



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>
- 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

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 everything else!

FLASH DRIVE

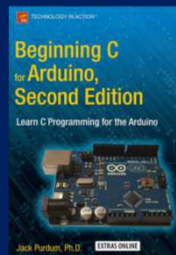
IntermediateProgramming-Software

- Kubuntu
 - `code_1.75.0-1675266613_amd64.deb`
- Windows
 - `VSCodeSetup-x64-1.75.0.exe`
 - `VSCodeUserSetup-x64-1.75.0`

IntermediateProgramming-Lessons

- IntermediateProgramming-Lesson1
- IntermediateProgramming-Lesson2
- **IntermediateProgramming-Lesson3**

Can't do reading homework without the files!



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

IntermediateProgramming-Documentation

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

Lesson 3 – MD_Parola Font Usage Review

Part A

- Sprintf review
- Bitmap fonts review
- MD_Parola font usage review

Part B

- The MD_Parola font format
- Designing your own fonts
- Declaring your own fonts
- Using your own fonts with MD_Parola

sprintf() Review

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

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

- sprintf() conversion specifiers have the following general syntax (all begin with a percent-sign):

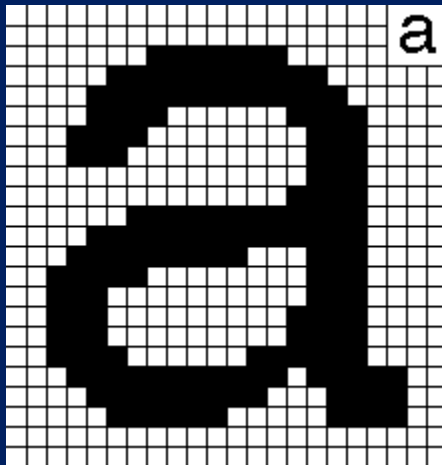
%[flags][minimum field width][.][precision][length][conversion character]

- % - special token that indicates the start of a conversion specifier
- Flags – these modify the behavior of the specification
- Minimum 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

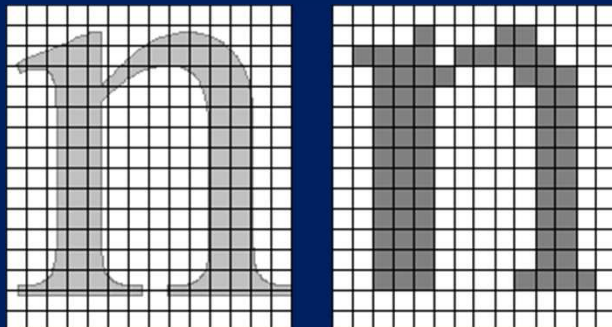
A summary sprintf() reference can be found at the end of each lesson

