no surprise) less-than sign: <
  - Equal to or greater than: =>
  - Less than or equal to: <=
  - And, as you've already seen, a simple negation, !
- Here are some examples; each expression will evaluate to true or false
  - `10/5 == 2`
  - `some_variable > 99`
  - `one_variable != another_variable`
  - `one_variable == another_variable + 17`

Note the absence of terminating ; -- these are *expressions*, not *statements*!

The If-Then Statement

- The if-then statement is probably the most important program flow control statement in C++ (or any other language).
- If-Then uses conditional expressions to determine whether a statement (or block of statements) is executed.
- The formal syntax for if-then is:

```
if (conditional expr) then statement;
```

- Usually, the “then” is placed on a line by itself:

```
if (TRUE)
then
        Serial.println(“Hello World”);
```

- But what you will most commonly see after the then is a code block (discussion later), like this:

```
if (2+2 == 4) {
        Serial.println(“It works on my machine!”);
}
```

If-Then – Let's Give It A Go

Let's modify our sketch to use if-then to reset the variable when it reaches a certain value. Update your code as shown below, then upload the sketch and watch the results in the monitor.

```
byte testvar = 0;

void setup() {
  // Setup port for serial monitor
  Serial.begin(9600);
}

void loop() {
  Serial.println(testvar,DEC);
  testvar = testvar + 8;
  if (testvar > 99) {
     testvar = 0;
  }
  delay(500);
}
```

Describe your results. What is happening, and why?

Brace Style

- The way braces are arranged in C++ is something of a religious war among programmers.
- The “holy canon” style used by the original C language authors works like this, and is what you will see most often:

```
if (TRUE) {
 do_something();
}
```

- The style used by us “reformationists”, because it makes checking brace congruity oh-so-much-easier, is this:

```
if (TRUE)
{
        do_something();
}
```

Comments

- Comments are any alphanumeric text that is completely ignored by the IDE/compiler
- Arduino C++ offers two methods for including comments in code
- The first you have already seen

```
// check it out, it's a comment!
```

- The second has been present in C since 1972, and is useful for including large blocks of text (or commenting out large blocks of code)

```
/*
Any text you want can go between the comment
delimeters. Note that the delimeters are the
_inverse_ of each other
*/
```

Code Blocks

- Code blocks are a means of grouping whole sections of code *so that they can be treated as a single statement.*
- You have already seen this used with the if-then statement:

```
if (TRUE) {
    // this is a block of code between the braces
}
```

- All program flow control statements will work with blocks
- But you don't have to use a control statement to use a block:

```
{
    // A bare code block
    a++;
}
```

- Code blocks are pervasive in C++; programmers prefer them for control statements because they allow the easy addition or removal of other statements within the block

Functions

- A function is a means for executing a code block from *within another code block*
- You have already seen a whole bunch of them:

```
setup()
loop()
delay();
```

- Depending on the function, it may or may not be called (invoked) using *parameters*.
- Parameters are data that is passed to the function, such as a time value:

```
delay(1000);
```

- There can be more than one, again depending on the function:

```
some_function(parameter1, 3.1415926);
```

- What *data type* the parameter(s) have, and what *values* they contain, are entirely dependent on the function (and what you want it to do.)

Function Definitions

- You have already seen what a function definition looks like, because that is precisely what you've been doing with the setup() and loop() functions!

```
setup() {
        // bunch of setup code goes here
}
```

- So the way you define a function is really simple:

```
function_name() {
    // code block
}
```

- The parentheses after the function name followed by a code block are the key.
- Except in very rare circumstances, **functions must be declared outside the setup() and loop() functions.** Declaring functions *inside* other functions is possible, but it is fraught due to scoping (which we are not talking about this semester) so most C++ compilers simply won't permit it, and you'll get errors if you try.
- We will cover declaring functions with parameters in another lesson

Function Definitions - Excercise

So just for kicks and to show what we've learned, let's move the bulk of our loop statements to a function. Update and upload, folks!

```
byte testvar = 0;

void setup() {
  // Setup port for serial monitor
  Serial.begin(9600);
}

void loop() {
  Serial.println(testvar,DEC);
  modifytestvar();
  delay(500);
}
Void modifytestvar() {
  testvar = testvar + 8;
  if (testvar > 99) {
     testvar = 0;
  }
}
```

Did it work? If not, check your braces and semicolons!

