SEICHE 2023
Intermediate Arduino Programming for IoT

Instructor: Paul Frommeyer

www.paulfrommeyer.com

Corporate Sponsor: DXC Technology

DXC TECHNOLOGY

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
- Class Exercise: Using complex data structures
- Introduction to linked lists
- Linked list usage example
- Bitmap font architecture review
- 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
- Installation

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

New Flash Drive Addition

- What is HackSpace magazine?
  HackSpace magazine is the new monthly magazine for people who love to make things and those who want to learn. Grab some duct tape, fire up a microcontroller, ready a 3D printer and hack the world around you!
- Digital copies in PDF for all issues are published free online for download
- Since educational use is encouraged, I have compiled all 63 issues (to-date) and placed them on your flash drives for your entertainment and reference
- https://hackspace.raspberrypi.com/issues?page=1

HackSpace Magazine

Lesson 2 – Pointers, Complex Data Structures

Part A

- CLASS EXERCISE – LESSON 1 REVIEW

Part B

- Introduction to complex data structures
- Declaring data structures with struct
- Accessing structure members
- The typedef statement
- Using struct and typedef
- Declaring data structures with union
- Layered data structures

CLASS EXERCISE – SKETCH 2A

- Go ahead and load SKETCH2A into your IDE
- Study the preprocessor directives at the beginning of the sketch
- Study the call-by-reference example using pointers to char

```
#define DISPLAYMSG "Welcome to SEICHE 2023"
#ifndef DISPLAYMSG
  #define DISPLAYMSG "BENGALS FTW 2023!"
#endif
#define CBR Foo

char displaytext[] = DISPLAYMSG;

void showmsg(char *msg)
{
  #ifdef DISPLAYMSG
    msg = "COLTS FTW 2024!";
  #endif
  pmx.displayText(msg,PA_LEFT,50,0,PA_SCROLL_LEFT,PA_SCROLL_LEFT);
}
// stuff removed

void loop()
{
  if(pmx.displayAnimate())
  {
    showmsg(displaytext);
    pmx.displayText(displaytext,PA_LEFT,50,0,PA_SCROLL_LEFT,PA_SCROLL_LEFT);
  }
}
```

- Upload the sketch. What were your results, and why?
- Next, modify your sketch to not alter the message by changing an appropriate `#define`
- Lastly, change the modification of the passed variables in `showmsg()`.

Complex Data Structures

- A complex data structure is a combination or "grouping together" of different variables of different types into a single data structure
- The statement that is used to accomplish this grouping is the `struct` directive. This directive creates a complex data type. The complex data type works similar to other data types like int and char.
- Syntax: `struct` *`structurename`*

```
struct structurename
{
    member1 type member1 name;
    member2 type member2 name;
}
```

Complex Data Structures (cont)

- Example 1

```
struct point
{
    int x-coordinate;
    int y-coordinate;
}
```

- Example 2

```
struct student_record
{
    char* name;
    int grade_science;
    int grade_english
    float gpa;
}
```

Complex Data Structures (cont)

- **Accessing structure members**

```
struct point
{
   int x-coordinate;
   int y-coordinate;
}
struct point coordinates;
coordinates.x = 10;
coordinates.y = 20;
```

- **Note that before it can be used, a variable must be declared of that type. In this case `coordinates` is the variable, and it is of type "point".**
- **Member access has the form *structurename.membername***
  Note the use of the period to separate the two entities
- **Since members are of standard variable types, the same operations can be performed on a member as can be performed on any other variables of that type**

Pointers to Structures

- The most common use of structures is to pass an entire structure to a function using call-by-reference. A *pointer* to the structure is passed, thus, the variable that is declared is a pointer to that type of structure.
- Pointers to structures

```
struct point

{
    int x-coordinate;
    int y-coordinate;
}
struct point *coordinates; // Note the * makes this a pointer
```

- Because this is largely what structures were made for, a special operator is used to dereference a pointer to a structure and access a member all in one go:

    pointer_to_structure->membername

  thus

```
coordinates->x = 88;
```

The TYPEDEF statement

- The typedef statement provides a way to “alias” an existing data type to a different name. This is most commonly used with pointers as seen below
- Syntax: `typedef` *`datatype aliasname`*
- Example

  ```
  typedef char *MyString
  ```

- The preceding example will allow the declaration of strings, as pointers to char, like this:

  ```
  MyString somestring;
  ```

- Typedefs just add a new name for an existing data type. They do not create *new* data types in any sense.