Review of Fonts - Bitmap

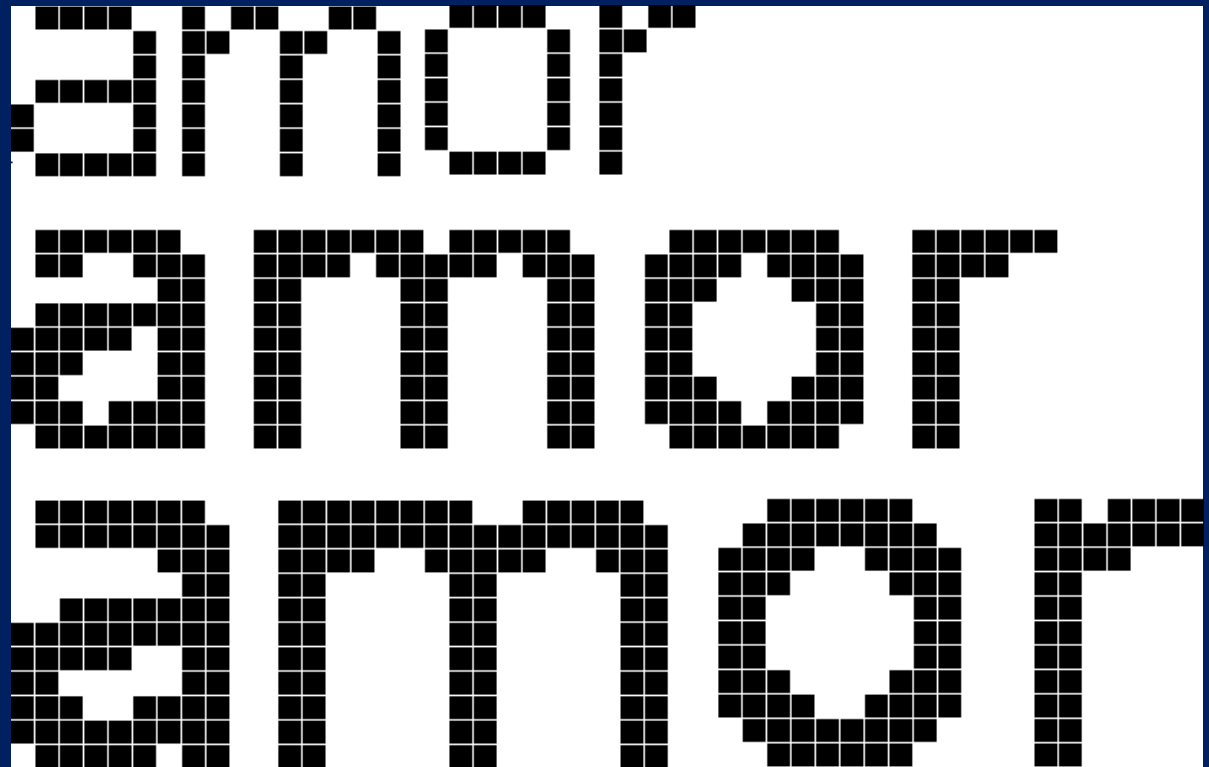
- Bitmap fonts specify which pixels in the character are on or off
- As a result, bitmap fonts are fixed in size, and not rescalable
- While outline fonts are normally sized by *points*, bitmaps are often sized by *pixels*



A 23x22 bitmap character



Comparison: outline vs 14x16 bitmap



A word in different bitmap sizes

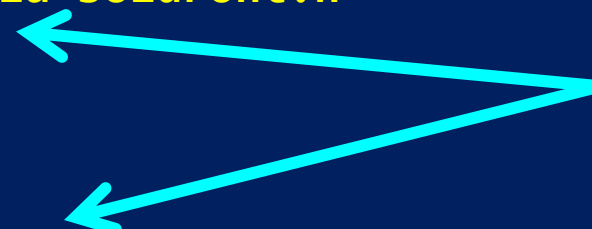
Classroom Exercise

Using Different Fonts

- Load SKETCH3A and upload to your display
- There are a number of font header files in your SKETCH3A directory; have a look at their filenames in your file explorer.
- Now modify your sketch to use a *different* font than the one already specified in SKETCH3A and re-upload to your display

```
#include <SPI.h>
#include <MD_MAX72xx.h>
#include <MD_Parola.h>
#include "Parola-boldFont.h"
...
pmx.setFont(parola_boldFont);
```

Note that the font header file begins with a capital “P” and the font name itself begins with a lowercase “p”!



MD_Parola Font Details

<https://arduinoplusplus.wordpress.com/2016/11/08/parola-fonts-a-to-z-defining-fonts/>

- The design of the MD_Parola/MD_MAX72xx libraries results in the font management code being located in the MD_MAX72xx library. This allows code written for MD_MAX72xx to easily use text and frees the MD_Parola library from (mostly) needing to know where and how characters are stored.
- At a basic level the font is just a byte data table in memory. The table is stored as a series of contiguous bytes, in a repeating pattern for each of the 256 ASCII characters. Each character bitmap is stored in the following format:
 - **byte 0** is the number of column bytes (n) that form this character (could be zero if the character is not defined).
 - **byte 1.. n** – one byte for each column of the character. Byte 1 is the leftmost column of the character. The most significant bit of each data byte is the bottom pixel position of the character matrix (ie, bit 7 is row 7 of the matrix).
- If a particular ASCII character value is not defined, the size byte is 0. An empty data table is therefore represented by 256 zero bytes.

MD_Parola Font Details (cont.)

An example of the standard font in the library shows how the code formatting can be used to logically separate the defined characters even though the data is physically a sequential byte stream:

```
5, 0x30, 0x38, 0x3e, 0x38, 0x30, // 30 - 'Up Pointer'  
5, 0x06, 0x0e, 0x3e, 0x0e, 0x06, // 31 - 'Down Pointer'  
2, 0x00, 0x00, // 32 - 'Space'  
1, 0x5f, // 33 - '!'  
3, 0x07, 0x00, 0x07, // 34 - '"'  
5, 0x14, 0x7f, 0x14, 0x7f, 0x14, // 35 - '#'
```

To find a character in the font table, the library does a sequential search by looking at the first byte (the size n), skips $n+1$ bytes to the next character's size byte, repeating until the last or target character is reached.