Formal End of Lesson 4

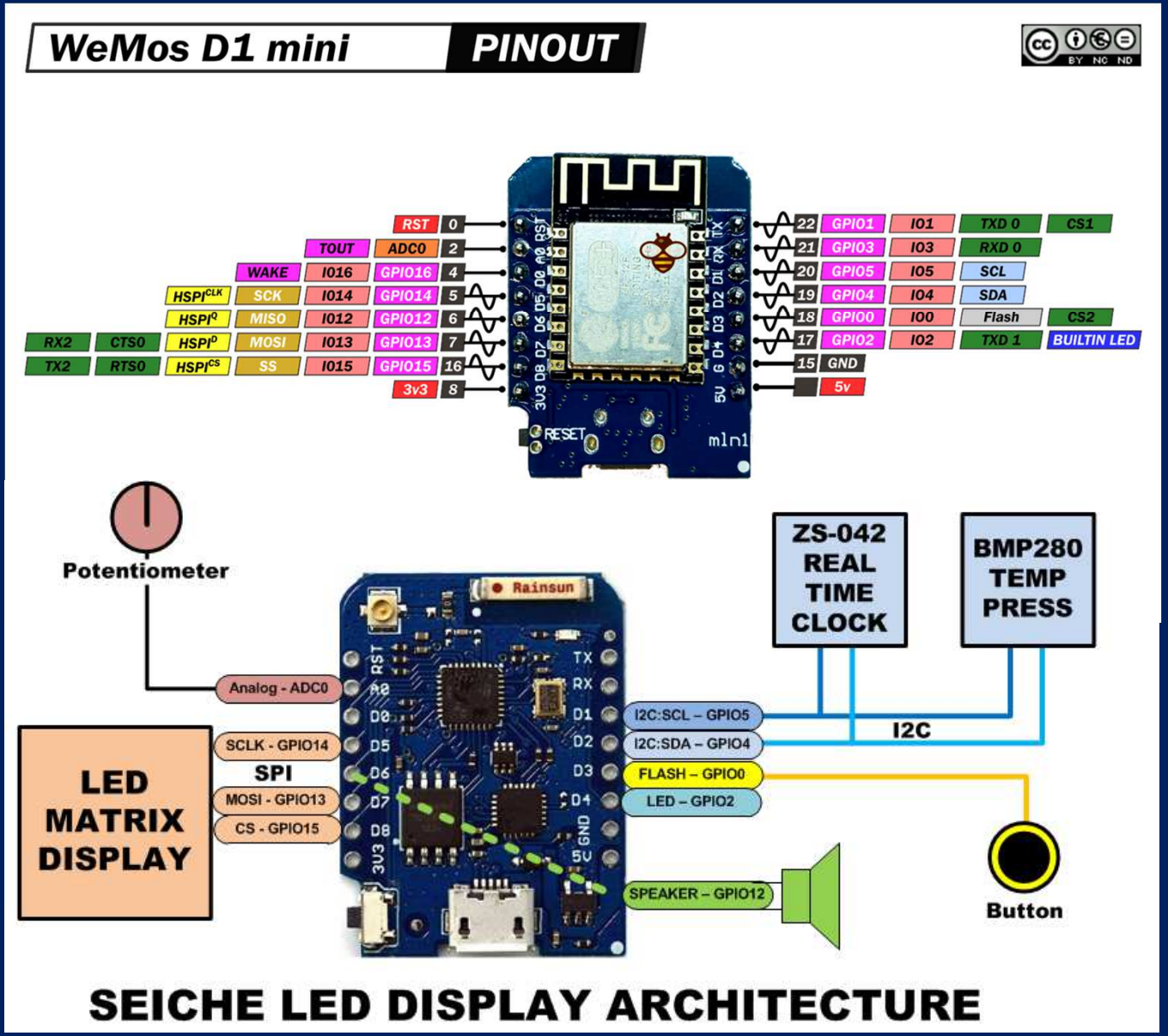
HOMEWORK!