`typedef` and `struct`

- `typedef` is frequently used with structures to easily declare and use pointers to those structures, thusly:

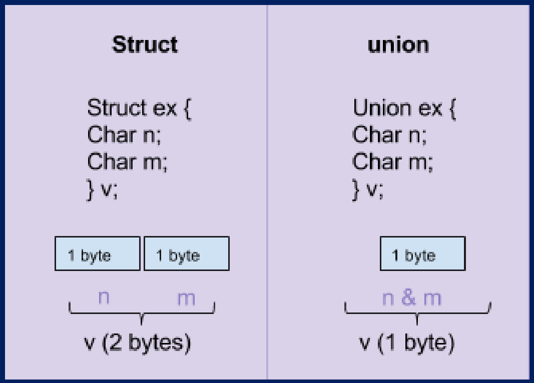
```
typedef struct point {int x; int y;} Coordinates;
typedef Coordinates *2D_Coordinates;
```

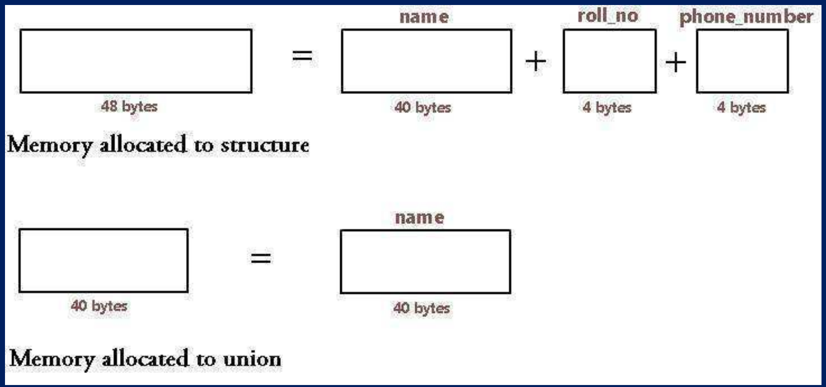
- The previous code will first declare `Coordinates` as an “alias” for `struct point`. This “alias” is then used to declare the variable `2D_Coordinates` as a pointer to data type `struct point`.

The `union` declaration

- The statement union is used to declare overlapping member variables. That is, the memory storage for the members in a union overlap each other. This can be a powerful way to treat multiple variables as a single “block” of data storage. The syntax is nearly identical to `struct`:
- Syntax: `union` *`structurename`*

```
{
    member1 type member1 name;
    member2 type member2 name;
}
```





`union` and layered data structures

- IP addresses, which are used to uniquely identify hosts on a computer network, consist of four integers. In fact, these integers are bytes, and can only hold values from 0-255. Separating the individual bytes with periods, to form a "dotted quad", is what tells us we're looking at an IP address:
  172.12.9.223 or 192.168.127.252

- Internally, IP addresses are handled by nearly all computers as 4-byte integers. However, at the human/user level, we prefer to use those dotted quads.

- Unions are terribly useful for manipulating these:

```
struct dotquad { byte quad1; byte quad2; byte quad3; byte quad4; };
union ipaddress
{
  dotquad ipaddr;
  uint32_t integer_address;
}
```

- Note the separate declaration of `dotquad` to hold the individual bytes. This is necessary to ensure that the bytes of the IP address are stored *sequentially* in memory. If the bytes were declared "bare" in the union, then they would *all overlap each other* as well as the *first* byte of `integer_address`. This is an example of a layered or *hierarchical* data structure with more than one "level".