Double height fonts are stored in one font definition table by restricting the range of displayable ASCII character to 0..127. The lower half of each character is located at ASCII codes 0 through 127 and the upper part is located in ASCII codes 128 through 255, offset 128 positions from the lower half. For example, character 'A' is stored in positions 65, the normal location for a single height character, and 193 (65 + 128).

Creating MD_Parola Fonts

- Creating font files manually is a tedious process, so the MD_MAX72xx library is supplied with two utilities to ease the pain.
- **txt2font** is a command line utility that processes a text definition of the font data file. This is implemented in C with source code supplied, allowing users on any operating system to compile a version to work with their OS.
- **FontBuilder** is a GUI utility implemented as VBA in Microsoft Excel. As FontBuilder requires Microsoft Office products, it does not work environments where these are not available. [So we will not be using this in class]
- For both utilities, several font definition files are supplied as examples or starting points for your own fonts.
- **UPDATE 16 Nov 2018:** Arduino forum user pjrj has created a web based to define and manage font data. Use it from [here](#) and you can access his code repository [here](#).
<https://pjrj.github.io/MDParolaFontEditor>

The TXT2FONT Utility

The txt2font utility is a command line application that converts a structured text definition of the font into a data file in the right format for MD_MAX72xx to use. The utility is supplied as an Win32 executable and source code for other OS.

The application is invoked from the command line and only the root name of the file is given as a command line parameter (eg “txt2font fred”). The application will look for an input file with a “.txt” extension (“fred.txt”) and produce an output file with a “.h” extension (“fred.h”).

The txt2font file format is line based. Lines starting with a ‘.’ are directives for the application. All other lines direct the current character definition. An example of the beginning of a font definition file is shown and explained below.

```
.NAME sys_var_single
.HEIGHT 1
.WIDTH 0
.CHAR 0
.NOTE 'Empty Cell'
.CHAR 1
.NOTE 'Sad Smiley'
@@@
@@@@@
@ @ @
@@@@@
@@ @@
@  @
@@@
.CHAR 2
.NOTE 'Happy Smiley'
```

The directives have the following meaning:

.NAME defines the name for the font and is used in naming the font table variable. The name can appear anywhere in the file. If omitted, a default name is used. This directive should appear once at the start of a file.

.HEIGHT defines the height for the font. Single height fonts are ‘1’ and double height fonts are ‘2’. If omitted, the application assumes single height font. This directive should appear once at the start of a file.

.WIDTH specifies the width of the font for all the characters defined between this WIDTH and the next WIDTH definition. Zero denotes variable (or unspecified) width; any other number defines the fixed width. WIDTH may be changed within the file – for example to define a fixed size space (where no pixels can be drawn!) character in a variable width font.

.CHAR ends the definition of the current character and starts the definition for the specified ASCII value. Valid parameters are 0..255 for single height, and 0..127 for double height. If a character code is omitted in the font definition file it is assumed to be empty.

.NOTE is an option note that will be added as a comment for the entry in the font data table.

Any lines not starting with a ‘.’ are data lines for the current character. The font characters are drawn using a non-space character (eg, ‘*’ or ‘@’) wherever a LED needs to be ‘on’. The application scans from the top down, so any lines missing at the bottom of the character definition are assumed to be blank. However, blank lines at the top need to be shown. Any extra rows are ignored will cause program errors.

The Online Utility

<https://pjrp.github.io/MDParolaFontEditor>

The online PJRP MD_MAX72XX/MD_Parola font utility allows for both the creation of entirely new fonts and the editing of existing fonts

You can set the width of individual characters and also specify double-height characters

The screenshot shows the MD_MAX72XX FONT EDITOR web interface. At the top, there's a navigation bar with a back arrow, a home icon, and the URL pjrp.github.io/MDParolaFontEditor. Below the navigation bar, there's a toolbar with buttons for "# 32 SPACE", "Prev", "Next", "Width 8", "- +", "Fit", "Save", "Clear", and "Delete". To the right of the toolbar, the text "MD_MAX72XX FONT EDITOR" is displayed. Below the toolbar, there's a "Label" input field and a text area containing "8, 0, 0, 0, 0, 0, 0, 0, // 32". To the right of the text area is a grid of 64 circles. Below the grid, there's a section with "Import" and "Export" buttons, a "Name" input field containing "newFont", and a "DOUBLE HEIGHT" checkbox. To the right of these controls is a large grid of 256 character slots, each with a number and a character. The grid is organized into 16 rows and 16 columns. The first row contains characters from 0 to 20, the second row from 21 to 41, and so on, up to the 16th row which contains characters from 147 to 167. The characters are arranged in a standard ASCII-like sequence.