- **Arduino Cookbook – *Make sure you have read ALL of Chapter 2 and Chapter 3***
- **A programming assignment! Yay!**
  Write a sketch to increment a number until reaches 200, then decrement it until it reaches 0, then start incrementing it again. You can use either the ++/-- operators or selfassignment for variable updating. You'll execute the sketch next week.
  Yeah, it's a desk-checked exercise, which I hate – although feel free to use a GNU C compiler locally if you just can't wait. ☺

In next week's exciting episode

- Arrays and strings
- Functions and parameters
- The for-next loop
- Nested for-next loops
- The binary number system (assuming I get those displays done!)

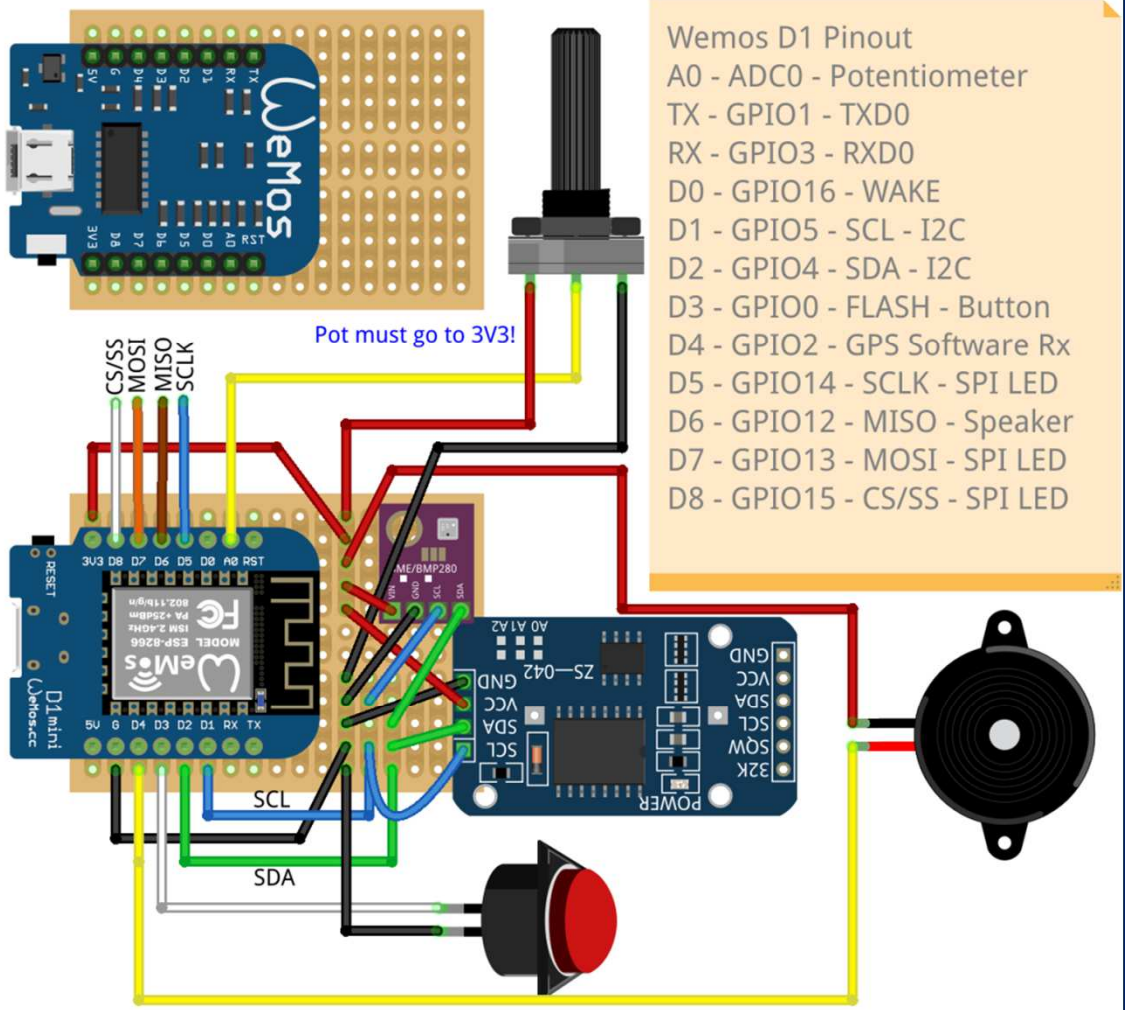
LESSON REFERENCE



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