Storage Classes

- **Auto** – Default storage class for variables in functions. Used to declare dynamic allocation for the variable, in Arduino IDE compiler, does nothing, and is rarely if ever used.
- **Register** – Tells the compiler to store the variable in a microcontroller register rather than memory. This allows faster access and processing of the variable data, and thus faster execution. The compiler does this automagically, so it's rarely used.
- **Static** - variables declared in functions "die" when you exit the function in which they are defined. This means that each time the function is called, a new set of these variables is created. This also means that any values for the variables in the function from the previous execution of the function's code are lost. The `static` storage class prevents this from happening and allows the variable contents to "persist" across function calls. You could also use a global variable to do the same thing, as all globals are static.
- **Extern** - is a storage class modifier that tells the compiler that the variable is defined in a different source code file but let's me use it in the current file as a variable of whatever type. Used in heavy-duty coding with multiple source files in a single project, which is Advanced Programming so not covered here.
- Syntax: [*storage-class*] *data-type variable-name*
  Example: `static int totalpixels;`

NOTE: The avr-objdump.exe program and documentation exist in the Tools directory for dumping AVR object files to inspect the generated code. Waaaaay beyond this class!!!

Formal End of Lesson 2

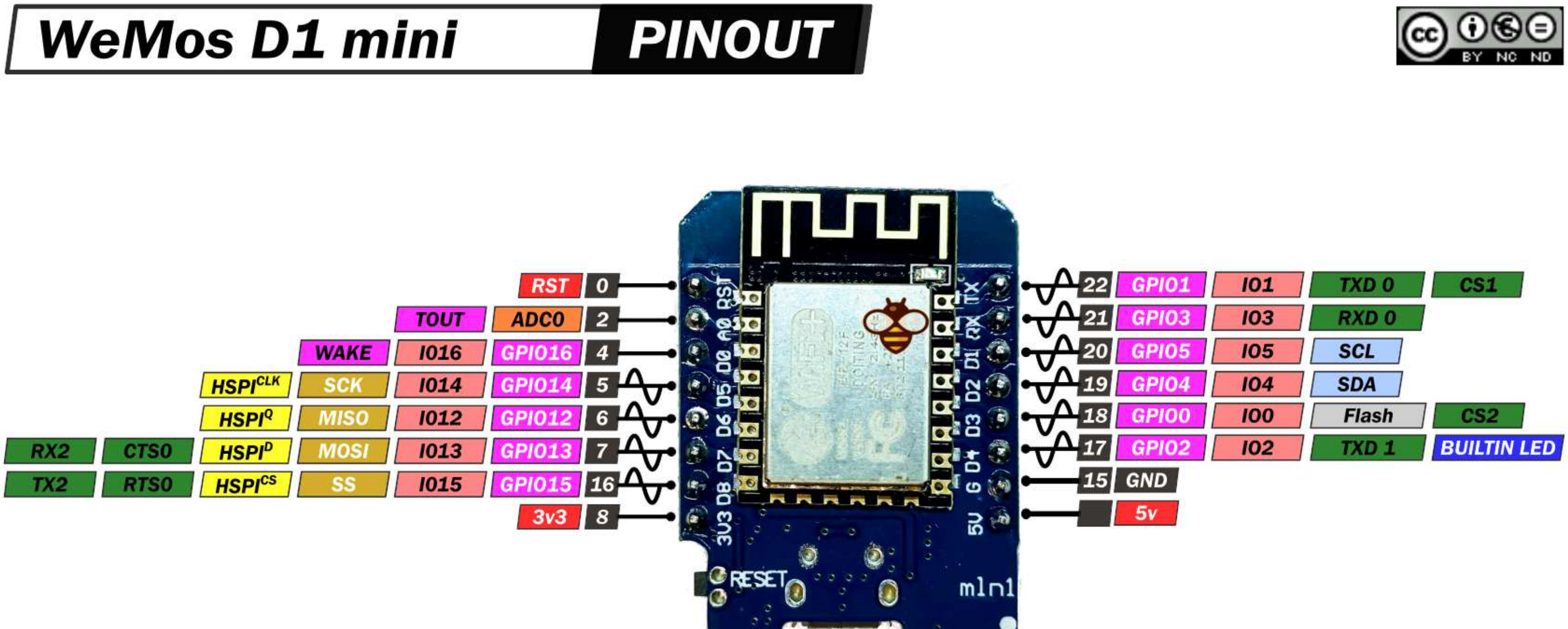
HOMEWORK

- **Have a look at the new HackSpace magazines added to your flash drives!**

In next week's exciting episode

- Class Exercise: Using complex data structures
- Introduction to linked lists
- Linked list usage example
- Bitmap font architecture review
- 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