The Online Utility

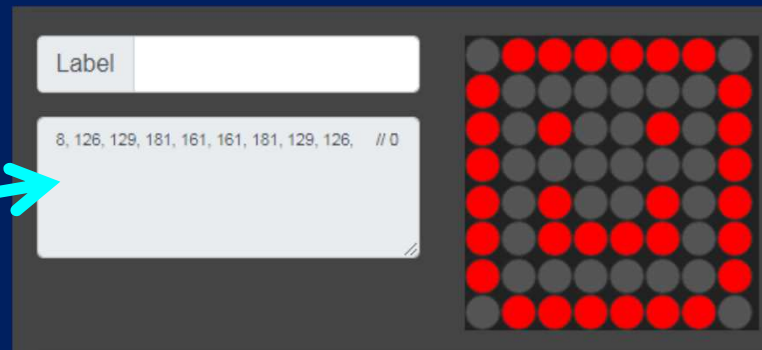
<https://pjrp.github.io/MDParolaFontEditor>

The dot-matrix “character designer” portion of the online utility is extremely useful; not only does it allow one to directly “draw” any font character, but once drawn, the decimal equivalent column specifiers are listed in the text window to the left.

This means that even if you are not going to use the utility to build an entire font, you can use the character designer to design a single character, then place the code for that single character directly into your sketch in such a way as to overwrite one of the character locations in an existing font (and more on that in a moment.)

Thus you can “supplement” a font of your choice with characters of your choice. If you have the header file (.h) for the font, you can make the modification directly in the header file prior to compile time. Since any ASCII value between 0 and 255 can be redefined, there are usually very low or very high characters that have been left undefined (empty) and can be repurposed for your own use.

Decimal
column
numbers are
shown in the
text box



Managing MD_Parola Fonts

<https://arduinoplusplus.wordpress.com/2016/11/13/parola-fonts-a-to-z-managing-fonts/>

The Parola library is designed to efficiently manage text displayed on a LED matrix display. On the other hand, the MD_MAX72xx library provides the hardware management primitives to control the hardware. This means that the MD_MAX72xx library should naturally contain the most basic methods for managing fonts and Parola provides 'higher order' methods.

Fonts in MD_MAX72XX

Font data is stored in PROGMEM, so it cannot be actually be 'closed' or 'deleted'. However, to allow font substitution, the libraries store a separate pointer to the current font table. This pointer can be changed to point to different tables in a very dynamic manner, enabling fonts to be efficiently changed during code execution. A NULL pointer resets the font to the system default – as no font definition would cause execution issues, this is co-opted to mean the default font.

```
bool MD_MAX72XX::setFont(fontType_t * f)
{
    _fontData = (f == NULL ? _sysfont_var : f);
    buildFontIndex();

    return(true);
}
```

Managing MD_Parola Fonts (cont.)

<https://arduinoplusplus.wordpress.com/2016/11/13/parola-fonts-a-to-z-managing-fonts/>

- **Fonts in MD_Parola**

Font parameters can be set individually for each zone or for the whole display. The discussion below is for the zone functions. To extend the same parameters into the whole display, analogous methods set identical parameters for all zones.

- The Parola library uses the MD_MAX72xx primitives to display standard fonts and also adds a few functions that extend the functionality of fonts.
- Specifying a new font simply stores a pointer to the required table for later setting in the MD_MAX72xx. As each zone can have a different font, the font table needs to be swapped just-in-time as the messages for each zone are displayed – MD_MAX72xx only knows about the 'current' font. The specified font needs to be defined in the user application and conform to the font table format described in [Part 1](#)

```
void setZoneFont(MD_MAX72XX::fontType_t *fontDef)
{
    _fontDef = fontDef;
};
```

Parola extends the functionality of fonts by allowing individual character in a font to be substituted at run time. This is useful when specific characters (for example, a combined '°C' character) are needed but the majority of the standard font is suitable. An interesting side note is that the character with ASCII code 0 ('\0' in C format) can never be processed by the library as it denotes the end of a string, not a displayable character.

Managing MD_Parola Fonts (cont.)

<https://arduinoplusplus.wordpress.com/2016/11/13/parola-fonts-a-to-z-managing-fonts/>

The replacement characters are stored in a linked list per zone with memory allocated from the heap. To minimize heap fragmentation the memory for deleted characters is retained for re-use at the next definition. In practice, once characters are defined they are not usually deleted. Also, to minimize RAM requirements, the library does not copy the in the data in the data definition but only retains a pointer to the data, so any changes to, or release of, the data storage in the calling program will be reflected in the library.

The character data must be the same format as those in the font table. If a character is specified with a code the same as an existing character the existing data is substituted for the new data.

Clearly, once we have the ability to define a character in the Parola library, any search for a specific character needs to check the local tables before it defaults to the specified font file. As for the MD_MAX72xx libraries, the data is loaded into a user buffer and the length in columns is returned from the function.

```
bool MD_PZone::addChar(uint8_t code, uint8_t *data)
{
    charDef *pcd;

    if (code == 0)
        return(false);

    // first see if we have the code in our list
    pcd = _userChars;
    while (pcd != NULL)
    {
        if (pcd->code == code)
        {
            pcd->data = data;
            return(true);
        }
        pcd = pcd->next;
    }

    // Now see if we have an empty slot in our list
    pcd = _userChars;
    while (pcd != NULL)
    {
        if (pcd->code == 0)
        {
            pcd->code = code;
            pcd->data = data;
            return(true);
        }
        pcd = pcd->next;
    }

    // default is to add a new node to the front of the list
    if ((pcd = new charDef) != NULL)
    {
        pcd->code = code;
        pcd->data = data;
        pcd->next = _userChars;
        _userChars = pcd;
    }

    return(pcd != NULL);
}
```

Formal End of Lesson 2

HOMEWORK

- **Read Chapters 7-9 in *Beginning C for Arduino***
(Sorry, Ch. 7 should have been on last week's assignment ☹)
- **Have a look at the new HackSpace magazines added to your flash drives! (if you forgot to copy them!)**
- **Check the Internet for articles or videos on Visual Studio Code**

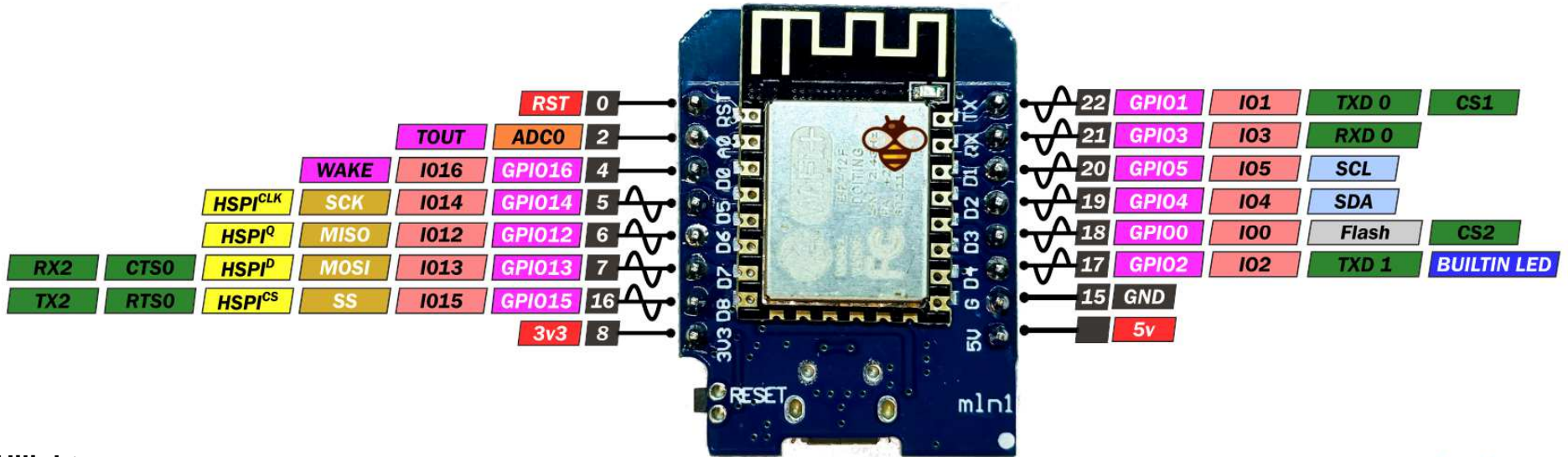
In next week's exciting episode

- Video – Installing Visual Studio Code
- Video – Using Visual Studio Code with Arduino
- Class Exercise – Installation of Visual Studio Code
- Instructor's time permitting – More on Parola font usage