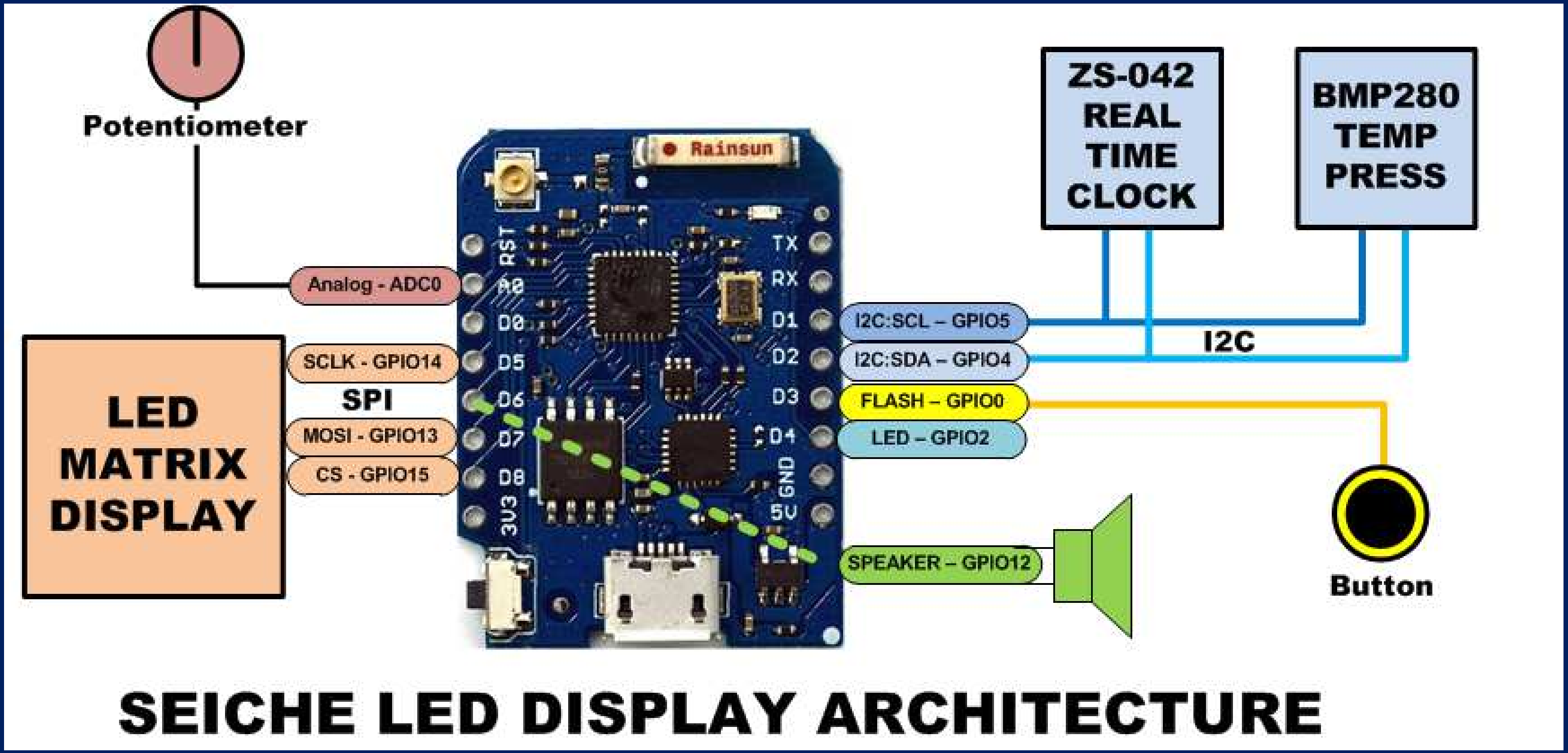
Our Microcontroller: The WeMos D1 Mini



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