Our Microcontroller: The WeMos D1 Mini

WeMos D1 mini **PINOUT**

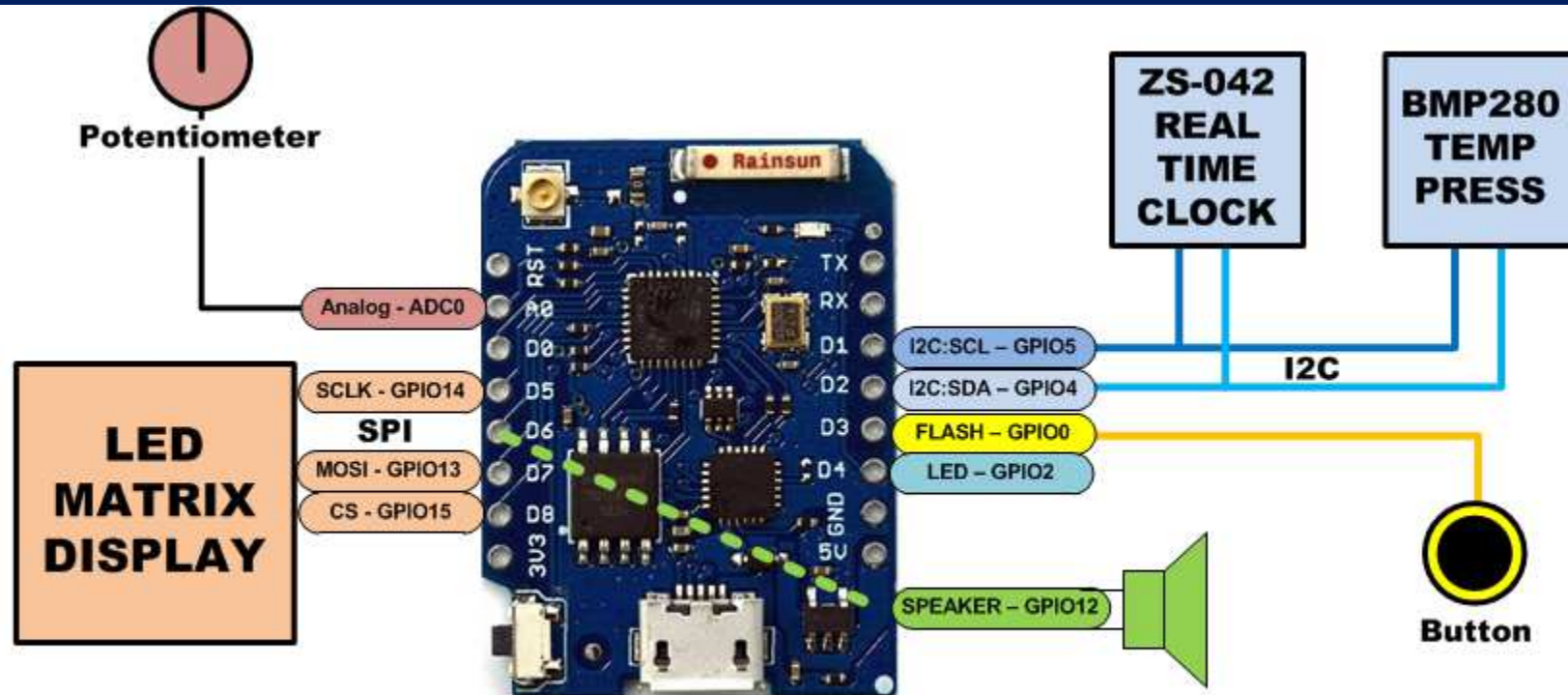
WeMos D1 mini **PINOUT**



Highlights

- **11 digital IO: all are interrupt and pwm capable (except D0/GPIO16)**
- **1 analog input (3.2V max input): A0/ADC0**
- **Micro USB or 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



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)
- 10K Ω 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() automatically 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
- **Minimum 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 <i>precision</i> 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
<i>spc</i> (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